Lecture Notes in Computer Science 1536
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Willem-Paul de Roever
Hans Langmaack Amir Pnueli (Eds.)

Compositionality: The Significant Difference

International Symposium, COMPOS'97
Bad Malente, Germany, September 8-12, 1997
Revised Lectures

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Willem-Paul de Roever
Hans Langmaack
Christian Albrechts University at Kiel
Department of Informatics and Applied Mathematics
Preußerstr. 1-9, D-24105 Kiel, Germany
E-mail: {wpr,hl}@informatik.uni-kiel.de

Amir Pnueli
Weizmann Institute of Science
Department of Applied Mathematics and Computer Science
P.O. Box 26, Rehovot 76100, Israel
E-mail: amir@wisdom.weizmann.ac.il

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Compositionality : the significant difference ; international symposium ;
revised lectures / COMPOS '97, Bad Malente, Germany, September 8 - 12, 1997.
Willem-Paul de Roever ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1536)
ISBN 3-540-65493-3

CR Subject Classification (1998): F.3.1, D.2.4, F.4.1, I.2.3

ISSN 0302-9743
ISBN 3-540-65493-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1998
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10692972 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

There is a growing awareness that most specification and verification methods have reached their limits. Model-checking (in spite of the striking progress due to symbolic methods) can check finite-state systems up to a certain size, and deductive methods, due to the heavy user interaction required, can handle only systems of small complexity. If there is hope that industrial-size designs can be handled by formal methods, it must be based on the two premises of *compositionality* and *abstraction*.

Under compositionality, we include any method by which the properties of a system can be inferred from properties of its constituents, without additional information about the internal structure of these constituents. The two main questions that have to be addressed in forming a viable compositional theory are:

- How to decompose a global specification into local specifications which will be satisfied by the individual components?
- Having shown that the local specifications are satisfied by their respective components, how to infer the global specification from this?

One of the recently suggested methods for combining algorithmic (model-checking) with deductive (proof-theoretic) verification methods is by the use of compositional methods, in which the local specifications are verified by algorithmic methods and the specification decomposition and re-composition are done using deductive technology.

This particular suggestion, as well as many others, was considered at the Symposium "Compositionality: The Significant Difference" (COMPOS'97), organized at Hotel Intermar, Bad Malente, Germany, September 8–12, 1997. The idea for organizing this symposium was suggested by Ben Moszkowski.

The present volume constitutes the proceedings of this symposium. It reflects the current state-of-the-art in compositional reasoning about concurrency. Apart from the contributions written by the speakers, this volume also contains a contribution by Mads Dam. In order to put all those contributions into proper perspective, one of the organizers, W.-P. de Roever, has written a survey for these proceedings describing the main issues in compositional reasoning and the history of their evaluation, as reflected in the current literature.

We gratefully acknowledge the financial support for this symposium by a grant from the Deutsche Forschungsgemeinschaft DFG (grant no. 4851/225/97), by a donation from one of the organizers, H. Langmaack, and by the Dutch "Stichting AFM" headed by J. Vytopil.

The local organization was in the able hands of Änne Straßner. We express our genuine gratitude to her for her efforts.

Last but not least, we would like to thank the speakers for their active and responsive participation, for giving such excellent talks, and for putting so much effort in writing their contributions. This made this symposium a memorable event not only for its participants but also for its organizers.

September 1998 W.-P. de Roever, H. Langmaack, A. Pnueli

Contents

The Need for Compositional Proof Systems: A Survey 1
Willem-Paul de Roever

Alternating-Time Temporal Logic 23
Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman

Compositionality in Dataflow Synchronous Languages: Specification and Code Generation 61
Albert Benveniste, Paul Le Guernic, and Pascal Aubry

Compositional Reasoning in Model Checking 81
Sergey Berezin, Sérgio Campos, and Edmund M. Clarke

Modeling Urgency in Timed Systems 103
Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis

Compositional Refinement of Interactive Systems Modelled by Relations .. 130
Manfred Broy

Toward Parametric Verification of Open Distributed Systems 150
Mads Dam, Lars-åke Fredlund, and Dilian Gurov

A Compositional Real-Time Semantics of STATEMATE Designs 186
Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli

Deductive Verification of Modular Systems 239
Bernd Finkbeiner, Zohar Manna, and Henny B. Sipma

Compositional Verification of Real-Time Applications 276
Jozef Hooman

Compositional Proofs for Concurrent Objects 301
Jerry James, Ambuj Singh

An Overview of Compositional Translations 327
Theo M.V. Janssen

Compositional Verification of Multi-Agent Systems:
A Formal Analysis of Pro-activeness and Reactiveness 350
Catholijn M. Jonker, Jan Treur

Modular Model Checking 381
Orna Kupferman, Moshe Y. Vardi

Composition: A Way to Make Proofs Harder 402
Leslie Lamport

Compositionality Criteria for Defining Mixed-Styles Synchronous Languages 424
Florence Maraninchi, Yann Rémond

Compositional Reasoning Using Interval Temporal Logic and Tempura ... 439
Ben C. Moszkowski

Decomposing Real-Time Specifications 465
Ernst-Rüdiger Olderog, Henning Dierks

On the Combination of Synchronous Languages 490
Axel Poigné, Leszek Holenderski

Compositional Verification of Randomized Distributed Algorithms 515
Roberto Segala

Lazy Compositional Verification 541
Natarajan Shankar

Compositional Reasoning Using the Assumption–Commitment Paradigm . 565
Qiwen Xu, Mohalik Swarup

An Adequate First Order Interval Logic 584
Chaochen Zhou, Michael R. Hansen

Compositional Transformational Design for Concurrent Programs 609
Job Zwiers

Compositional Proof Methods for Concurrency: A Semantic Approach 632
Frank S. de Boer, Willem-Paul de Roever

Author Index 647

The Need for Compositional Proof Systems: A Survey

Willem-Paul de Roever

Christian Albrechts University at Kiel
Institut für Informatik, Preußerstraße 1–9
D-24105 Kiel, Germany
wpr@informatik.uni-kiel.de

Abstract. A survey is given of the main issues in compositional reasoning about state-based parallelism and of the history of their evolution, as reflected in the current literature. Compositional proof techniques are presented as the proof-theoretical analogue of Dijkstra's hierarchically-structured program development. Machine-support for compositional reasoning, and the relationship between compositionality and modularity are discussed. The issues when compositional reasoning about concurrency is successful, and when it isn't, are commented upon. Pointers to the other papers in this volume are provided.

1 Introduction

1.1 What this paper is about

The paper in front of you presents a survey of the main issues and trends in state-based compositional reasoning about concurrency, and the history of their evolution, as reflected in the current literature. It includes a list of references to this subject, as well as pointers to the other papers in this volume. It is written as introduction for the proceedings of the International Symposium COMPOS '97.

These proceedings reflect the current state of the art in state-based compositional reasoning about concurrency.

1.2 Structure of this paper

Section 1.3 gives a brief account of the development of state-based program verification and compositional reasoning, up to the point where the contributions in this volume take over. Namely, the fundamental results by Misra & Chandy [MC81] and Jones [Jon81, Jon83], presenting the first compositional proof rules for, respectively, synchronous message passing and shared variable concurrency. Section 2 discusses the verify-while-develop paradigm, and section 3 its relationship with compositional reasoning, leading to the conclusion that compositional proof techniques can be viewed as the proof-theoretical analogue of Dijkstra's hierarchically-structured program development. The assumption-commitment and rely-guarantee paradigms are the subject of section 4, and machine-support

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 1-22, 1998.
Springer-Verlag Berlin Heidelberg 1998

for compositional verification that of section 5. Section 6 discusses the relationship between compositionality and modularity, section 7 represents an attempt at demystifying of the complexity of compositional reasoning, and section 8 comments briefly upon the issue when compositional reasoning is successful. A conclusion and a list of references to the subject complete the paper.

1.3 Stages in the history of program verification

Within the context of predicate logic, the principle of compositionality was already formulated in 1923 by G. Frege [Fre23]. The first ones to relate Frege's notion of compositionality to logics for program proving were P. van Emde Boas and T.M.V. Janssen in [JvEB80]. This principle and its application to specifying compiler correctness are the themes of Janssen's contribution to this volume.

Interestingly, when one follows the development of program proving methods (which started in 1949 with a publication by A. Turing [Tur49]), the main direction is from a-posteriori nonstructured program proving methods to structured compositional methods [dR85, HdR86].

For sequential programs this development is easy to trace. First R.W. Floyd's noncompositional method appeared in 1967 [Flo67] (a year earlier, P. Naur developed in [Nau66] related ideas). His method, called the *inductive assertion method*, is based on labelled transition diagrams – directed graphs whose edges are labelled with guarded multiple assignments – whose nodes are associated with boolean state functions. The resulting *assertion network* is called *inductive,* whenever for every pair of nodes $\langle l, l' \rangle$ in a diagram, which is connected by a directed edge, the corresponding pair $\langle \mathcal{Q}_l, \mathcal{Q}_{l'} \rangle$ of associated boolean state functions satisfies the so-called *verification condition* associated with that edge. This verification condition expresses that whenever $\mathcal{Q}_l$ is satisfied in some state, and the transition associated with that edge is taken, the resulting state satisfies $\mathcal{Q}_{l'}$. This method is noncompositional because there is "no room", so to speak, for program development between a specification and the atomic parts of a transition diagram. As we will see, this situation changes when transition diagrams are provided with an algebraic structure, s.t. they, as well as their associated inductive assertion networks, can be decomposed into their components. To use an analogy: in Floyd's original approach a house is considered to consist of bricks, beams, tubes, electric wires, glass etc., rather than of a roof, walls, floors, plumbing, power supply and the like. Clearly, when developing programs from their specification one needs, mutatis mutandis, the latter higher level of abstraction.

Floyd's method was cast in an axiomatic compositional style by C.A.R. Hoare for sequential programs in 1969 [Hoa69]. Hoare observed that programs had a syntactic structure, and that this structure could also be given to their correctness proofs for purposes of program development.

His paper is the first one that defines a programming language in terms of how its programs can be proved correct with respect to their specifications rather than in terms of how they are executed. That is, it reduces proving every specification for some composed program construct – in this case obtained by

application of the sequential composition, if-then-else and while operators – to proving specifications for its constituent operands, without the need for any further information whatsoever. And this is what compositionality is all about.

Also in 1969, Edsger W. Dijkstra wrote the following paragraph in [Dij69a]:

> *"On Understanding Programs.*
>
> *On a number of occasions I have stated the requirement that if we ever want to be able to compose really large programs reliably, we need a discipline such that the intellectual effort E (measured in some loose sense) needed to understand a program does not grow more rapidly than proportional to the program length L (measured in an equally loose sense) and that if the best we can attain is a growth of E proportional to, say, L^2, we had better admit defeat. As an aside I used to express my fear that many programs were written in such a fashion that the functional dependence was more like an exponential growth!"*

Since the complexity of compositional reasoning grows linearly w.r.t. the program size (as will be discussed later), this paragraph is the first reference within the computer science literature to the desirability of compositional reasoning. The above quotation is very general. A related argument, but then focusing on concurrency, appears in [Dij72, especially on page 75].

It is not our intention here to give a full account of the rich development of verification methods for sequential program constructs.

For concurrent and distributed programs proof methods turned out to be difficult to develop due to the problem of how to formalize the, in general, close interaction between the separate processes while they are executing. For certain types of programs, such as operating systems, this close interaction is even their one and only purpose.

For shared variable concurrency, the first Hoare-like proof system was due to Owicki & Gries in 1976 [OG76]. Technically this feat was performed by ramifying the notion of specification by Hoare triples by means of the introduction of proof outlines. Proof outlines are systematically annotated program texts in which at every control location a predicate is inserted, which characterizes the program state at that point. By requiring that these predicates remain invariant under the execution of assignments in other processes – as expressed by the so-called *interference freedom test* – the influence of concurrent operations on shared variables is mathematically captured. Because proof outlines annotate the program text, they use additional information about the underlying execution mechanism, and therefore any logic based on them is noncompositional.

For distributed synchronous communication such proof systems were independently discovered by two teams: Apt, Francez & de Roever [AFdR80] and Levin & Gries [LG81]. Here, synchronous communication was captured proof-theoretically by the so-called *cooperation test*. This test also operates on proof outlines for the various processes which constitute a program. It requires essentially that for communicating pairs of input-output actions – the actual fact of communication is characterized by a global invariant, basically over the communication histories of the various processes – the conjunction of the two predicates

associated by the respective proof outlines with the control locations immediately before these input and output actions implies after communication (which is mathematically expressed as a distributed assignment) the two predicates associated with the locations immediately after these two actions. Since only the input action can change the state of program variables, in practice this test involves checking that the input values have been appropriately specified in the postcondition of an input action w.r.t. the particular environment in question. Also this method is noncompositional.

A third development was that of temporal logic by A. Pnueli in 1977, introduced originally as a logic for formal reasoning about the various so-called fairness requirements needed to implement various kinds of semaphores in mutual exclusion algorithms [Pnu77]. Also temporal logic is noncompositional.

After the development of these first-generation proof principles for almost every programming construct in existence, the stage was set for the next leap forward: The formulation of compositional proof methods for concurrency, thus enabling, as we shall see, the extension of Dijkstra's paradigm of hierarchically--structured program development to the systematic derivation of concurrent programs.

All the proof methods and systems for concurrency which are discussed above apply to *closed* systems only, i.e., systems without any communication with processes outside of them. These derive their interest from the fact that one typically models the environment as a separate component, and then considers the system *composed* with its environment, which constitutes a closed system. However, that presupposes a *fixed* environment. And, for purposes of *reusability* one does not want to have this. That leads to consideration of *open* (i.e., nonclosed) systems, their specification, and their correctness.

But how can one specify an open system which is intended to interact via, e.g., certain prespecified shared variables? Certainly without knowing its ultimate environment such a system cannot be verified using the method of Owicki & Gries, or any other theory mentioned above, because one doesn't know on beforehand which interference to expect.

A solution to that problem was given for shared-variable concurrency by Cliff Jones in [Jon81, Jon83] and, for distributed communication by Jay Misra and Mani Chandy in [MC81].

Jones proposed in his so-called *rely-guarantee* formalism *to specify the interference allowed by the environment of a component during its execution without endangering fulfillment of the purpose of that component.* The proposal of Misra & Chandy is similar, be it that it applies to synchronous communication, and is called the *assumption-commitment* (A/C) formalism.

Both formalisms are compositional, because the purpose of a compositional verification approach is to shift the burden of verification from the global level to the local, component, level, so that global properties are established by composing together independently (specified and) verified component properties. And this is exactly what's needed to specify open systems, for which one doesn't know which environment will be provided.

At this point these proceedings take over. More details about the A/C paradigm can be found in section 4.

Acknowledgments

Ben Moszkowski suggested organizing a symposium on compositionality to me, and, therefore, created the opportunity for writing this paper. My co-organizers are thanked for their help in converting Ben's idea into reality. Amir Pnueli gave his support in drafting the call for papers, and helped with the composition of a representative list of lecturers, whose work reflects the state of the art in compositional reasoning about concurrency. Hans Langmaack undertook the arduous task of raising money at a time when money was tight, and even generously provided from his own personal funds! Änne Straßner did an excellent job as local organizer of this symposium.

Ben Lukoschus meticulously composed the LaTeX script of many versions of this paper. Michael Siegel made helpful comments.

Finally, I would like to take the opportunity to warmly thank Reinhard Wilhelm and Angelika Mueller from the Dagstuhl office of the University of Saarbrücken, and, especially, the team at Schloß Dagstuhl, consisting of Dietmar Kunzler, Melanie Spang, Nicole Paulus, Petra Meyer, Christel Franz, Peter Schneider and Josephine Franzen, for creating the cordial and relaxed atmosphere in which this paper was written.

2 The verify-while-develop paradigm

Most of the literature on program correctness deals with *a-posteriori* program verification, i.e., the situation that *after* the program has been obtained it is proved correct, see, for instance [Man74, And91, AO91, Dah92, Fra92, Sch97]. Although a-posteriori verification has been indispensable for the development of a truly mathematical theory of program correctness, it is a frustrating approach in practice, because almost all programs contain errors. Discovering that error then leads to the need for correcting the program. And the so corrected program contains most of the time errors again. Nevertheless one hopes that this is no nonterminating cycle. Hence the attempt at a correctness proof mostly leads to the discovery of errors, i.e., a counterexample violating the original specification, rather than a successful proof.

This situation becomes even more grave when considering concurrent programs and systems. For then the number of cases to be verified when checking the correctness of the interaction between their individual processes grows in general exponential w.r.t. the number of those processes. Consequently, checking them all becomes well-neigh impossible, except, maybe, when this process of checking has been mechanized. But even then the limits of naive machine verification are quickly reached.

It would make sense, if, rather than disproving finished programs over and over again in the course of attempting to prove them correct, one could be

convinced of the correctness of a design decision *at the very moment one took that decision* during the process of top-down development of a program from its specifications – a paradigm called *"verify-while-develop"*. This paradigm requires one to verify that a program-in-the-making meets its specification by making use of specifications for its constituent parts or modules *while the implementing program text of those parts is not yet available.* This has the advantage that, since every top-down development step now requires a correctness proof based on the specifications of those parts (rather than their programming text), any errors made during development of those parts according to these specifications *do not influence the correctness of this particular development step.* Consequently, no redundant verification work is done when following this paradigm, in contrast to the situation with a-posteriori program verification.

Of course, the verify-while-develop strategy is not new. It is merely a reformulation of the notion of *hierarchically-structured program development,* advocated by Dijkstra and others since the end of the sixties [Dij68, Dij76, Dij82].

3 Compositionality

The technical property which is required of program correctness methods for supporting the verify-while-develop paradigm is called *"compositionality":*

> "That a program meets its specification should be verified on the basis of specifications of its constituent components only, without additional knowledge of the interior construction of those components."

To make this verification strategy possible, programs and their parts are specified using predicates *only.* Such specifications are called *assertional.* Now the verification that a program satisfies its assertional specification *should be entirely carried out at the level of reasoning about these predicates*, i.e., *no additional knowledge about the underlying execution mechanism of its parts is allowed.*

To be precise, compositional verification that a program P satisfies an assertional specification φ amounts to application of the following *recursive* proof strategy:

1. In case P cannot be further decomposed, compositional verification that P satisfies φ is done directly, i.e., either by application of an axiom, or by a proof based on the semantics of P and φ. Hence this amounts to a traditional verification proof.
2. In case P is composed of parts $P_1, \ldots, P_n$, e.g., P is a network $P_1 \| \ldots \| P_n$ with "$\|$" expressing some form of parallel composition, compositional verification that P satisfies φ amounts to executing the following steps:
   (a) Find assertional specifications $\varphi_1, \ldots, \varphi_n$ for $P_1, \ldots, P_n$ such that steps (b) and (c) below can be proved.
   (b) *Prove that P satisfies φ whenever P is composed of parts P_i satisfying φ_i, $i = 1, \ldots, n$.* This proof consists of checking the validity of an implication involving the specifications $\varphi_1, \ldots, \varphi_n$ and φ, only, and is called a *compositional proof step.*

(c) Recursive application of this proof strategy to compositional verification that P_i satisfies φ_i, $i = 1, \ldots, n$.

Since the syntax tree of P is finite, this process of proof decomposition terminates, because it is syntax directed. For the incorporation of recursive procedures into this proof strategy, we refer to the contributions of Dam, Fredlund and Gurov, and of Finkbeiner, Manna and Sipma, to this volume.

How exactly are such compositional frameworks obtained? First of all, for every composition operator op^P in the programming language (e.g., sequential composition ";", and parallel composition "||") there should exist an operator op^S in the specification language s.t.:

> *Whenever programming constructs P_i satisfy specifications φ_i, for $i = 1, \ldots, n$, one also has for every n-ary operator op^P that $op^P(P_1, \ldots, P_n)$ satisfies $op^S(\varphi_1, \ldots, \varphi_n)$.*

Secondly, it is required that:

> *Whenever $op^P(P_1, \ldots, P_n)$ satisfies specification φ, there exist specifications φ_i for P_i s.t. $P_i \models \varphi_i$, $i = 1, \ldots, n$, and $\models op^S(\varphi_1, \ldots, \varphi_n) \rightarrow \varphi$ holds, i.e., $op^S(\varphi_1, \ldots, \varphi_n) \rightarrow \varphi$ is valid in the interpretation concerned.*

These two properties imply that steps (a), (b) and (c) above can be carried out.

Compositional reasoning owes its attractiveness to its application to parallel composition, since it replaces operational reasoning with a complexity *increasing exponentially* in the number of parallel components by reasoning compositionally on the basis of given specifications, and the complexity of this way of reasoning increases *linearly* w.r.t. the number of those specifications. This is discussed later in section 7.

Compositional proof techniques have the advantage that they allow a systematic top-down development of programs from their specification, which is correct by construction, as illustrated in contributions of Broy, of Hooman and of Olderog and Dierks to this volume. And this process of program derivation is exactly the *verify-while-develop paradigm!* For, as a consequence of using compositional techniques, each compositional proof (reduction) step (as defined above) can be viewed as *the verification of a design step*, which only involves reasoning about (the assertional specification of) that particular step and does not involve any future design steps. This explains why in the definition of compositional proof techniques above we stipulate that no additional knowledge about the underlying execution mechanism of the constituent components (of the to-be-verified program) is allowed. For, without that clause, reasoning about a particular design step might involve reasoning about future design steps, and we want these two stages to be independent.

This explains why compositional techniques can be viewed as the proof-theoretical analogue of Dijkstra's hierarchically structured program development.

This point of view is further worked out in Broy's contribution to this volume; Broy proves the compositionality of his mathematical model w.r.t. three forms of refinement relations. In the contributions of Hooman and of Olderog and Dierks, this viewport is extended to real-time systems; specifically, the latter prove that real-time systems specified in a certain subset of the Duration Calculus can be decomposed into an untimed system with suitable timers, and the former illustrates how to specify real-time systems in a platform-independent way.

Finally, observe that nowhere in the account above assertional specifications are required to be syntactically expressed in some logic language. They can also be boolean-valued state functions. In fact, the above characterization of compositionality can be extended as well to transition diagrams with parallel composition and, in case of nested concurrency, sequential composition as additional operations used for expressing the structure of those diagrams. Now compositional verification can be formulated equally well in a semantic framework based on transition diagrams and inductive assertion networks, by extending the notion of verification condition in such a way that independence of any influence by the environment is obtained (e.g., in case of input transitions, these conditions should hold regardless of the particular input value). This is worked out in the contribution of de Boer and me to this volume, and clarifies, in my opinion, one of the points raised in Lamport's contribution. In fact, this semantic approach to compositionality is in our opinion very close to the one which is used by Hooman, Shankar and others, in order to define within PVS a semantic basis for machine-supported compositional reasoning.

4 Specification and verification of open systems: the assumption-commitment paradigm

In the context of a-posteriori program verification one tends to focus on proof methods and systems for concurrency which apply to *closed* systems only, i.e., systems without any communication with processes outside of them. Obviously, as remarked above, for the purpose of *reusability* the specification, development and verification of *open* systems is far more important.

But how can one specify an open system which is intended to interact via, e.g., certain prespecified shared variables without knowing that interaction on beforehand? Observe that the answer to this question is consistent with the very purpose of compositional reasoning, since, because the environment of an open system is not known, *it can only be specified without giving any implementation details about that environment,* and this is exactly what compositional reasoning is about.

A solution to that problem was suggested by Cliff Jones in [Jon81, Jon83] for shared variable concurrency, and, for synchronous communication by Jay Misra and Mani Chandy in [MC81].

Jones proposed in his so-called *rely-guarantee* formalism *to specify the interference allowed by the environment of a component during its execution without*

endangering fulfillment of the purpose of that component. Here the rely condition expresses the interference allowed, and the guarantee condition the task to be performed, by that component. The proposal of Misra & Chandy applies to synchronous communication, is called the *assumption-commitment* (A/C) formalism, and embodies a similar kind of specification strategy. Although the rely-guarantee and assumption-commitment paradigms were originally developed independently, it was later realized that the induction principles reflected by their associated parallel composition rules are similar [XCC94]. We shall adhere, therefore, to the name assumption-commitment for both formalisms, because the A/C paradigm was published earlier.

Both formalisms are compositional, because the purpose of a compositional verification approach is to shift the burden of verification from the global level to the local, component, level, so that global properties are established by composing together independently (specified and) verified component properties.

Of all the approaches to the compositional verification of concurrency, this *assumption-commitment* paradigm is the one studied best [MC81, Jon83, Oss83, Sou84, Pnu84, ZdBdR84, BK85, Sta85, ZdRvEB85, GdR86, Lar87, Sti88, Zwi89, Jos90, Hoo91, PJ91, XH91, AL93, AP93, Col93, Jos93, KR93, Col94, CMP94, DH94, XCC94, AL95, DJS95, XdRH97]. Manna and Pnueli incorporate similar ideas in section 4.3 of [MP95] for proving invariance properties in a setting of temporal logic; additional references to compositional approaches in that setting are [MCS82, NDOG86, CMP94, Jon94].

The formulation of the A/C paradigm for invariance properties is simpler than for (the combination of invariance and) liveness properties expressed in temporal logic. The reader is referred to the contributions of Shankar and of Xu and Swarup to this volume for an explanation why this is the case and how this problem is resolved. Suffice to say here that, in case of combinations of invariance and liveness properties, by a result of [AS85, Sch87] these properties can be written as the conjunction of a maximal invariance and minimal liveness property, to which separate forms of A/C reasoning are applied [AL91]. This is worked out in, e.g., [MCS82, AL93, AP93, Col94, CMP94, AL95].

The contribution of Xu and Swarup extends this paradigm to real-time. The contribution of Kupfermann and Vardi to this volume investigates the complexity of the A/C paradigm in the context of model-checking, for different combinations of linear time and branching time logics used for expressing the separate specifications of assumption and commitment.

The specification and proof of open distributed systems using a new *alternating-time* temporal logic is proposed in the contribution of Alur, Henzinger and Kupfermann to this volume. The same topic is addressed in the contribution of Dam, Fredlund and Gurov, using a temporal logic based on a first-order extension of the modal μ-calculus for the specification of component behavior.

5 Compositionality and machine-supported verification

The promise of the previously discussed joint verify-while-develop paradigm/ compositional proof-generation strategy is based on its potential for application in the large, as witnessed by the growing list of academic and industrial claims in this respect [Lar87, LT87, CLM89, Jos90, KR90, LT91, HRdR92, LS92, Bro93, Jos93, KL93, CMP94, DH94, GL94, Jon94, Mos94, DJS95, Hoo95, Cau96, KV96, SY96, BLO98a, BLO98b, DJHP98, Sha98]. This is mainly due to the effort made in recent years towards machine-support of this form of reasoning. This effort is caused by the following factors.

As we have seen, compositional methods enable reasoning about complex systems because they reduce properties of complex systems to properties of their components which can be checked independently of their environment. Now the same applies to reasoning about complex *finite state* systems. E.g., verification approaches based on the fully automatic technique of *model checking* fail to scale up gracefully because the global state space that has to be explored can grow exponentially in the number of components [GL94]. Also here compositional verification offers a solution by restricting the model checking procedure to verifying local, component level properties, and using combinations of interactive and machine-based theorem proving for verifying that the machine-verified local properties imply the desired global property [CLM89, Jos90, Jos93, Kle93, GL94, DJS95, KV96, Sha98].

This is discussed in the contributions of Alur, Henzinger and Kupfermann, of Damm, Josko, Hungar and Pnueli, and of Berezin, Campos and Clarke to this volume.

Secondly, complex systems, and especially safety critical ones, are in great need of formal verification. Also, the formal verification of crucial components of such systems has acquired higher industrial priority ever since INTEL lost an estimated 500 million US Dollars due to a bug in its Pentium chip. Now compositional proof methods are reported to allow such verification tasks in practice for medium-size examples. Also in case of the verification of *infinite state systems*, it makes therefore sense to complement the above mentioned combination of automatic and theorem-proving-based verification techniques by compositional reasoning supported by semi-automated proof checking methods such as, e.g., PVS [ORSvH95], as reported in [Hoo95, HvR97, Sha98], and other methods, for which we refer to the contributions of Hooman, and of Shankar to this volume. As already stated, Hooman and Shankar encode a *semantical* characterization of compositional verification in PVS, which is similar to the one discussed in the contribution by de Boer and me.

6 Compositionality, completeness and modularity

In specification, the compositionality principle implies a *separation of concerns* between the use of (and reasoning about) a module, and its implementation. As

Lamport says [Lam83], it provides a *contract* between programmer and implementor:

- One can program a task by the use of modules of which one only knows the specification *without any knowledge about their implementation,* and
- one can implement a module solely on the basis of satisfying its specification *without any knowledge of its use.*

Thus one separates *programming with modules on the basis of their specifications* from *implementing a module such that its specification is satisfied.*

To succeed in this contract between programmer and implementor, all aspects of program execution which are required to define the meaning of a construct must be explicitly addressed in semantics and assertion language alike. Hence all assumptions which are needed regarding the environments in which a module will function – because these influence the behavior of that module – must be made explicit as *parameters* (in both the semantics and the specification of that module), for only then one may abstract from the remaining aspects. This is especially required when specifying fixed components, which can be taken ready-made from the shelf, so to say.

In order to understand the relationship between the compositionality principle and Lamport's view of a specification as a contract between programmer and implementor, in [Zwi89, ZHLdR95] the terminology regarding compositionality is refined by distinguishing the *compositionality* property, which is needed when developing a program in top-down fashion from its specification, from the property called *modularity* (also called *modular completeness* in [Zwi89]), which is needed when constructing programs from re-usable parts with fixed proven specifications in bottom-up fashion. Similar distinctions are made in the work of Abadi and Lamport [AL93, AL95].

Thus, *compositionality* refers to the *top-down approach*, stating that for every specification of a compound construct *there exist* specifications for its parts such that those specifications imply the original specification without further information regarding the implementation of these parts. The principle of *modularity* refers to the *bottom-up approach*, and requires that, whenever a property of a compound statement follows semantically from the *a priori given specifications of its fixed components,* this should always be deducible in the proof system in question. As argued in [Zwi89] this amounts to requiring compositionality plus a complete, so-called, *adaptation* rule.

The simplest know adaptation rule is the consequence rule, familiar from, e.g., Hoare and temporal logics. In general, given the fact that a certain *given* property φ has been proved of a *given* component C, adaptation rules state how to "plug" this proof into a proof context in which another property, say ψ, of that component C is needed. (The formulation of such rules, e.g., for assumption-commitment-based formalisms, is far from trivial [ZHLdR95], and still represents a veritable research effort.) Completeness of such a rule amounts to the property that, whenever some requested property ψ of C is semantically implied by the already established property φ of C, then the adaptation rule can be applied, resulting in a proof that component C satisfies the required property ψ.

Consequently, also when programming solely on the basis of given, fixed, specifications, a compositional framework is required in order to convince oneself of the correctness of one's programs in a formal set-up which has both the potential for application in the large and allows modules to be specified so as to provide a contract between programmer and implementor in Lamport's sense above.

The relationship between compositional proof systems, completeness and modularity is further discussed in a paper by Trakhtenbrot [Tra].

The paper by Finkbeiner, Manna and Sipma in this volume presents a formalism for modular specification and deductive verification of parameterized fair transition systems, with an extension to recursion.

7 The complexity of compositional reasoning

Take an automaton with 10 states and consider the parallel composition of 80 of such automata. This results in a product automaton of 10^{80} states – that is, more states than the number of electrons within our universe!

Consider now the Internet. Certainly only 80 computers operating in parallel is chicken feed; many many more are operating in parallel! As we saw above, the complexity of the product automaton is *exponential* in the number states of the constituting automata. Consequently, a description of the Internet using an enumeration of in principle possible states is completely irrealistic.

This example shows that an analysis using product automata may not be the appropriate one to analyze the parallel composition of programs (although it sets lower bounds for worst-case complexity). After all, also human beings operate in parallel, and do not perceive their human environment as being composed of a product automaton recording the changes inside that environment.

Using compositional specification methods, the parallel composition of processes can be specified using the *conjunction* of the specification of the separate processes. Hence the complexity of this description is *linear*, rather than exponential, in the number of those specifications.

Next, assume that inside these specifications *disjunctions* are used to list various possibilities, and the conditions subject to which these arise. Then, using de Morgan's law about the distribution of disjunctions over conjunctions such a linear description may still, in the worst case, lead to the same exponential number of separate cases to be considered. So what did we gain?

Return to the analogy with man, above. What other (wo)men observe are not the state changes inside their fellow human beings but rather changes in their *(inter)faces* or attitudes, from which a lot of local internal changes have been eliminated.

Similarly, compositional specifications record changes in interfaces between programs (or processes), i.e., the change of externally (i.e. to other programs) visible quantities, and not the change in internal quantities, s.a. local variables, communication along local channels and the like. Also, as already observed,

compositional specifications express the *conditions* under which such external changes occur.

Now, de Morgan's law still applies w.r.t. distributing disjunctions inside (compositional) specifications over conjunctions between specifications. However, due to the fact that compositional specifications also express the conditions under which the externally visible changes occur, in case the interaction between processes is not too tight, only *a manageable amount of such combinations of externally visible state changes* (of the separate processes) needs to be considered, i.e., are *consistent* – one hopes at least so few that the number of consistent combinations has a manageable complexity. And this belief is supported by current programming practice.

The assumption behind this belief is that communicating computers, operating in parallel, are invented by man, and, since mankind cannot cope with exponential complexity, any man-made artifact *which works*, and consists of parallel components, must somehow be possible to characterize using a very much lower than exponential complexity in the number of processes.

Combining the observations made so far leads to the conclusion that, in general, compositional reasoning is successful for correctness-preserving top-down derivation of concurrent programs, where processes do not interact too tightly. However, there are also programs to which this assumption does not immediately apply, e.g., Chandy & Misra's solution to the "Drinking-Philosophers Problem" [CM84] or the distributed computation of greatest common divisors, in which some form of global property must be maintained which doesn't easily lend itself to decomposition. Then compositional reasoning does not help, and does not lead to clearer proofs. The issue when compositional reasoning is successful, and when it isn't, will be further discussed in the next section.

Consequently, when considering program verification, it is certainly the case that compositional reasoning plays an important rôle, especially in case of top-down development (or reconstruction). But this does not detract from the need to develop other manageable verification techniques, which improve the complexity of the reasoning process in case of programs whose development requires application of other kinds of principles. Examples of such noncompositional verification methods and development principles, s.a. the communication closed layers principle, can be found in the contribution of Zwiers to this volume.

8 When is compositional reasoning successful?

In which areas of verification is compositional reasoning successful?

The main focus of this volume is on compositional techniques for reasoning about parallelism. Therefore we shall discuss how well various language concepts combine with parallelism, from the point of view of obtaining a successful compositional characterization.

Before doing so, one should realize that the issue here is not which language features can be characterized compositionally and which cannot, because *all*

language constructs can be characterized in a compositional way. This can be seen as follows.

The meaning of every programming construct can be defined by giving its denotational semantics. In a denotational semantics the meaning of a composed construct is a (mathematical) function of the meaning of its components. (The distinction between denotational and compositional semantics is the use of fixed-point operations in the former.) But this implies that *all* the information required for reasoning about that composed construct is already obtainable from the meanings of its components. Therefore, the very point of compositional reasoning – that it can be carried out within a single uniform formalisms – is met. Thus, in principle, the language of mathematics can be used as assertion language for reasoning compositionally about *every* programming construct, on the basis of the denotational semantics of that construct.

However, *the measure of success of a compositional characterization is its simplicity.* And this forces one to proceed in a more subtle manner, when looking for a really successful compositional characterization.

The first issue in obtaining a compositional characterization of a language feature or construct is which observables must be additionally introduced in order to make its meaning a function of its parts. E.g., when reasoning compositionally about real-time, not only the time at which actions occur must be recorded as an event in the semantics, but also the time required for waiting for synchronization. (Of course, this analysis is also required in the "denotational" approach sketched above.) Then one should make sure to use the coarsest compositional semantics which is consistent with the observables chosen, and which still distinguishes observably different constructs. I.e., one tries to obtain a, so-called, *fully abstract* semantics. On the basis of this information, the simplest possible logic is designed for reasoning about this semantics, which distinguishes programming constructs if, and only if, those constructs have a different meaning in this semantics. In this logic every semantical function associated with a programming construct should be represented by a corresponding logical operator, which has the logical characterization of the operands of that function as arguments.

Details about this can be found in my forthcoming textbook [dRdBH$^+$].

The simplest denotational semantics of a parallel operator is obtained by intersecting the semantics of its arguments. Consequently, the simplest logical characterization of that operator is conjunction. Such a characterization applies to synchronous communication, and to synchronous languages s.a., for instance, LUSTRE, ESTEREL, SIGNAL, and TLA (the former three synchronous languages were originally designed for the description of real-time embedded systems). No wonder that many papers in this volume – those by Benveniste, Le Guernic and Aubry, by Berezin, Campos and Clarke, by Lamport, by Maraninchi and Rémond, and by Poigné and Holenderski – are based on that paradigm, which is now getting successfully established in industry. Since hardware operates synchronously (as Gérard Berry has relentlessly been pointing out), also compositional reasoning about hardware is the natural thing to do, as ev-

idenced by the contribution of Berezin et al. Benveniste et al. describe models and techniques for *distributed* code generation for synchronous languages on not necessarily asynchronous architectures; Berezin et al. describe and illustrate several compositional model checking techniques used in practice; Maraninchi et al. study *mixing* ARGOS and LUSTRE inside the formalism of *mode-automata*, and discuss three criteria for compositionality which are used in selecting an appropriate semantics for the resulting mixed language; Poigné and Holenderski provide a *generic framework* for synchronous programming and its implementation, called the SYNCHRONIE WORKBENCH. Finally, Lamport considers compositional reasoning to be necessary for open systems, and very useful in case of machine-supported verification. He continues by arguing that "when distracting language issues are removed and the underlying mathematics is revealed, compositionality is of little use," advocating the use of ordinary mathematics for "semantic reasoning," instead. This is close to the position, taken by de Boer and me, that the essence of compositionality is revealed when language issues are removed and the underlying mathematics is exposed. What then remains is, in Lamport's case, temporal-logic-based reasoning about the next-state relation in the context of TLA, and, in our case, reasoning about transition diagrams in a suitable inductive assertion framework formulated inside mathematics.

On the basis of the synchronous paradigm, simple compositional characterizations of real-time have been obtained, with an impressive number of applications. Such characterizations are discussed in the contributions by Moszkowski, by Hooman, by Olderog and Dierks, by Bornot, Sifakis and Tripakis, by Xu and Swarup, and by Zhou and Hansen. The latter contribution extends compositional reasoning in the interval temporal logic ITL to continuous time, and, in principle, hybrid systems. The contribution by Bornot et al. argues that many different ways exist for composing time progress conditions in the context of timed automata, and that these are all practically relevant. Moszkowski's contribution also discusses compositional proof rules for liveness properties, and deals with comparing different granularities of time, all in the context of a simulator for the automatic analysis of specifications, called Tempura.

Shared-variable concurrency requires a more complex semantics in order to obtain a compositional characterization, which corresponds to the much tighter possible interaction between shared-variable processes. That semantics, discovered by Peter Aczel in 1983, of which a fully abstract version was developed in [dBKPR91] called *reactive-sequence* semantics, provides an underlying model for Jones' rely-guarantee formalism. The corresponding logical operator for reasoning about shared-variable concurrency additionally requires substitution and renaming. To my knowledge, few applications of the corresponding logics exist, although compositional correctness proofs for some mutual exclusion algorithms have been given [dBHdR97a, dBHdR97b].

A special position is taken by the compositional semantics of Statecharts, which is the subject of the contribution by Damm, Josko, Hungar and Pnueli. Since parallel composition in Statecharts is only synchronous at the level of its micro-step semantics, but not purely synchronous for its super-step semantics,

defining its compositional semantics requires quite an effort. This semantics acts as reference model for a verification tool currently under development allowing to verify temporal properties of embedded control systems modelled using the Statemate system; this tool combines symbolic model checking with assumption-commitment-based reasoning.

Also object orientation is the subject of recent compositional characterizations. Consult, e.g., the contributions by Mads Dam, Fredlund and Gurov, and by James and Singh. Since object-oriented constructs are characterized by their sophisticated information-hiding techniques, progress in this field can only be expected once the localization of information within objects is adequately reflected by their associated logics [dB98].

Finally, compositional verification methods are extended to multi-agents in the contribution by Jonker and Treur, and to randomization in the contribution by Segala.

9 Conclusion

A survey has been given of the main issues and directions in state-based compositional reasoning about parallelism. Compositional proof techniques are presented as proof-theoretical analogue of Dijkstra's hierarchically structured program development. Machine-support for compositional reasoning has been discussed, as well as the relationship between compositionality, modularity and completeness. Compositional reasoning is considered successful in case of:

- correctness-preserving top-down derivation of concurrent programs, whose processes do not interact too tightly,
- synchronous communication and real-time, and for synchronous languages.

The history of the subject has been briefly discussed, and pointers to the papers in this volume have been given.

Real progress in the (semi-)automated verification of programs can only be expected once specification formalisms are supported which are *both* compositional w.r.t. parallelism *and w.r.t. abstraction/refinement.* For these are the two main mental tools which man possesses for tackling complexity: reduction-to-smaller-problems and abstraction.

Broy's and Lamport's contributions to this volume testify that they have realized this already for a long time. A temporal logic plus associated proof rules which enables compositional and stutter-invariant reasoning about both parallelism and refinement has been developed in [Cau96]. And in [BLO98a, BLO98b] the InVeSt tool is reported which supports the verification of invariance properties of infinite-state systems; this tool computes abstractions of such systems compositionally and automatically, and then uses pre-fixed-points to find auxiliary invariants using a combination of algorithmic and deductive techniques.

There's still lots of work to do!

Schloß Dagstuhl, September 10, 1998.

References

[AFdR80] K.R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[AL91] Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AL93] Martin Abadi and Leslie Lamport. Comparing specifications. *Toplas*, 15(1):73–132, 1993.

[AL95] Martin Abadi and Leslie Lamport. Conjoining specifications. *Toplas*, 17(3):507–534, May 1995.

[And91] Gregory R. Andrews. *Concurrent Programming, Principles and Practice.* The Benjamin/Cummings Publishing Company, 1991.

[AO91] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[AP93] Martin Abadi and Gordon D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[BK85] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In *Proc. of a Seminar on Concurrency*, LNCS 197. Springer-Verlag, 1985.

[BLO98a] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[BLO98b] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In *CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[Bro93] Manfred Broy. Interaction refinement—the easy way. In M. Broy, editor, *Program Design Calculi*, volume 118. Springer NATO ASI Series, Series F: Computer and System Sciences, 1993.

[Cau96] Antonio Cau. Compositional verification and specification of refinement for reactive systems in a dense time temporal logic. Technical Report Bericht Nr. 9601, Institut für Informatik und Praktische Mathematik, University of Kiel, 1996.

[CLM89] Ed M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proc. LICS '89*, pages 353–362. IEEE Computer Society Press, 1989.

[CM84] K.M. Chandy and J. Misra. The drinking-philosophers problem. *TOPLAS*, 6(4):632–646, 1984.

[CMP94] Edward Chang, Zohar Manna, and Amir Pnueli. Compositional verification of real-time systems. In *Proc LICS '94*. IEEE Computer Society Press, 1994.

[Col93] Pierre Collette. Application of the composition principle to UNITY-like specifications. In *Proc. of TAPSOFT '93*, LNCS 668. Springer-Verlag, 1993.

[Col94] Pierre Collette. An explanatary presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50(1):31–35, 1994.

[Dah92] Ole-Johan Dahl. *Verifiable Programming.* Prentice Hall, 1992.

[dB98] F.S. de Boer. Reasoning about asynchronous communication in dynamically evolving object structures. In *CONCUR '98*, volume 1466 of *LNCS*. Springer-Verlag, 1998.

[dBHdR97a] F.S. de Boer, U. Hannemann, and W.-P. de Roever. A compositional proof system for shared-variable concurrency. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *LNCS*, pages 515–532, Berlin, Heidelberg, New York, 1997. Springer Verlag.

[dBHdR97b] F.S. de Boer, U. Hannemann, and W.-P. de Roever. Hoare-style compositional proof systems for reactive shared variable concurrency. In *FSTTCS '97: Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, Berlin, Heidelberg, New York, 1997. Springer Verlag.

[dBKPR91] F.S. de Boer, J.N. Kok, C. Palamedessi, and J.J.M.M. Rutten. The failure of failures: towards a paradigm for asynchronous communication. In Baeten and Groote, editors, *CONCUR'91*, LNCS 527. Springer-Verlag, 1991.

[DH94] Werner Damm and Johannes Helbig. Linking visual formalisms: A compositional proof system for statecharts based on symbolic timing diagrams. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94). North-Holland*, pages 337–356, 1994.

[Dij68] E.W. Dijkstra. The structure of the "THE" multiprogramming system. *CACM*, 11(5):341–346, 1968.

[Dij69a] E.W. Dijkstra. EWD 264. Published in an extended version as [Dij69b], August 1969.

[Dij69b] E.W. Dijkstra. Structured programming. In J.N. Buxton and B. Randell, editors, *Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee*, pages 84–88. NATO Science Committee, 1969.

[Dij72] E.W. Dijkstra. Hierarchical ordering of sequential processes. In C.A.R. Hoare and R.H. Perrot, editors, *Operating Systems Techniques*, pages 72–98, London and New York, 1972. Academia Press. Proceedings of a seminar held at Queen's University, Belfast, 1971.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

[Dij82] E.W. Dijkstra. *Selecting Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982.

[DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In *Proceedings of the International Symposium COMPOS '97.* Springer-Verlag, 1998.

[DJS95] W. Damm, B. Josko, and R. Schlör. Specification and verification of vhdl-based system-level hardware designs. In E. Börger, editor, *Specification and Validation Methods*, pages 331–410. Oxford University Press, 1995.

[dR85] Willem-Paul de Roever. The quest for compositionality - a survey of assertion-based proof systems for concurrent programs, part 1: Concurrency based on shared variables. In *Proc. of IFIP Working Conf, The Role of Abstract Models in Computer Science, North-Holland*, 1985.

[dRdBH+] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Fron Noncompositional to Compositional Proof Methods.* Submitted for publication in 1998.

[Flo67] R.W. Floyd. Assigning meanings to programs. In *Proceedings AMS Symp. Applied Mathematics*, volume 19, pages 19–31, Providence, R.I., 1967. American Mathematical Society.

[Fra92] Nissim Francez. *Program Verification.* Addison-Wesley, Wokingham, 1992.

[Fre23] G. Frege. *Gedankengefüge, Beiträge zur Philosophie des Deutschen Idealismus*, volume Band III, pp. 36/51. 1923. Translation: Coumpound Thoughts, in P. Geach & N. Black (eds.), Logical Investigations, Blackwells, Oxford, 1977.

[GdR86] R.T. Gerth and W.-P. de Roever. Proving monitors revisited: A first step towards verifying object-oriented systems. *Fundamenta Informatica, North-Holland*, IX:371–400, 1986.

[GL94] Orna Grumberg and David Long. Model checking and modular verification. *Toplas*, 16(3):843–871, 1994.

[HdR86] J. Hooman and W.-P. de Roever. The quest goes on: towards compositional proof systems for CSP. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, LNCS 224, pages 343–395. Springer-Verlag, 1986.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580,583, 1969.

[Hoo91] J. Hooman. *Specification and compositional verification of real-time systems.* LNCS 558. Springer-Verlag, 1991.

[Hoo95] Jozef Hooman. Verifying part of the access.bus protocol using pvs. In *Proc. of* 15^{th} *FSTTCS*, LNCS 1026. Springer-Verlag, 1995.

[HRdR92] J. Hooman, S. Ramesh, and W.-P. de Roever. A compositional axiomatization of statecharts. *Theoretical Computer Science*, 101:289–335, 1992.

[HvR97] J. Hooman and O. van Roosmalen. Platform-independent verification of real-time programs. In *Proc. of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 183–192. IEEE Computer Society Press, 1997.

[Jon81] C.B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University Computing Laboratory, 1981.

[Jon83] C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Jon94] Bengt Jonsson. Compositional specification and verification of distributed systems. *Toplas*, 16(2):259–303, March 1994.

[Jos90] B. Josko. Verifying the correctness of AADL-modules using model checking. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430. Springer-Verlag, 1990.

[Jos93] Bernhard Josko. *Modular Specification and Verification of Reactive Systems.* Habilitationsschrift, Universität Oldenburg, 1993.

[JvEB80] T.M.V. Janssen and P. van Emde-Boas. The impact of Frege's compositionality principle for the semantics of programming and natural languages. In D. Alexander, editor, *Proc. of the First Frege memorial conference, May '79, Jena*, pages 110–129. Friedrich-Schiller Universität Jena, May 1980. previously as preprint, Report MI-UVA-79-07, Mathematisch Centrum, Amste rdam, 1979.

[KL93] Rob P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In *Computer-Aided Verification, Proc. of the 5^{th} Int. Conf. CAV '94*, LNCS 697, pages 166–174. Springer-Verlag, Berlin, Heidelberg, New-York, 1993.

[Kle93] S. Kleuker. Case study: Stepwise development of a communication processor using trace logic. In Andrews, Groote, and Middelburg, editors, *Proc. of the International Workshop on Semantics of Specification Languages SoSL*, Utrecht, 1993.

[KR90] A. Kay and J.N. Reed. A specification of a telephone exchange in timed CSP. Technical Report PRG-TR-19-90, Oxford University Programming Research Group, 1990.

[KR93] A. Kay and J.N. Reed. A rely and guarantee method for timed CSP. *IEEE Transactions on Software Engineering*, 19(6), 1993.

[KV96] Orna Kupfermann and Moshe Y. Vardi. Module checking. In *Proc. of CAV '96*, LNCS 1102. Springer-Verlag, 1996.

[Lam83] L. Lamport. What good is temporal logic. In R.E.A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congres*, pages 657–668, Paris, September 1983. IFIP, North-Holland.

[Lar87] Kim G. Larsen. A context-dependent bisimulation between processes. *Theoretical Computer Science*, 49, 1987.

[LG81] G.M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, 15:281–302, 1981.

[LS92] Kim G. Larsen and Arne Skou. Compositional verification of probabilistic processes. In W.R. Cleaveland, editor, *Proc. of CONCUR '92*, LNCS 630. Springer-Verlag, 1992.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. PoDC '87*, ACM, New York, 1987.

[LT91] Kim G. Larsen and Bent Thomsen. Partial specifications and compositional specification. *Theoretical Computer Science*, 88:15–32, 1991.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engeneering*, 7(7):417–426, 1981.

[MCS82] J. Misra, K.M. Chandy, and T. Smith. Proving safety and liveness of communicating processes with examples. In *Proc. PoDC '82*, ACM, NEW York, 1982.

[Mos94] Ben Moszkowski. Some very compositional properties. In E.-R. Olderog, editor, *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET '94). North-Holland*, pages 307–327, 1994.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[Nau66] P. Naur. Proof of algoritms by general snapshots. *BIT*, 6:310–316, 1966.

[NDOG86] V. Nguyen, A. Demers, S. Owicki, and D. Gries. A modal and temporal proof system for networks of processes. *Distributed Computing*, 1(1):7–25, 1986.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software*, 21(2):107–125, 1995.

[Oss83] M. Ossefort. Correctness proofs of communicating processes: Three illustrative examples from the literature. *ACM Transactions on Programming Languages and Systems*, 5(4):620–640, 1983.

[PJ91] P. Pandya and M. Joseph. P-A logic – a compositional proof system for distributed programs. *Distributed Computing*, 4(4), 1991.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Programming Semantics*, pages 46–57, 1977.

[Pnu84] Amir Pnueli. In transition from global to modular reasoning about programs. *Logic and Models of Concurrent Systems*, pages 123–144, 1984. K.R. Apt (ed.), NATO ASI Series, Springer-Verlag.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report Technical Report 87–874, Dept. of Computer Science, Cornell University, Ithaca, NY, 1987.

[Sch97] F.B. Schneider. *On Concurrent Programming*. Springer-Verlag, 1997.

[Sha98] N. Shankar. Machine-assisted verification using theorem proving and model checking. *Mathematical Methods in Program Development*, 1998. Manfred Broy (ed.), Springer-Verlag.

[Sou84] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *Toplas*, 6:647–662, 1984.

[Sta85] E. Stark. A proof technique for rely/guarantee properties. In *Proceedings of 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, pages 369–391. Springer-Verlag, 1985.

[Sti88] Colin Stirling. A generalization of Owicki & Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.

[SY96] Joseph Sifakis and Serge Yovine. Compositional specification of timed systems. In *Proc. of STACS '96*, LNCS 1046. Springer-Verlag, 1996.

[Tra] B.A. Trakhtenbrot. On the power of compositional proofs for nets: relationships between completeness and modularity. Dedicated to the memory of Helena Rasiowa. Undated draft.

[Tur49] A. Turing. On checking a large routine. Report of a conference on high-speed automatic calculating machines, University Mathematical Laboratory, Cambridge, 1949.

[XCC94] Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In Jonsson and Parrow, editors, *Proc. of CONCUR '94*, LNCS 836. Springer-Verlag, 1994.

[XdRH97] Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared-variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[XH91] Q. Xu and J. He. A theory of state-based parallel programming: Part 1. In Morris, editor, *Proceedings of BCS FACS 4th Refinement Workshop*. Springer-Verlag, January 1991.

[ZdBdR84] J. Zwiers, A. de Bruin, and W.-P. de Roever. A proof system for partial correctness of dynamic networks of processes. In *Proceedings of the Conference on Logics of Programs 1983*, LNCS 164, 1984.

[ZdRvEB85] J. Zwiers, W.-P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: soundness and completeness of a proof system. In *Proceedings of 12th ICALP*, LNCS 194, pages 509–519, Nafplion, Greece, jul 15–19 1985. Springer-Verlag.

[ZHLdR95] J. Zwiers, U. Hannemann, Y. Lakhnech, and W.-P. de Roever. Synthesizing different development paradigms: Combining top-down with bottom-up reasoning about distributed systems. In *Proceedings of FST & TCS Bangalore*, LNCS 1026. Springer-Verlag, 1995.

[Zwi89] J. Zwiers. *Compositionality and Partial Correctness.* LNCS 321. Springer-Verlag, 1989.

Alternating-time Temporal Logic* **

Rajeev Alur[1] and Thomas A. Henzinger[2] and Orna Kupferman[2]

[1] Department of Computer and Information Science,
University of Pennsylvania, Philadelphia, PA 19104,
and Computing Science Research Center, Bell Laboratories, Murray Hill, NJ 07974.
Email: alur@cis.upenn.edu. URL: www.cis.upenn.edu/~alur.
[2] Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, CA 94720.
Email: {tah,orna}@eecs.berkeley.edu. URL: www.eecs.berkeley.edu/~{tah,orna}.

Abstract Temporal logic comes in two varieties: linear-time temporal logic assumes implicit universal quantification over all paths that are generated by system moves; branching-time temporal logic allows explicit existential and universal quantification over all paths. We introduce a third, more general variety of temporal logic: *alternating-time temporal logic* offers selective quantification over those paths that are possible outcomes of games, such as the game in which the system and the environment alternate moves. While linear-time and branching-time logics are natural specification languages for closed systems, alternating-time logics are natural specification languages for open systems. For example, by preceding the temporal operator "eventually" with a selective path quantifier, we can specify that in the game between the system and the environment, the system has a strategy to reach a certain state. Also the problems of receptiveness, realizability, and controllability can be formulated as model-checking problems for alternating-time formulas.

Depending on whether we admit arbitrary nesting of selective path quantifiers and temporal operators, we obtain the two alternating-time temporal logics ATL and ATL*. We interpret the formulas of ATL and ATL* over *alternating transition systems*. While in ordinary transition systems, each transition corresponds to a possible step of the system, in alternating transition systems, each transition corresponds to a possible move in the game between the system and the environment. Fair alternating transition systems can capture both synchronous and asynchronous compositions of open systems. For synchronous systems, the expressive power of ATL beyond CTL comes at no cost: the model-checking complexity of synchronous ATL is linear in the size of the system and the length of the formula. The symbolic model-checking algorithm for CTL extends with few modifications to synchronous ATL, and with some work, also to asynchronous ATL, whose model-checking complexity is quadratic. This makes ATL an obvious candidate for the automatic verification of open systems. In the case of ATL*, the model-checking problem is closely related to the synthesis problem for linear-time formulas, and requires doubly exponential time for both synchronous and asynchronous systems.

* A preliminary version of this paper appeared in the *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science* (FOCS 1997), pp. 100–109.

** This work was supported in part by the ONR YIP award N00014-95-1-0520, by the NSF CAREER award CCR-9501708, by the NSF grant CCR-9504469, by the AFOSR contract F49620-93-1-0056, by the ARO MURI grant DAAH-04-96-1-0341, by the ARPA grant NAG2-892, and by the SRC contract 97-DC-324.041.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 23-60, 1998.
Springer-Verlag Berlin Heidelberg 1998

1 Introduction

In 1977, Pnueli proposed to use *linear-time temporal logic* (LTL) to specify requirements for reactive systems [Pnu77]. A formula of LTL is interpreted over a computation, which is an infinite sequence of states. A reactive system satisfies an LTL formula if all its computations do. Due to the implicit use of universal quantification over the set of computations, LTL cannot express existential, or possibility, properties. *Branching-time temporal logics* such as CTL and CTL*, on the other hand, do provide explicit quantification over the set of computations [CE81, EH86]. For instance, for a state predicate φ, the CTL formula $\forall\Diamond\,\varphi$ requires that a state satisfying φ is visited in all computations, and the CTL formula $\exists\Diamond\,\varphi$ requires that there exists a computation that visits a state satisfying φ. The problem of *model checking* is to verify whether a finite-state abstraction of a reactive system satisfies a temporal-logic specification [CE81, QS81]. Efficient model checkers exist for both LTL (e.g., SPIN [Hol97]) and CTL (e.g., SMV [McM93]), and are increasingly being used as debugging aids for industrial designs.

The logics LTL and CTL have their natural interpretation over the computations of closed systems, where a *closed system* is a system whose behavior is completely determined by the state of the system. However, the compositional modeling and design of reactive systems requires each component to be viewed as an open system, where an *open system* is a system that interacts with its environment and whose behavior depends on the state of the system as well as the behavior of the environment. Models for open systems, such as CSP [Hoa85], I/O automata [Lyn96], and Reactive Modules [AH96], distinguish between *internal* nondeterminism, choices made by the system, and *external* nondeterminism, choices made by the environment. Consequently, besides universal (do all computations satisfy a property?) and existential (does some computation satisfy a property?) questions, a third question arises naturally: can the system resolve its internal choices so that the satisfaction of a property is guaranteed no matter how the environment resolves the external choices? Such an *alternating* satisfaction can be viewed as a winning condition in a two-player game between the system and the environment. Alternation is a natural generalization of existential and universal branching, and has been studied extensively in theoretical computer science [CKS81].

Different researchers have argued for game-like interpretations of LTL and CTL specifications for open systems. We list four such instances here.

Receptiveness [Dil89, AL93, GSSL94]: Given a reactive system, specified by a set of *safe* computations (typically, generated by a transition relation) and a set of *live* computations (typically, expressed by an LTL formula), the receptiveness problem is to determine whether every finite safe computation can be extended to an infinite live computation irrespective of the behavior of the environment. It is sensible, and necessary for compositionality, to require an affirmative answer to the receptiveness problem.

Realizability (program synthesis) [ALW89, PR89a, PR89b]: Given an LTL formula ψ over sets of input and output signals, the synthesis problem requires the construction of a reactive system that assigns to every possible input sequence an output sequence so that the resulting computation satisfies ψ.

Supervisory control [RW89]: Given a finite-state machine whose transitions are partitioned into controllable and uncontrollable, and a set of safe states, the control problem requires the construction of a controller that chooses the controllable transitions so that the machine always stays within the safe set (or satisfies some more general LTL formula).

Module checking [KV96]: Given an open system and a CTL* formula φ, the module-checking problem is to determine if, no matter how the environment restricts the external choices, the system satisfies φ.

All the above approaches use the temporal-logic syntax that was developed for specifying closed systems, and reformulate its semantics for open systems. In this paper, we propose, instead, to enrich temporal logic so that alternating properties can be specified explicitly within the logic: we introduce *alternating-time temporal logics* for the specification and verification of open systems. Our formulation of open systems considers, instead of just a system and an environment, the more general setting of a set Σ of agents that correspond to different components of the system and the environment. We model open systems by *alternating transition systems*. The transitions of an alternating transition system correspond to possible moves in a game between the agents. In each move of the game, every agent chooses a set of successor states. The game then proceeds to the (single) state in the intersection of the sets chosen by the agents. Special cases of the game are *turn-based synchronous* (in each state, only one agent restricts the set of successor states, and that agent is determined by the state), *lock-step synchronous* (the state is partitioned according to the agents, and in each step, every agent updates its component of the state), and *turn-based asynchronous* (in each state, only one agent restricts the set of successor states, and that agent is chosen by a fair scheduler). These subclasses of alternating transition systems capture various notions of synchronous and asynchronous interaction between open systems.

For a set $A \subseteq \Sigma$ of agents, a set Λ of computations, and a state q of the system, consider the following game between a protagonist and an antagonist. The game starts at the state q. At each step, to determine the next state, the protagonist chooses the choices controlled by the agents in the set A, while the antagonist chooses the remaining choices. If the resulting infinite computation belongs to the set Λ, then the protagonist wins. If the protagonist has a winning strategy, we say that the alternating-time formula $\langle\langle A \rangle\rangle \Lambda$ is satisfied in the state q. Here, $\langle\langle A \rangle\rangle$ is a *path quantifier*, parameterized with the set A of agents, which ranges over all computations that the agents in A can force the game into, irrespective of how the agents in $\Sigma \setminus A$ proceed. Hence, the parameterized path quantifier $\langle\langle A \rangle\rangle$ is a generalization of the path quantifiers of branching-time

temporal logics: the existential path quantifier $\exists$ corresponds to $\langle\langle\Sigma\rangle\rangle$, and the universal path quantifier $\forall$ corresponds to $\langle\langle\emptyset\rangle\rangle$. In particular, closed systems can be viewed as systems with a single agent *sys*, which represents the system. Then, the two possible parameterized path quantifiers $\langle\langle sys\rangle\rangle$ and $\langle\langle\emptyset\rangle\rangle$ match exactly the path quantifiers $\exists$ and $\forall$ required for specifying such systems. Depending on the syntax used to specify the set Λ of computations, we obtain two alternating-time temporal logics: in the logic ATL*, the set Λ is specified by a formula of LTL; in the more restricted logic ATL, the set Λ is specified by a single temporal operator applied to a state formula. Thus, ATL is the alternating generalization of CTL, and ATL* is the alternating generalization of CTL*.

Alternating-time temporal logics can conveniently express properties of open systems as illustrated by the following five examples:

1. In a multi-process distributed system, we can require any subset of processes to attain a goal, irrespective of the behavior of the remaining processes. Consider, for example, the cache-coherence protocol for Gigamax verified using SMV [McM93]. One of the desired properties is the absence of deadlocks, where a deadlocked state is one in which a processor, say a, is permanently blocked from accessing a memory cell. This requirement was specified using the CTL formula

   $$\forall\Box\,(\exists\Diamond\, \mathit{read} \;\wedge\; \exists\Diamond\, \mathit{write}).$$

   The ATL formula

   $$\forall\Box\,(\langle\langle a\rangle\rangle\Diamond\, \mathit{read} \;\wedge\; \langle\langle a\rangle\rangle\Diamond\, \mathit{write})$$

   captures the informal requirement more precisely. While the CTL formula only asserts that it is always possible for all processors to *cooperate* so that a can eventually read and write ("collaborative possibility"), the ATL formula is stronger: it guarantees a memory access for processor a, *no matter what the other processors in the system do* ("adversarial possibility").
2. While the CTL formula $\forall\Box\,\varphi$ asserts that the state predicate φ is an invariant of a system component irrespective of the behavior of all other components ("adversarial invariance"), the ATL formula $[\![a]\!]\Box\varphi$ (which stands for $\langle\langle\Sigma\setminus\{a\}\rangle\rangle\,\Box\varphi$) states the weaker requirement that φ is a *possible invariant* of the component a; that is, a cannot violate $\Box\varphi$ on its own, and therefore the other system components may cooperate to achieve $\Box\varphi$ ("collaborative invariance"). For φ to be an invariant of a complex system, it is necessary (but not sufficient) to check that every component a satisfies the ATL formula $[\![a]\!]\Box\varphi$.
3. The *receptiveness* of a system whose live computations are given by the LTL formula ψ is specified by the ATL* formula $\forall\Box\,\langle\langle sys\rangle\rangle\,\psi$.
4. Checking the *realizability (program synthesis)* of an LTL formula ψ corresponds to model checking of the ATL* formula $\langle\langle sys\rangle\rangle\psi$ in a maximal model that considers all possible inputs and outputs.

5. The *controllability* of a system whose safe states are given by the state predicate φ is specified by the ATL formula $\langle\langle control \rangle\rangle \Box \varphi$. Controller synthesis, then, corresponds to model checking of this formula. More generally, for an LTL formula ψ, the ATL* requirement $\langle\langle control \rangle\rangle \psi$ asserts that the controller has a strategy to ensure the satisfaction of ψ.

Notice that ATL is better suited for compositional reasoning than CTL. For instance, if a component a satisfies the CTL formula $\exists\Diamond \varphi$, we cannot conclude that the compound system $a \| b$ also satisfies $\exists\Diamond \varphi$. On the other hand, if a satisfies the ATL formula $\langle\langle a \rangle\rangle \Diamond\varphi$, then so does $a \| b$.

The model-checking problem for alternating-time temporal logics requires the computation of winning strategies. In the case of synchronous ATL, all games are *finite* reachability games. Consequently, the model-checking complexity is linear in the size of the system and the length of the formula, just as in the case of CTL. While checking existential reachability corresponds to iterating the existential next-time operator $\exists\bigcirc$, and checking universal reachability corresponds to iterating the universal next $\forall\bigcirc$, checking alternating reachability corresponds to iterating an appropriate mix of $\exists\bigcirc$ and $\forall\bigcirc$, as governed by a parameterized path quantifier. This suggests a simple symbolic model-checking procedure for synchronous ATL, and shows how existing symbolic model checkers for CTL can be modified to check ATL specifications, at no extra cost. In the asynchronous model, due to the presence of fairness constraints, ATL model checking requires the solution of *infinite* games, namely, generalized Büchi games [VW86b]. Consequently, the model-checking complexity is quadratic in the size of the system, and the symbolic algorithm involves a nested fixed-point computation. The model-checking problem for ATL* is much harder: we show it to be complete for 2EXPTIME in both the synchronous and asynchronous cases.

The remaining paper is organized as follows. Section 2 defines the model of alternating transition systems, and Section 3 defines the alternating-time temporal logics ATL and ATL*. Section 4 presents symbolic model-checking procedures, and Section 5 establishes complexity bounds on model checking for alternating-time temporal logics. In Section 6, we consider more general ways of introducing game quantifiers in temporal logics. Specifically, we define an alternating-time μ-calculus and a game logic, and study their relationship to ATL and ATL*. Finally, Section 7 considers models in which agents have only partial information about (global) states. We show that for this case, of alternating transition systems with incomplete information, the model-checking problem is generally undecidable, and we describe a special case that is decidable in exponential time.

2 Alternating Transition Systems

We model open systems by alternating transition systems. While in ordinary transition systems, each transition corresponds to a possible step of the system, in alternating transition systems, each transition corresponds to a possible step in a game between the agents that constitute the system.

2.1 Definition of ATS

An *alternating transition system* (ATS, for short) is a 5-tuple $S = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$ with the following components:

- Π is a set of propositions.
- Σ is a set of agents.
- Q is a set of states.
- $\pi : Q \to 2^{\Pi}$ maps each state to the set of propositions that are true in the state.
- $\delta : Q \times \Sigma \to 2^{2^Q}$ is a transition function that maps a state and an agent to a nonempty set of choices, where each choice is a set of possible next states. Whenever the system is in state q, each agent a chooses a set $Q_a \in \delta(q, a)$. In this way, an agent a ensures that the next state of the system will be in its choice Q_a. However, which state in Q_a will be next depends on the choices made by the other agents, because the successor of q must lie in the intersection $\bigcap_{a \in \Sigma} Q_a$ of the choices made by all agents. The transition function is non-blocking and the agents together choose a unique next state state. Thus, we require that this intersection always contains a unique state: assuming $\Sigma = \{a_1, \ldots, a_n\}$, then for every state $q \in Q$ and every set $Q_1, \ldots, Q_n$ of choices $Q_i \in \delta(q, a_i)$, the intersection $Q_1 \cap \ldots \cap Q_n$ is a singleton.

The number of transitions of S is defined to be $\sum_{q \in Q, a \in \Sigma} |\delta(q, a)|$. For two states q and q' and an agent a, we say that q' is an *a-successor* of q if there exists a set $Q' \in \delta(q, a)$ such that $q' \in Q'$. We denote by $succ(q, a)$ the set of a-successors of q. For two states q and q', we say that q' is a *successor* of q if for all agents $a \in \Sigma$, we have $q' \in succ(q, a)$. Thus, q' is a successor of q iff whenever the system S is in state q, the agents in Σ can cooperate so that q' will be the next state. A *computation* of S is an infinite sequence $\lambda = q_0, q_1, q_2, \ldots$ of states such that for all positions $i \geq 0$, the state q_{i+1} is a successor of the state q_i. We refer to a computation starting at state q as a *q-computation*. For a computation λ and a position $i \geq 0$, we use $\lambda[i]$, $\lambda[0, i]$, and $\lambda[i, \infty]$ to denote the i-th state in λ, the finite prefix $q_0, q_1, \ldots, q_i$ of λ, and the infinite suffix $q_i, q_{i+1}, \ldots$ of λ, respectively.

Example 1. Consider a system with two processes a and b. The process a assigns values to the boolean variable x. When $x = \mathit{false}$, then a can leave the value of x unchanged or change it to true. When $x = \mathit{true}$, then a leaves the value of x unchanged. In a similar way, the process b assigns values to the boolean variable y. When $y = \mathit{false}$, then b can leave the value of y unchanged or change it to true. When $y = \mathit{true}$, then b leaves the value of y unchanged. We model the composition of the two processes by the following ATS $S_{xy} = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$:

- $\Pi = \{x, y\}$.
- $\Sigma = \{a, b\}$.
- $Q = \{q, q_y, q_x, q_{xy}\}$. The state q corresponds to $x = y = \mathit{false}$, the state q_x corresponds to $x = \mathit{true}$ and $y = \mathit{false}$, and similarly for q_y and q_{xy}.

- The labeling function $\pi : Q \to 2^{\Pi}$ is therefore as follows:
  - $\pi(q) = \emptyset$.
  - $\pi(q_x) = \{x\}$.
  - $\pi(q_y) = \{y\}$.
  - $\pi(q_{xy}) = \{x, y\}$.
- The transition function $\delta : Q \times \Sigma \to 2^{2^Q}$ is as follows:
  - $\delta(q, a) = \{\{q, q_y\}, \{q_x, q_{xy}\}\}$.
  - $\delta(q, b) = \{\{q, q_x\}, \{q_y, q_{xy}\}\}$.
  - $\delta(q_x, a) = \{\{q_x, q_{xy}\}\}$.
  - $\delta(q_x, b) = \{\{q, q_x\}, \{q_y, q_{xy}\}\}$.
  - $\delta(q_y, a) = \{\{q, q_y\}, \{q_x, q_{xy}\}\}$.
  - $\delta(q_y, b) = \{\{q_y, q_{xy}\}\}$.
  - $\delta(q_{xy}, a) = \{\{q_x, q_{xy}\}\}$.
  - $\delta(q_{xy}, b) = \{\{q_y, q_{xy}\}\}$.

  Consider, for example, the transition $\delta(q, a)$. As the process a controls only the value of x, and can change its value from *false* to *true*, the agent a can determine whether the next state of the system will be some q' with $x \in \pi(q')$ or some q' with $x \notin \pi(q')$. It cannot, however, determine the value of y. Therefore, $\delta(q, a) = \{\{q, q_y\}, \{q_x, q_{xy}\}\}$, letting a choose between $\{q, q_y\}$ and $\{q_x, q_{xy}\}$, yet leaving the choice between q and q_y, in the first case, and between q_x and q_{xy}, in the second case, to process b.

Consider the state q_x. While the state q_y is a b-successor of q_x, the state q_y is not an a-successor of q_x. Therefore, the state q_y is not a successor of q_x: when the system is in state q_x, the processes a and b cannot cooperate so that the system will move to q_y. On the other hand, the agents can cooperate so that the system will stay in state q_x or move to q_{xy}. By similar considerations, it follows that the infinite sequences $q, q, q_x, q_x, q_x, q_{xy}^{\omega}$ and $q, q_y, q_y, q_{xy}^{\omega}$ and q, q_{xy}^{ω} are three possible q-computations of the ATS S_{xy}.

Now suppose that process b can change y from *false* to *true* only when x is already *true*. The resulting ATS $S'_{xy} = \langle \Pi, \Sigma, Q, \pi, \delta' \rangle$ differs from S_{xy} only in the transition function: $\delta'(q, b) = \{\{q, q_x\}\}$, and in all other cases δ' agrees with δ. While $q, q, q_x, q_x, q_x, q_{xy}^{\omega}$ is a possible q-computation of the ATS S'_{xy}, the sequences $q, q_y, q_y, q_{xy}^{\omega}$ and q, q_{xy}^{ω} are not.

Third, suppose that process b can change y from *false* to *true* either when x is already *true*, or when simultaneously x is set to *true*. The transition function of the resulting ATS $S''_{xy} = \langle \Pi, \Sigma, Q, \pi, \delta'' \rangle$ differs from δ only in $\delta''(q, b) = \{\{q, q_x\}, \{q, q_{xy}\}\}$. In state q, if process b decides to leave y unchanged, it chooses the first option $\{q, q_x\}$. If, on the other hand, process b decides to change the value of y to *true* provided that x is simultaneously changed to *true* by process a, then b chooses the second option $\{q, q_{xy}\}\}$. Then $q, q, q_x, q_x, q_x, q_{xy}^{\omega}$ and q, q_{xy}^{ω} are possible q-computations of the ATS S''_{xy}, while $q, q_y, q_y, q_{xy}^{\omega}$ is not.

Finally, suppose we consider process b on its own. In this case, we have two agents, b and *env*, where *env* represents the environment, which may, in any

state, change the value of x arbitrarily. The resulting ATS $S'''_{xy} = \langle \Pi, \Sigma''', Q, \pi, \delta''' \rangle$ has the set $\Sigma''' = \{b, env\}$ of agents and the following transition relation:

- $\delta'''(q, b) = \delta'''(q_x, b) = \{\{q, q_x\}, \{q_y, q_{xy}\}\}$.
- $\delta'''(q_y, b) = \delta'''(q_{xy}, b) = \{\{q_y, q_{xy}\}\}$.
- $\delta'''(q, env) = \delta'''(q_x, env) = \delta'''(q_y, env) = \delta'''(q_{xy}, env) = \{\{q, q_y\}, \{q, q_{xy}\}, \{q_x, q_{xy}\}, \{q_x, q_y\}\}$.

□

An ordinary *labeled transition system*, or Kripke structure, is the special case of an ATS where the set $\Sigma = \{sys\}$ of agents is a singleton set. In this special case, the sole agent sys can always determine the successor state: for all states $q \in Q$, the transition $\delta(q, sys)$ must contain a nonempty set of choices, each of which is a singleton set.

2.2 Synchronous ATS

In this section we present two special cases of alternating transition systems. Both cases correspond to a synchronous composition of agents.

Turn-based synchronous ATS In a turn-based synchronous ATS, at every state only a single agent is scheduled to proceed and that agent determines the next state. It depends on the state which agent is scheduled. Accordingly, an ATS is *turn-based synchronous* if for every state $q \in Q$, there exists an agent $a_q \in \Sigma$ such that $\delta(q, a_q)$ is a set of singleton sets and for all agents $b \in \Sigma \backslash \{a_q\}$, we have $\delta(q, b) = \{Q\}$. Thus, in every state q only the agent a_q constrains the choice of the successor state. Equivalently, a turn-based synchronous ATS can be viewed as a 6-tuple $S = \langle \Pi, \Sigma, Q, \pi, \sigma, R \rangle$, where $\sigma : Q \to \Sigma$ maps each state q to the agent a_q that is scheduled to proceed at q, and $R \subseteq Q \times Q$ is a total transition relation. Then q' is a successor of q iff $R(q, q')$.

Example 2. Consider the ATS $S_1 = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$ shown in Figure 1:

- $\Pi = \{out_of_gate, in_gate, request, grant\}$.
- $\Sigma = \{train, ctr\}$.
- $Q = \{q_0, q_1, q_2, q_3\}$.
-
  - $\pi(q_0) = \{out_of_gate\}$.
  - $\pi(q_1) = \{out_of_gate, request\}$.
  - $\pi(q_2) = \{out_of_gate, grant\}$.
  - $\pi(q_3) = \{in_gate\}$.
-
  - $\delta(q_0, train) = \{\{q_0\}, \{q_1\}\}$.
  - $\delta(q_1, ctr) = \{\{q_0\}, \{q_1\}, \{q_2\}\}$.
  - $\delta(q_2, train) = \{\{q_0\}, \{q_3\}\}$.
  - $\delta(q_3, ctr) = \{\{q_0\}, \{q_3\}\}$.
  - $\delta(q_0, ctr) = \delta(q_1, train) = \delta(q_2, ctr) = \delta(q_3, train) = \{Q\}$.

Since S_1 is a turn-based synchronous ATS, its transition function δ induces an assignment of agents to states: $\sigma(q_0) = \sigma(q_2) = \mathit{train}$ and $\sigma(q_1) = \sigma(q_3) = \mathit{ctr}$. The ATS describes a protocol for a train entering a gate at a railroad crossing. At each moment, the train is either *out_of_gate* or *in_gate*. In order to enter the gate, the train issues a request, which is serviced (granted or rejected) by the controller in the next step. After a grant, the train may enter the gate or relinquish the grant. The system has two agents: the train and the controller. Two states of the system, labeled *ctr*, are controlled; that is, when a computation is in one of these states, the controller chooses the next state. The other two states are not controlled, and the train chooses successor states. □

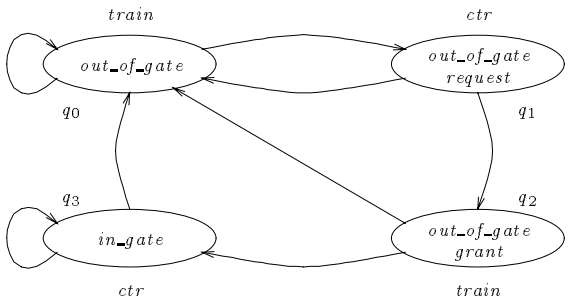


Fig. 1. A train controller as a turn-based synchronous ATS

Lock-step synchronous ATS In a lock-step synchronous ATS, the state space is the product of local state spaces, one for each agent. Then, in every state, all agents proceed simultaneously. Each agent determines its next local state, possibly dependent on the current local states of the other agents but independent of the choices taken by the other agents. Accordingly, an ATS is *lock-step synchronous* if the following two conditions are satisfied:

1. The state space has the form $Q = \Pi_{a \in \Sigma} Q_a$. Given a (global) state $q \in Q$ and an agent $a \in \Sigma$, we write $q[a]$ for the component of q local to a. Then, assuming $\Sigma = \{a_1, \ldots, a_n\}$, every state has the form $q = \langle q[a_1], \ldots, q[a_n] \rangle$.
2. For every state $q \in Q$ and every agent $a \in \sigma$, there exists a set $\{q_1, \ldots, q_k\} \subseteq Q_a$ of states local to a such that $\delta(q, a) = \{Q_1, \ldots, Q_k\}$ for $Q_i = \{q \in Q \mid q[a] = q_i\}$. Thus, while the agent a can determine its next local state, it cannot determine the next local states of the other agents.

Equivalently, the transition function δ can be replaced by a set of local transition functions $\delta_a : Q \to 2^{Q_a}$, one for each agent $a \in \Sigma$ and all of them total. Then q' is a successor of q iff for all agents $a \in \Sigma$, we have $q'[a] \in \delta_a(q)$.

Example 3. The ATS S_{xy} from Example 1 is lock-step synchronous. To see this, note that its state space $Q = \{q, q_x, q_y, q_{xy}\}$ can be viewed as the product of $Q_a = \{u, u_x\}$ and $Q_b = \{v, v_y\}$ with $q = \langle u, v\rangle$, $q_x = \langle u_x, v\rangle$, $q_y = \langle u, v_y\rangle$, and $q_{xy} = \langle u_x, v_y\rangle$. The local transition functions are as follows:

- $\delta_a(q) = \delta_a(q_y) = \{u, u_x\}$.
- $\delta_a(q_x) = \delta_a(q_{xy}) = \{u_x\}$.
- $\delta_b(q) = \delta_b(q_x) = \{v, v_y\}$.
- $\delta_b(q_y) = \delta_b(q_{xy}) = \{v_y\}$.

Also the ATS S'_{xy} from Example 1 is lock-step synchronous, but the ATS S''_{xy} and S'''_{xy} are not. For S''_{xy}, this is because the ability of process b to change the value of y depends on what process a does at the same step. □

2.3 Fair ATS

When systems are modeled as ordinary transition systems, to establish liveness properties, it is often necessary to rule out certain infinite computations that ignore enabled choices forever. For instance, in an asynchronous system consisting of many processes, we may like to restrict attention to the computations in which all the processes take infinitely many steps. Such assumptions can be incorporated in the model by adding fairness conditions. Motivated by similar concerns, we define fairness conditions for ATS.

A *fairness condition* for the ATS $S = \langle \Pi, \Sigma, Q, \pi, \delta\rangle$ is a set of fairness constraints for S, each defining a subset of the transition function. More precisely, a *fairness constraint* for S is a function $\gamma : Q \times \Sigma \to 2^{2^Q}$ such that $\gamma(q, a) \subseteq \delta(q, a)$ for all states $q \in Q$ and all agents $a \in \Sigma$. As with ordinary transition systems, a fairness condition partitions the computations of an ATS into computations that are fair and computations that are not fair. We elaborate on two interpretations for fairness constraints. Consider a computation $\lambda = q_0, q_1, q_2, \ldots$ of the ATS S, a fairness constraint $\gamma : Q \times \Sigma \to 2^{2^Q}$ for S, and an agent $a \in \Sigma$. We say that γ is *a-enabled* at position $i \geq 0$ of λ if $\gamma(q_i, a) \neq \emptyset$. We say that γ is *a-taken* at position i of λ if there exists a set $Q' \in \gamma(q_i, a)$ such that $q_{i+1} \in Q'$. The two interpretations for fairness constraints are defined with respect to a set $A \subseteq \Sigma$ of agents as follows:

- The computation λ is *weakly $\langle \gamma, A\rangle$-fair* if for each agent $a \in A$, either there are infinitely many positions of λ at which γ is not a-enabled, or there are infinitely many positions of λ at which γ is a-taken.
- The computation λ is *strongly $\langle \gamma, A\rangle$-fair* if for each agent $a \in A$, either there are only finitely many positions of λ at which γ is a-enabled, or there are infinitely many positions of λ at which γ is a-taken. With these standard definitions, strong fairness implies weak fairness.

Now, given a fairness condition Γ for the ATS S and a set $A \subseteq \Sigma$ of agents, the computation λ is *weakly/strongly $\langle \Gamma, A\rangle$-fair* if λ is weakly/strongly $\langle \gamma, A\rangle$-fair

for all fairness constraints $\gamma \in \Gamma$. Note that for every fairness condition Γ and every set A of agents, each prefix of a computation of S can be extended to a computation that is strongly $\langle \Gamma, A \rangle$-fair. Note also that a computation λ is weakly/strongly $\langle \Gamma, A_1 \cup A_2 \rangle$-fair, for $A_1, A_2 \subseteq \Sigma$, iff λ is weakly/strongly both $\langle \Gamma, A_1 \rangle$-fair and $\langle \Gamma, A_2 \rangle$-fair,

Example 4. Consider the ATS S_{xy} from Example 1 and the fairness condition $\Gamma_y = \{\gamma\}$ with the fairness constraint $\gamma(q, b) = \gamma(q_x, b) = \{\{q_y, q_{xy}\}\}$ (we specify only the nonempty values of a fairness constraint). All computations of the ATS S_{xy} are strongly $\langle \Gamma_y, \{a\} \rangle$-fair. However, only computations in which the value of the variable y is eventually *true* are weakly or strongly $\langle \Gamma_y, \{b\} \rangle$-fair or, for that matter, $\langle \Gamma_y, \{a, b\} \rangle$-fair. This is because, as long as the value of y is *false*, the ATS S_{xy} is either in state q or in state q_x. Therefore, as long as the value of y is *false*, the fairness constraint γ is b-enabled. Thus, in a fair computation, γ will eventually be b-taken, changing the value of y to *true*. □

As with ordinary transition systems, fairness enables us to exclude some computations of an ATS. In particular, fairness enables us to model asynchronous systems.

Turn-based asynchronous ATS In a turn-based asynchronous ATS, at every state only a single agent determines the next state. However, unlike in a turn-based synchronous ATS, the state does not determine which agent is scheduled to proceed. Rather, a turn-based asynchronous ATS has a designated agent *sch*, which represents a *scheduler*. The scheduler *sch* proceeds at all states and determines one other agent to proceed with it. That other agent determines the next state. Fairness constraints are used to guarantee that the scheduling policy is fair. Accordingly, an ATS is *turn-based asynchronous* if there exists an agent $sch \in \Sigma$ and for every state $q \in Q$ and every agent $a \in \Sigma \setminus \{sch\}$, there exists a local transition function $\delta_a : Q \to 2^Q$ such that the following four conditions are satisfied:

1. For all states $q \in Q$ and all agents $a, b \in \Sigma \setminus \{sch\}$, if $a \neq b$ then $\delta_a(q) \cap \delta_b(q) = \emptyset$. We say that agent a is *enabled* in state q if $\delta_a(q) \neq \emptyset$.
2. For all states $q \in Q$, we have $\delta(q, sch) = \{\delta_a(q) \mid \text{the agent } a \in \Sigma \setminus \{sch\} \text{ is enabled in } q\}$. That is, if the scheduler *sch* chooses the option $\delta_a(q)$, the agent a is scheduled to proceed in state q.
3. For all states $q \in Q$ and all agents $a \in \Sigma \setminus \{sch\}$ that are not enabled in q, we have $\delta(q, a) = \{Q\}$. That is, if the agent a is not enabled, it does not influence the successor state.
4. For all states $q \in Q$ and all agents $a \in \Sigma \setminus \{sch\}$ that are enabled in q, assuming $\delta_a(q) = \{q_1, \ldots, q_k\}$, we have $\delta(q, a) = \{(Q \setminus \delta_a(q)) \cup \{q_1\}, \ldots, (Q \setminus \delta_a(q)) \cup \{q_k\}\}$. That, if the agent a is enabled in state q, it chooses a successor state in $\delta_a(q)$ provided it is scheduled to proceed. If, however, a is not scheduled to proceed in q, then it does not influence the successor state, which must lie in $Q \setminus \delta_a(q)$ because of the first condition.

Equivalently, a turn-based asynchronous ATS can be viewed as a 6-tuple $S = \langle \Pi, \Sigma \setminus \{sch\}, Q, \pi, R, \sigma \rangle$, where $R \subseteq Q \times Q$ is a total transition relation, and $\sigma : R \to \Sigma \setminus \{s\}$ maps each transition to an agent. Then $\delta_a(q) = \{q' \in Q \mid R(q, q')$ and $\sigma(q, q') = a\}$. Note that, while in a turn-based synchronous ATS we label states with agents, in a turn-based asynchronous ATS we label transitions with agents.

In order to ensure fairness of the scheduler, we impose a fairness condition $\Gamma = \{\gamma_a \mid a \in \Sigma \setminus \{sch\}\}$ with a turn-based asynchronous ATS. The fairness condition Γ contains a fairness constraint γ_a for each agent a different from sch, which ensures that the scheduler does not neglect a forever. For all states $q \in Q$, we have $\gamma_a(q, sch) = \{\delta_a(q)\}$ and for all agents $b \in \Sigma \setminus \{sch\}$ (possibly $b = a$), we have $\gamma_a(q, b) = \emptyset$. Then, a computation λ is weakly $\langle \gamma_a, \{sch\} \rangle$-fair iff either there are infinitely many positions of λ at which the agent a is not enabled, or there are infinitely many positions of λ at which a is scheduled to proceed. Similarly, λ is strongly $\langle \gamma_a, \{sch\} \rangle$-fair iff either there are only finitely many positions of λ at which the agent a is enabled, or there are infinitely many positions of λ at which a is scheduled to proceed.

Example 5. As an example of a turn-based asynchronous ATS consider the modeling of the sender process of the alternating-bit protocol shown in Figure 2. There are two agents, the sender and the environment. In the initial state q_0, only the sender is enabled, and it chooses either to stay in q_0 or to move to state q_1. The transition from q_0 to q_1 corresponds to sending a message tagged with the bit 0. In state q_1 the sender is waiting to receive an acknowledgment. Both agents are enabled, and the scheduler chooses one of them. The sender, if scheduled to proceed in state q_1, continues to wait. Each environment transitions correspond to the reception of an acknowledgment by the sender. If the acknowledgment bit is 0, the sender proceeds to toggle its bit by moving to state q_2, and if the acknowledgment bit is 1, the sender attempts to resend the message by moving back to state q_0. This phenomenon is modeled by letting the environment, when scheduled in state q_1, choose between q_0 and q_2. State q_1 is similar to state q_0, and q_3 is similar to q_1.

Formally, $Q = \{q_0, q_1, q_2, q_3\}$ and $\Sigma = \{sender, env, sch\}$. The set Π contains four propositions: *send0* is true in state q_0, *wait0* is true in state q_1, *send1* is true in state q_2, and *wait1* is true in state q_3. The local transition functions are as follows:

- $\delta_{sender}(q_0) = \{q_0, q_1\}$.
- $\delta_{env}(q_0) = \emptyset$.
- $\delta_{sender}(q_1) = \{q_1\}$.
- $\delta_{env}(q_1) = \{q_0, q_2\}$.
- $\delta_{sender}(q_2) = \{q_2, q_3\}$.
- $\delta_{env}(q_2) = \emptyset$.
- $\delta_{sender}(q_3) = \{q_3\}$.
- $\delta_{env}(q_3) = \{q_0, q_2\}$.

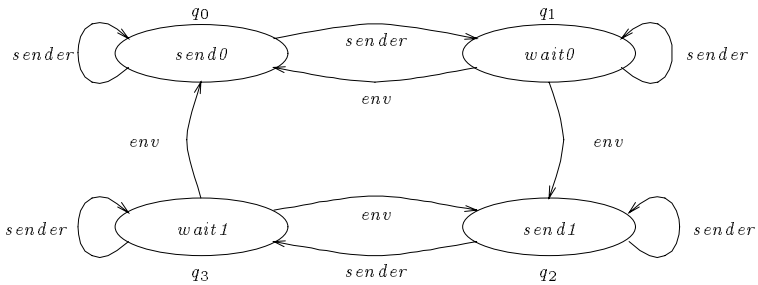


Fig. 2. A send protocol as a turn-based asynchronous ATS

These local transition functions induce the following transition function:

- $\delta(q_0, sch) = \{\{q_0, q_1\}\}$.
- $\delta(q_0, sender) = \{\{q_0, q_2, q_3\}, \{q_1, q_2, q_3\}\}$.
- $\delta(q_0, env) = \{\{q_0, q_1, q_2, q_3\}\}$.
- $\delta(q_1, sch) = \{\{q_1\}, \{q_0, q_2\}\}$.
- $\delta(q_1, sender) = \{\{q_0, q_2, q_3, q_1\}\}$.
- $\delta(q_1, env) = \{\{q_1, q_3, q_0\}, \{q_1, q_3, q_2\}\}$.
- $\delta(q_2, sch) = \{\{q_2, q_3\}\}$.
- $\delta(q_2, sender) = \{\{q_0, q_1, q_2\}, \{q_0, q_1, q_3\}\}$.
- $\delta(q_2, env) = \{\{q_0, q_1, q_2, q_3\}\}$.
- $\delta(q_3, sch) = \{\{q_3\}, \{q_0, q_2\}\}$.
- $\delta(q_3, sender) = \{\{q_0, q_1, q_2, q_3\}\}$.
- $\delta(q_3, env) = \{\{q_1, q_3, q_0\}, \{q_1, q_3, q_2\}\}$.

The weak-fairness constraint γ_{env} ensures that if the sender is waiting in state q_1 or q_2, it will eventually receive an acknowledgment:

- $\gamma_{env}(q_1, sch) = \gamma_{env}(q_3, sch) = \{\{q_0, q_2\}\}$

(we specify only the nonempty values of a fairness constraint). The assumption that the environment does not keep sending incorrect acknowledgments forever, which ensures progress of the protocol, can be modeled by a strong-fairness constraint γ':

- $\gamma'(q_1, env) = \{\{q_1, q_3, q_2\}\}$.
- $\gamma'(q_3, env) = \{\{q_1, q_3, q_0\}\}$.

□

3 Alternating-time Temporal Logic

3.1 ATL Syntax

The temporal logic ATL (*Alternating-time Temporal Logic*) is defined with respect to a finite set Π of *propositions* and a finite set Σ of *agents*. An ATL

formula is one of the following:

(S1) p, for propositions $p \in \Pi$.
(S2) $\neg\varphi$ or $\varphi_1 \vee \varphi_2$, where φ, φ_1, and φ_2 are ATL formulas.
(S3) $\langle\langle A\rangle\rangle \bigcirc\varphi$, $\langle\langle A\rangle\rangle \Box\varphi$, or $\langle\langle A\rangle\rangle \varphi_1 \mathcal{U} \varphi_2$, where $A \subseteq \Sigma$ is a set of agents, and φ, φ_1, and φ_2 are ATL formulas.

The operator $\langle\langle\ \rangle\rangle$ is a *path quantifier*, and $\bigcirc$ ("next"), $\Box$ ("always"), and $\mathcal{U}$ ("until") are *temporal operators*. The logic ATL is similar to the branching-time temporal logic CTL, only that path quantifiers are parameterized by sets of agents. Sometimes we write $\langle\langle a_1, \ldots, a_n\rangle\rangle$ instead of $\langle\langle\{a_1, \ldots, a_n\}\rangle\rangle$. Additional boolean connectives are defined from $\neg$ and $\vee$ in the usual manner. As in CTL, we write $\langle\langle A\rangle\rangle \Diamond\varphi$ for $\langle\langle A\rangle\rangle true\, \mathcal{U} \varphi$.

3.2 ATL Semantics

We interpret ATL formulas over the states of a given ATS S that has the same propositions and agents. The labeling of the states of S with propositions is used to evaluate the atomic formulas of ATL. The logical connectives $\neg$ and $\vee$ have the standard interpretation. To evaluate a formula of the form $\langle\langle A\rangle\rangle \psi$ at a state q of S, consider the following two-player game. The game proceeds in an infinite sequence of rounds, and after each round, the position of the game is a state of S. The initial position is q. Now consider the game in some position u. To update the position, first the protagonist chooses for every agent $a \in A$, a set $Q_a \in \delta(u, a)$. Then, the antagonist chooses a successor v of u such that $v \in Q_a$ for all $a \in A$, and the position of the game is updated to v. In this way, the game continues forever and produces a computation. The protagonist wins the game if the resulting computation satisfies the subformula ψ, read as a linear temporal formula whose outermost operator is $\bigcirc$, $\Box$, or $\mathcal{U}$. The ATL formula $\langle\langle A\rangle\rangle \psi$ holds at the state q if the protagonist has a winning strategy in this game.

In order to define the semantics of ATL formally, we first define the notion of strategies. Consider an ATS $S = \langle \Pi, \Sigma, Q, \pi, \delta\rangle$. A *strategy* for an agent $a \in \Sigma$ is a mapping $f_a : Q^+ \to 2^Q$ such that for all $\lambda \in Q^*$ and all $q \in Q$, we have $f_a(\lambda \cdot q) \in \delta(q, a)$. Thus, the strategy f_a maps each finite prefix $\lambda \cdot q$ of a computation to a set in $\delta(q, a)$. This set contains possible extensions of the computation as suggested to agent a by the strategy. Each strategy f_a induces a set of computations that agent a can enforce. Given a state q, a set A of agents, and a set $F_A = \{f_a \mid a \in A\}$ of strategies, one for each agent in A, we define the *outcomes* of F_A from q to be the set $out(q, F_A)$ of all q-computations that the agents in A can enforce when they cooperate and follow the strategies in F_A; that is, a computation $\lambda = q_0, q_1, q_2, \ldots$ is in $out(q, F_A)$ if $q_0 = q$ and for all positions $i \geq 0$, the state q_{i+1} is a successor of q_i satisfying $q_{i+1} \in \bigcap_{a \in A} f_a(\lambda[0, i])$.

We can now turn to a formal definition of the semantics of ATL. We write $S, q \models \varphi$ ("state q satisfies formula φ in the structure S") to indicate that the formula φ holds at state q of S. When S is clear from the context we omit it

and write $q \models \varphi$. The relation $\models$ is defined, for all states q of S, inductively as follows:

- For $p \in \Pi$, we have $q \models p$ iff $p \in \pi(q)$.
- $q \models \neg\varphi$ iff $q \not\models \varphi$.
- $q \models \varphi_1 \vee \varphi_2$ iff $q \models \varphi_1$ or $q \models \varphi_2$.
- $q \models \langle\langle A \rangle\rangle \bigcirc \varphi$ iff there exists a set F_A of strategies, one for each agent in A, such that for all computations $\lambda \in out(q, F_A)$, we have $\lambda[1] \models \varphi$.
- $q \models \langle\langle A \rangle\rangle \Box \varphi$ iff there exists a set F_A of strategies, one for each agent in A, such that for all computations $\lambda \in out(q, F_A)$ and all positions $i \geq 0$, we have $\lambda[1] \models \varphi$.
- $q \models \langle\langle A \rangle\rangle \varphi_1 \mathcal{U} \varphi_2$ iff there exists a set F_A of strategies, one for each agent in A, such that for all computations $\lambda \in out(q, F_A)$ there exists a position $i \geq 0$ such that $\lambda[i] \models \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \models \varphi_1$.

Note that the next operator $\bigcirc$ is local: $q \models \langle\langle A \rangle\rangle \bigcirc \varphi$ iff for each agent $a \in A$ there exists a set $Q_a \in \delta(q, a)$ such that for each state $q' \in \bigcap_{a \in A} Q_a$, if q' is a successor of q, then $q' \models \varphi$.

It is often useful to express an ATL formula in a dual form. For that, we use the path quantifier $[\![A]\!]$, for a set A of agents. While the ATL formula $\langle\langle A \rangle\rangle \psi$ intuitively means that the agents in A can cooperate to make ψ true (they can "enforce" ψ), the dual formula $[\![A]\!]\psi$ means that the agents in A cannot cooperate to make ψ false (they cannot "avoid" ψ). Using the path quantifier $[\![\ ]\!]$, we can write, for a set A of agents and an ATL formula φ, the ATL formula $[\![A]\!] \bigcirc \varphi$ for the ATL formula $\neg\langle\langle A \rangle\rangle \bigcirc \neg\varphi$, $[\![A]\!] \Box \varphi$ for $\neg\langle\langle A \rangle\rangle \Diamond \neg\varphi$, and $[\![A]\!] \Diamond \varphi$ for $\neg\langle\langle A \rangle\rangle \Box \neg\varphi$ (similar abbreviations can be defined for the dual of the $\mathcal{U}$ operator). Let us make this more precise. For a state q and a set Λ of q-computations, we say that the agents in A can *enforce* the set Λ of computations if $out(q, F_A) \subseteq \Lambda$ for some set F_A of strategies for the agents in A. Dually, we say that the agents in A can *avoid* the set Λ of computations if $\Lambda \cap out(q, F_A) = \emptyset$ for some set F_A of strategies for the agents in A. If the agents in A can enforce a set Λ of computations, then the agents in $\Sigma \setminus A$ cannot avoid Λ. Therefore, $q \models \langle\langle A \rangle\rangle \psi$ implies $q \models [\![\Sigma \setminus A]\!] \psi$. The converse of this statement is not necessarily true. To see this, consider $\Sigma = \{a, b\}$, $\delta(q, a) = \{\{q_1, q_2\}, \{q_3, q_4\}\}$ and $\delta(q, b) = \{\{q_1, q_3\}, \{q_2, q_4\}\}$, assuming each state q_i satisfies the proposition p_i and no other propositions. Then $q \not\models \langle\langle a \rangle\rangle \bigcirc (p_1 \vee p_4)$ and $q \not\models [\![b]\!] \bigcirc (p_1 \vee p_4)$; that is, neither does a have a strategy to enforce $\bigcirc (p_1 \vee p_4)$ nor does b have a strategy to avoid $\bigcirc (p_1 \vee p_4)$.

Example 6. Recall the turn-based synchronous ATS S_1 from Example 2. Recall that in a turn-based synchronous ATS, every state is labeled with an agent that determines the successor state. In this simplified setting, to determine the truth of a formula with path quantifier $\langle\langle A \rangle\rangle$, we can consider the following simpler version of the ATL game. In every state u, if the agent scheduled to proceed in u belongs to A, then the protagonist updates the position to some successor

of u, and otherwise, the antagonist updates the position to some successor of u. Therefore, every state of S_1 satisfies the following ATL formulas:

1. Whenever the train is out of the gate and does not have a grant to enter the gate, the controller can prevent it from entering the gate.
$$\langle\langle\ \rangle\rangle \Box((out_of_gate \wedge \neg grant) \rightarrow \langle\langle ctr \rangle\rangle \Box out_of_gate)$$
2. Whenever the train is out of the gate, the controller cannot force it to enter the gate.
$$\langle\langle\ \rangle\rangle \Box(out_of_gate \rightarrow [\![ctr]\!] \Box out_of_gate)$$
3. Whenever the train is out of the gate, the train and the controller can cooperate so the train will enter the gate.
$$\langle\langle\ \rangle\rangle \Box(out_of_gate \rightarrow \langle\langle ctr, train \rangle\rangle \Diamond in_gate)$$
4. Whenever the train is out of the gate, it can eventually request a grant for entering the gate, in which case the controller decides whether the grant is given or not.
$$\langle\langle\ \rangle\rangle \Box(out_of_gate \rightarrow \langle\langle train \rangle\rangle \Diamond(request \wedge (\langle\langle ctr \rangle\rangle \Diamond grant) \wedge (\langle\langle ctr \rangle\rangle \Box \neg grant))$$
5. Whenever the train is in the gate, the controller can force it out in the next step.
$$\langle\langle\ \rangle\rangle \Box(in_gate \rightarrow \langle\langle ctr \rangle\rangle \bigcirc out_of_gate)$$

These natural requirements cannot be stated in CTL or CTL*. Consider the first two ATL formulas. They provide more information than the CTL formula

$$\forall\Box(out_of_gate \rightarrow \exists\Box out_of_gate).$$

While the CTL formula only requires the existence of a computation in which the train is always out of the gate, the two ATL formulas guarantee that no matter how the train behaves, the controller can prevent it from entering the gate, and no matter how the controller behaves, the train can decide to stay out of the gate. By contrast, since the train and the controller are the only agents in this example, the third ATL formula is equivalent to the CTL formula

$$\forall\Box\,(out_of_gate \rightarrow \exists\Diamond\, in_gate).$$

□

Turn-based synchronous ATS It is worth noting that in the special case of a turn-based synchronous ATS, the agents in A can enforce a set Λ of computations iff the agents in $\Sigma \setminus A$ cannot avoid Λ. Therefore, for all states q of a turn-based synchronous ATS, $q \models \langle\langle A \rangle\rangle \psi$ iff $q \models [\![\Sigma \setminus A]\!] \psi$, or equivalently, $[\![A]\!] = \langle\langle \Sigma \setminus A \rangle\rangle$. Due to this strong duality, over turn-based synchronous ATS, we can define the temporal operator $\Box$ from $\Diamond$: $\langle\langle A \rangle\rangle \Box \varphi = [\![\Sigma \setminus A]\!] \Box \varphi = \neg [\![A]\!] \Diamond \neg \varphi = \neg \langle\langle \Sigma \setminus A \rangle\rangle \Diamond \neg \varphi$.

Single-agent ATS Recall that a labeled transition system is an ATS with the single agent *sys*. In this case, which is a special case of turn-based synchronous, there are only two path quantifiers: $\langle\langle sys \rangle\rangle = [\![\ ]\!]$ and $\langle\langle\ \rangle\rangle = [\![sys]\!]$. Then each set $out(q, \{f_{sys}\})$ of outcomes contains a single q-computation, and each set $out(q, \emptyset)$ of outcomes contains all q-computations. Accordingly, the path quantifiers $\langle\langle sys \rangle\rangle$ and $\langle\langle\ \rangle\rangle$ are equal, respectively, to the existential and universal path quantifiers $\exists$ and $\forall$ of the logic CTL. In other words, over labeled transition systems, ATL is identical to CTL. We write, over arbitrary ATS, $\exists$ for the path quantifier $\langle\langle \Sigma \rangle\rangle$, and $\forall$ for the path quantifier $[\![\Sigma]\!]$. This is because, regarding $\exists\psi$, all agents can cooperate to enforce a condition ψ iff there exists a computation that fulfills ψ, and regarding $\forall\psi$, all agents cannot cooperate to avoid ψ iff all computations fulfill ψ.

3.3 Fair-ATL

Since fairness constraints rule out certain computations, in their presence we need to refine the interpretation of formulas of the form $\langle\langle A \rangle\rangle \psi$. In particular, in the Fair-ATL game we require the antagonist to satisfy all fairness constraints. This leads us to the following definition. The logic Fair-ATL has the same syntax as ATL. The formulas of Fair-ATL are interpreted over an ATS S, a fairness condition Γ for S, and a state q of S. The satisfaction relation $S, \Gamma, q \models_F \varphi$ ("state q *fairly* satisfies formula φ in the structure S with respect to fairness condition Γ") for propositions and boolean connectives is defined as in the case of ATL. Moreover:

- $q \models_F \langle\langle A \rangle\rangle \bigcirc \varphi$ iff there exists a set F_A of strategies, one for each agent in A, such that for all $\langle \Gamma, \Sigma \setminus A \rangle$-fair computations $\lambda \in out(q, F_A)$, we have $\lambda[1] \models_F \varphi$.
- $q \models \langle\langle A \rangle\rangle \Box \varphi$ iff there exists a set F_A of strategies, one for each agent in A, such that for all $\langle \Gamma, \Sigma \setminus A \rangle$-fair computations $\lambda \in out(q, F_A)$ and all positions $i \geq 0$, we have $\lambda[1] \models \varphi$.
- $q \models_F \langle\langle A \rangle\rangle \varphi_1 \, \mathcal{U} \, \varphi_2$ iff there exists a set F_A of strategies, one for each agent in A, such that for all $\langle \Gamma, \Sigma \setminus A \rangle$-fair computations $\lambda \in out(q, F_A)$ there exists a position $i \geq 0$ such that $\lambda[i] \models_F \varphi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j] \models_F \varphi_1$.

Note that the path quantifier $\langle\langle A \rangle\rangle$ ranges over the computations that are fair only with respect to the agents in $\Sigma \setminus A$. To see why, observe that once Γ contains a fairness constraint γ for which there exists an agent $a \in A$ such that $\gamma(q, a)$ is nontrivial for some state q (that is, $\emptyset \subset \gamma(q, a) \subset \delta(q, a)$), the agents in A can enforce computations that are not $\langle \Gamma, \Sigma \rangle$-fair. The above definition assures that the agents in A do not accomplish their tasks in such a vacuous way, by violating fairness.

Example 7. Consider the ATS S_1 from Example 2. Unless the controller cooperates with the train, there is no guarantee that the train eventually enters the

gate:

$$q_0 \not\models \langle\langle train \rangle\rangle \Diamond in_gate$$

So suppose we add a fairness condition $\Gamma_1 = \{\gamma_{ctr}\}$ for S_1, which imposes fairness on the control decisions in state q_1, namely, $\gamma_{ctr}(q_1, ctr) = \{\{q_2\}\}$ (all other values of γ_{ctr} are empty). If we interpret γ_{ctr} as a strong fairness constraint, then the train has a strategy to eventually enter the gate:

$$q_0 \models_F \langle\langle train \rangle\rangle \Diamond in_gate$$

To see this, whenever the train is in q_0, let it move to q_1. Eventually, due to the strong fairness constraint, the controller will move to q_2. Then the train can move to q_3. On the other hand, if we interpret γ_{ctr} as a weak fairness constraint, cooperation between the train and the controller is still required to enter the gate, and the Fair-ATL formula is not satisfied in q_0. To see this, note that the train cannot avoid the weakly $\langle \gamma_{ctr}, \{train, ctr\}\rangle$-fair computation $q_0, q_1, q_0, q_1, \ldots$ □

3.4 ATL*

The logic ATL is a fragment of a more expressive logic called ATL*. There are two types of formulas in ATL*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific computation. Formally, an ATL* state formula is one of the following:

(S1) p, for propositions $p \in \Pi$.
(S2) $\neg\varphi$ or $\varphi_1 \vee \varphi_2$, where φ, φ_1, and φ_2 are ATL* state formulas.
(S3) $\langle\langle A \rangle\rangle \psi$, where $A \subseteq \Sigma$ is a set of agents and ψ is an ATL* path formula.

An ATL* path formula is one of the following:

(P1) An ATL* state formula.
(P2) $\neg\psi$ or $\psi_1 \vee \psi_2$, where ψ, ψ_1, and ψ_2 are ATL* path formulas.
(P3) $\bigcirc\psi$ or $\psi_1 \mathcal{U} \psi_2$, where ψ, ψ_1, and ψ_2 are ATL* path formulas.

The logic ATL* consists of the set of state formulas generated by the rules (S1–3). The logic ATL* is similar to the branching-time temporal logic CTL*, only that path quantification is parameterized by agents. Additional boolean connectives and temporal operators are defined from $\neg$, $\vee$, $\bigcirc$, and $\mathcal{U}$ in the usual manner; in particular, $\Diamond\psi = true\,\mathcal{U}\,\psi$ and $\Box\psi = \neg\Diamond\neg\psi$. As with ATL, we use the dual path quantifier $[\![A]\!]\psi = \neg\langle\langle A \rangle\rangle\neg\psi$, and the abbreviations $\exists = \langle\langle \Sigma \rangle\rangle$ and $\forall = [\![\Sigma]\!]$. The logic ATL can be viewed as the fragment of ATL* that consists of all formulas in which every temporal operator is immediately preceded by a path quantifier.

The semantics of ATL* formulas is defined with respect to an ATS S. We write $S, \lambda \models \psi$ to indicate that the path formula ψ holds at computation λ of the structure S. The satisfaction relation $\models$ is defined, for all states q and computations λ of S, inductively as follows:

- For state formulas generated by the rules (S1–2), the definition is the same as for ATL.
- $q \models \langle\langle A \rangle\rangle \psi$ iff there exists a set F_A of strategies, one for each agent in A, such that for all computations $\lambda \in out(q, F_A)$, we have $\lambda \models \psi$.
- $\lambda \models \varphi$ for a state formula φ iff $\lambda[0] \models \varphi$.
- $\lambda \models \neg\psi$ iff $\lambda \not\models \psi$.
- $\lambda \models \psi_1 \vee \psi_2$ iff $\lambda \models \psi_1$ or $\lambda \models \psi_2$.
- $\lambda \models \bigcirc\psi$ iff $\lambda[1, \infty] \models \psi$.
- $\lambda \models \psi_1 \mathcal{U} \psi_2$ iff there exists a position $i \geq 0$ such that $\lambda[i, \infty] \models \psi_2$ and for all positions $0 \leq j < i$, we have $\lambda[j, \infty] \models \psi_1$.

For example, the ATL* formula

$$\langle\langle a \rangle\rangle((\Diamond\Box req) \vee (\Box\Diamond grant))$$

asserts that agent a has a strategy to enforce computations in which only finitely many requests are sent or infinitely many grants are given. Such a requirement cannot be expressed in CTL* or in ATL. Since weak and strong fairness constraints can be expressed within ATL* (provided appropriate propositions are available), there is no need for Fair-ATL*,

Remark. In the definitions of ATL and ATL*, the strategy of an agent may depend on an unbounded amount of information, namely, the full history of the game up to the current state. When we consider finite ATS, all involved games are ω-regular. Then, the existence of a winning strategy implies the existence of a winning *finite-state* strategy [Rab70], which depends only on a finite amount of information about the history of the game. Thus, the semantics of ATL and ATL* with respect to finite ATS can be defined, equivalently, using the outcomes of finite-state strategies only. This is interesting, because a strategy can be thought of as the parallel composition of the system with a *controller*, which makes sure that the system follows the strategy. Then, for an appropriate definition of parallel composition, finite-state strategies can be implemented using finite ATS. Indeed, for the finite reachability games and generalized Büchi games of ATL, it suffices to consider *memory-free* strategies [EJ88], which can be implemented as control maps (i.e., controllers without state). This is not the case for ATL*, whose formulas can specify the winning positions of Streett games [Tho95].

4 Symbolic Model Checking

4.1 ATL Symbolic Model Checking

The *model-checking problem* for ATL asks, given an ATS $S = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$ and an ATL formula φ, for the set $[\varphi] \subseteq Q$ of states of S that satisfy φ. ATL Model checking is similar to CTL model checking [CE81, QS81, BCM$^+$90]. We present a *symbolic* algorithm, which manipulates state sets of S. The algorithm is shown in Figure 3, and uses the following primitive operations:

- The function Sub, when given an ATL formula φ, returns a queue $Sub(\varphi)$ of subformulas of φ such that if φ_1 is a subformula of φ and φ_2 is a subformula of φ_1, then φ_2 precedes φ_1 in the queue $Sub(\varphi)$.
- The function Reg, when given a proposition $p \in \Pi$, returns the state set $[p]$.
- The function Pre, when given a set $A \subseteq \Sigma$ of agents and a set $\tau \subseteq Q$ of states, returns the set containing all states q such that whenever S is in state q, the agents in A can cooperate and force the next state to lie in τ. Formally, $Pre(A, \tau)$ contains state $q \in Q$ iff for each agent $a \in A$ there exists a set $Q_a \in \delta(q, a)$ such that for each state $q' \in \bigcap_{a \in A} Q_a$, if q' is a successor of q, then $q' \in \tau$.
- Union, intersection, difference, and inclusion test for state sets.

These primitives can be implemented using symbolic representations, such as binary decision diagrams, for state sets and the transition relation. If given a symbolic model checker for CTL, such as SMV [McM93], only the Pre operation needs to be modified for checking ATL. In the special case that the ATS S is turn-based synchronous, the computation of the function Pre used in the symbolic model checking is particularly simple. Recall that in this case, $\sigma(q)$ denotes the agent that is scheduled to proceed in state q. Then, when given a set A of agents and a set τ of states, $Pre(A, \tau)$ returns the set containing all states q such that either $\sigma(q) \in A$ and some successor of q is in τ, or $\sigma(q) \notin A$ and all successors of q are in τ.

foreach φ' **in** $Sub(\varphi)$ **do**
  case $\varphi' = p$: $[\varphi'] := Reg(p)$
  case $\varphi' = \neg\theta$: $[\varphi'] := [true] \setminus [\theta]$
  case $\varphi' = \theta_1 \vee \theta_2$: $[\varphi'] := [\theta_1] \cup [\theta_2]$
  case $\varphi' = \langle\langle A \rangle\rangle \bigcirc \theta$: $[\varphi'] := Pre(A, [\theta])$
  case $\varphi' = \langle\langle A \rangle\rangle \Box \theta$:
    $\rho := [true]$; $\tau := [\theta]$;
    while $\rho \not\subseteq \tau$ **do** $\rho := \rho \cap \tau$; $\tau := Pre(A, \rho) \cap [\theta]$ **od**;
    $[\varphi'] := \rho$
  case $\varphi' = \langle\langle A \rangle\rangle \theta_1 \, \mathcal{U} \, \theta_2$:
    $\rho := [false]$; $\tau := [\theta_2]$;
    while $\tau \not\subseteq \rho$ **do** $\rho := \rho \cup \tau$; $\tau := Pre(A, \rho) \cap [\theta_1]$ **od**;
    $[\varphi'] := \rho$
  end case
od;
return $[\varphi]$.

Fig. 3. ATL symbolic model checking

4.2 Fair-ATL Symbolic Model Checking

We turn our attention to the model-checking problem for Fair-ATL: given an ATS $S = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$, a fairness condition Γ for S, and a Fair-ATL formula φ, compute the set $[\varphi]_F$ of states of S that fairly satisfy φ with respect to Γ. We use the weak interpretation for Γ; the case of strong fairness constraints can be handled similarly. Recall that to evaluate a formula of the form $\langle\langle A \rangle\rangle \psi$, we need to restrict attention to computations that satisfy all fairness constraints for agents not in A. To determine which fairness constraints are satisfied by a computation, we augment the state space by adding new propositions that indicate for each agent $a \in \Sigma$ and each fairness constraint γ, whether or not γ is a-enabled, and whether or not γ is a-taken. For this purpose, we define the ATS $S_F = \langle \Pi_F, \Sigma, Q_F, \pi_F, \delta_F \rangle$:

- For every agent $a \in \Sigma$ and every fairness constraint $\gamma \in \Gamma$, there is a new proposition $\langle \gamma, a, \mathit{enabled} \rangle$ and a new proposition $\langle \gamma, a, \mathit{taken} \rangle$: $\Pi_F = \Pi \cup (\Gamma \times \Sigma \times \{\mathit{enabled}, \mathit{taken}\})$.
- The states of S_F correspond to the transitions of S: $Q_F = \{\langle q, q' \rangle \mid q' \text{ is a successor of } q \text{ in } S\}$.
- For every state $\langle q, q' \rangle \in Q_F$ and every agent $a \in \Sigma$, the transition $\delta_F(\langle q, q' \rangle, a)$ is obtained from $\delta(q', a)$ by replacing each state q'' appearing in $\delta(q', a)$ by the state $\langle q', q'' \rangle$. For example, if $\delta(q_0, a) = \{\{q_1, q_2\}, \{q_3\}\}$, then

$$\delta_F(\langle q, q_0 \rangle, a) = \{\{\langle q_0, q_1 \rangle, \langle q_0, q_2 \rangle\}, \{\langle q_0, q_3 \rangle\}\}.$$

- For every state $\langle q, q' \rangle \in Q_F$, we have

$$\begin{aligned} \pi_F(\langle q, q' \rangle) \;=\; & \pi(q) \cup \{\langle \gamma, a, \mathit{enabled} \rangle \mid \gamma(q, a) \neq \emptyset\} \cup \\ & \{\langle \gamma, a, \mathit{taken} \rangle \mid \text{ there exists } Q' \in \gamma(q, a) \text{ such that } q' \in Q'\}. \end{aligned}$$

Intuitively, a state of the form $\langle q, q' \rangle$ in S_F corresponds to the ATS S being in state q with the agents deciding that the successor of q will be q'. There is a one-to-one correspondence between computations of S and S_F, and between strategies in S and S_F. The new propositions in $\Gamma \times \Sigma \times \{\mathit{enabled}, \mathit{taken}\}$ allow us to identify the fair computations. Consequently, evaluating formulas of Fair-ATL over states of S can be reduced to evaluating, over states of S_F, ATL* formulas that encode the fairness constraints in Γ.

Proposition1. *A state q of the ATS S fairly satisfies the Fair-ATL formula $\langle\langle A \rangle\rangle \psi$ with respect to the fairness condition Γ iff for each agent $a \in A$, there exists a set $Q_a \in \delta(q, a)$ such that for every successor q' of q with $q' \in \bigcap_{a \in A} Q_a$, the state $\langle q, q' \rangle$ of the ATS S_F satisfies the ATL* formula*

$$\langle\langle A \rangle\rangle \Big(\psi \;\vee \bigvee_{\gamma \in \Gamma, a \in \Sigma \setminus A} \Diamond\Box(\langle \gamma, a, \mathit{enabled} \rangle \wedge \neg \langle \gamma, a, \mathit{taken} \rangle)\Big).$$

This proposition reduces Fair-ATL model checking to a special case of ATL* model checking. Rather than presenting the model-checking algorithm for full Fair-ATL, we consider the sample formula $\langle\langle A\rangle\rangle\Diamond p$, for a proposition p. Consider the following game on the structure S_F. When a state labeled by p is visited, the protagonist wins. If the game continues forever, then the protagonist wins iff the resulting computation is not weakly $\langle \Gamma, \Sigma \setminus A\rangle$-fair. The winning condition for the antagonist can therefore be specified by the LTL formula

$$\Box(\neg p \ \wedge \bigwedge_{\gamma\in\Gamma, a\in\Sigma\setminus A} \Diamond(\neg\langle\gamma, a, enabled\rangle \vee \langle\gamma, a, taken\rangle)).$$

This is a generalized Büchi condition. The set of winning states in such a game can be computed using nested fixed points. To obtain an algorithm for this example, we note that the CTL* formula $\exists\Box(p \wedge \bigwedge_{1\le i\le k}\Diamond p_i)$ can be computed symbolically as the greatest fixpoint

$$\nu X.(p \wedge \exists\bigcirc(p\,\mathcal{U}\,(p_1 \wedge p\,\mathcal{U}\,(p_2 \wedge \ldots \wedge p\,\mathcal{U}\,(p_k \wedge X))))).$$

Consequently, the algorithm of Figure 4 computes the set $\hat\rho \subseteq Q_F$ of winning states for the protagonist. The function Pre_F is like Pre, but operates on the structure S_F. By Proposition 1, the first projection of $\hat\rho$ gives the desired set $[\langle\langle A\rangle\rangle\Diamond p]_F \subseteq Q$ of states in the original structure S.

$\rho := [true]$; $\tau := [\neg p]$;
while $\rho \not\subseteq \tau$ **do**
  $\rho := \rho \cap \tau$;
  foreach $\gamma \in \Gamma$ **do**
    foreach $a \in \Sigma \setminus A$ **do**
      $\rho'' := [\rho] \cap (([true] \setminus Reg(\langle\gamma, a, enabled\rangle)) \cup Reg(\langle\gamma, a, taken\rangle))$;
      $\rho' := [false]$; $\tau := [\rho] \cap \rho''$;
      while $\tau' \not\subseteq \rho'$ **do** $\rho' := \rho' \cup \tau'$; $\tau' := Pre_F(\Sigma \setminus A, \rho') \cap [\neg p])$ **od**;
      $\rho := \tau'$;
    od
  od;
  $\tau := Pre_F(\Sigma \setminus A, \rho) \cap [\neg p]$
od;
return $\hat\rho := [true] \setminus \tau$

Fig. 4. Nested fixed-point computation for Fair-ATL symbolic model checking

5 Model-checking Complexity

We measure the complexity of the model-checking problem in two different ways: the *joint complexity* of model checking considers the complexity in terms of

both the structure and the formula; the *structure complexity* of model checking (called "program complexity" in [VW86a]) considers the complexity in terms of the structure, assuming the formula is fixed. Since the structure is typically much larger than the formula, and its size is the most common computational bottle-neck [LP85], the structure-complexity measure is of particular practical interest.

5.1 ATL Model-checking Complexity

Theorem 2. *The model-checking problem for ATL is PTIME-complete, and can be solved in time $O(m\ell)$ for an ATS with m transitions and an ATL formula of length ℓ. The structure complexity of the problem is also PTIME-complete, even in the special case of turn-based synchronous ATS.*

Proof: Consider an ATS S with m transitions and an ATL formula φ of length ℓ. We claim that the algorithm presented in Figure 3 can be implemented in time $O(m\ell)$. To see this, observe that the size of $Sub(\varphi)$ is bounded by ℓ, and that executing each of the case statements in the algorithm involves, at most, a calculation of a single fixed point, which can be done in time linear in m (see [Cle93]). Since reachability in AND-OR graphs is known to be PTIME-hard [Imm81], and can be specified using the fixed ATL formula $\langle\langle a \rangle\rangle \Diamond p$ interpreted over a turn-based synchronous ATS, hardness in PTIME, for both the joint and the structure complexity, is immediate. □

It is interesting to compare the model-checking complexities of turn-based synchronous ATL and CTL. While the two problems can be solved in time $O(m\ell)$ [CES86], the structure complexity of CTL model checking is only NLOGSPACE-complete [BVW94]. This is because CTL model checking is related to graph reachability, whereas turn-based synchronous ATL model checking is related to AND-OR graph reachability.

5.2 Fair-ATL Model-checking Complexity

As in Section 4.2, we consider the case of fairness constraints.

Theorem 3. *The model-checking problem for Fair-ATL is PTIME-complete, and can be solved in time $O(m^2 n^2 c^2 \ell)$ for a fair ATS with m transitions and n agents, c weak fairness constraints, and an ATL formula of size ℓ. The structure complexity of the problem is also PTIME-complete.*

Proof: Consider an ATS S with m transitions, n agents, and c weak fairness constraints. Let φ be a Fair-ATL formula. Each state of S is labeled with each subformula of φ, starting with the innermost subformulas. Let us consider the case corresponding to a subformula of the form $\langle\langle A \rangle\rangle \Diamond \theta$ (the cases corresponding to $\Box$ and $\mathcal{U}$ are similar). As described in Section 4.2, we first construct the ATS S', and the truth of $\langle\langle A \rangle\rangle \Diamond \theta$ can be evaluated by solving a generalized Büchi

game over the structure S'. The number of transitions in S' equals m. Note that the winning condition for the antagonist corresponds to visiting, for each fairness constraint γ and each agent $a \notin A$, infinitely often a state satisfying $\langle\langle \gamma, a, \mathit{taken} \rangle\rangle \vee \neg \langle\langle \gamma, a, \mathit{enabled} \rangle\rangle$. Thus, there are cn Büchi constraints. Since the complexity of solving Büchi games is quadratic (use the nested fixed-point computation of Figure 4), the cost of processing a temporal connective is $O(m^2 n^2 c^2)$. This concludes the upper bound. Since the model-checking problem for ATL is a special case of the model-checking problem for Fair-ATL (with $\Gamma = \emptyset$), hardness in PTIME follows from Theorem 2. □

5.3 ATL* Model-checking Complexity

We have seen that the transition from CTL to ATL does not involve a substantial computational price. In this section we consider the model-checking complexity of ATL*. While there is an exponential price to pay in model-checking complexity when moving from CTL to CTL*, this price becomes even more significant (namely, doubly exponential) when we consider the alternating-time versions of both logics.

Before we discuss ATL* model checking, let us briefly recall CTL* model checking [EL85]. The computationally difficult case corresponds to evaluating a state formula of the form $\exists\psi$, for an LTL formula ψ. The solution is to construct a Büchi automaton $\mathcal{A}$ that accepts all computations that satisfy ψ. To determine whether a state q satisfies the formula $\exists\psi$, we need to check if some q-computation is accepted by the automaton $\mathcal{A}$, and this can be done by analyzing the product of $\mathcal{A}$ with the structure. The complexity of CTL* model checking reflects the cost of translating LTL formulas to ω-automata. In case of an ATL* state formula $\langle\langle A \rangle\rangle\psi$, the solution is similar, but requires the use of tree automata, because satisfaction corresponds to the existence of winning strategies. Therefore, model checking requires checking the nonemptiness of the intersection of two tree automata: one accepting trees in which all paths satisfy ψ, and the other accepting trees that correspond to possible strategies of the protagonist.

In order to solve the model-checking problem for ATL*, we first define the notion of execution trees. Consider an ATS S, a set A of agents, and a set $F_A = \{f_a \mid a \in A\}$ of strategies for the agents in A. For a state q of S, the set $\mathit{out}(q, F_A)$ of q-computations is fusion-closed, and therefore induces a tree $\mathit{exec}(q, F_A)$. Intuitively, the tree $\mathit{exec}(q, F_A)$ is obtained by unwinding S starting from q according to the successor relation, while pruning subtrees whose roots are not chosen by the strategies in F_A. Formally, the tree $\mathit{exec}(q, F_A)$ has as nodes the following elements of Q^*:

- q is a node (the root).
- For a node $\lambda \cdot q' \in Q^*$, the successor nodes (children) of $\lambda \cdot q'$ are all strings of the form $\lambda \cdot q' \cdot q''$, where q'' is a successor of q' and $q'' \in \bigcap_{a \in A} f_a(\lambda \cdot q')$.

A tree t is a $\langle q, A\rangle$-*execution tree* if there exists a set F_A of strategies, one for each agent in A, such that $t = exec(q, F_A)$.

Theorem 4. *The model-checking problem for ATL* is 2EXPTIME-complete, even in the special case of turn-based synchronous ATS. The structure complexity of the problem is PTIME-complete.*

Proof: Consider an ATS S and an ATL* formula φ. As in the algorithm for CTL* model checking, we label each state q of S by all state subformulas of φ that are satisfied in q. We do this in a bottom-up fashion, starting from the innermost state subformulas of φ. For subformulas generated by the rules (S1–2), the labeling procedure is straightforward. For subformulas φ' generated by (S3), we employ the algorithm for CTL* module checking [KV96] as follows. Let $\varphi' = \langle\langle A\rangle\rangle\psi$; since the satisfaction of all state subformulas of ψ has already been determined, we can assume that ψ is an LTL formula. We construct a Rabin tree automaton $\mathcal{A}_\psi$ that accepts precisely the trees satisfying the CTL* formula $\forall\psi$, and for each state q of S, we construct a Büchi tree automaton $\mathcal{A}_{S,q,A}$ that accepts precisely the $\langle q, A\rangle$-execution trees. The automaton $\mathcal{A}_\psi$ has $2^{2^{O(|\psi|)}}$ states and $2^{O(|\psi|)}$ Rabin pairs [ES84]. The automaton $\mathcal{A}_{S,q,A}$ has $|Q|$ states. The product of the two automata $\mathcal{A}_\psi$ and $\mathcal{A}_{S,q,A}$ is a Rabin tree automaton that accepts precisely the $\langle q, A\rangle$-execution trees satisfying $\forall\psi$. Hence, $q \models \langle\langle A\rangle\rangle\psi$ iff the product automaton is nonempty. The nonemptiness problem for a Rabin tree automaton with n states and r pairs can be solved in time $O(nr)^{3r}$ [EJ88, PR89a]. Hence, labeling a single state with φ' requires at most time $(|Q| \cdot 2^{2^{|\psi|}})^{2^{O(|\psi|)}} = |Q|^{2^{O(|\psi|)}}$. Since there are $|Q|$ states and at most $|\varphi|$ subformulas, membership in 2EXPTIME follows.

For the lower bound, we use a reduction from the realizability problem for LTL [PR89a], which is shown to be 2EXPTIME-hard in [Ros92]. In this problem, we are given an LTL formula ψ over a set Π of propositions and we determine whether there exists a turn-based synchronous ATS S with two agents, *sys* and *env*, such that

1. the transitions in S alternate between *sys* states and *env* states,
2. every *env* state has 2^Π successors, each labeled by a different subset of 2^Π, and
3. some state of S satisfies $\langle\langle sys\rangle\rangle\psi$.

Intuitively, a state of S that satisfies $\langle\langle sys\rangle\rangle\psi$ witnesses a strategy of the system to satisfy ψ irrespective of what the environment does. Let S_Π be the maximal two-agent turn-based synchronous ATS over Π that alternates between *sys* and *env* states:

$$S_\Pi = \langle \Pi, \{sys, env\}, 2^\Pi \times \{s, e\}, \pi, \sigma, (2^\Pi \times \{s\}) \times (2^\Pi \times \{e\}) \cup (2^\Pi \times \{e\}) \times (2^\Pi \times \{s\}) \rangle,$$

where for every $w \subseteq \Pi$, we have $\pi(\langle w, s\rangle) = \pi(\langle w, e\rangle) = w$, $\sigma(\langle w, s\rangle) = \{sys\}$, and $\sigma(\langle w, e\rangle) = \{env\}$. It is easy to see that ψ is realizable iff there exists some

state in S_Π that satisfies $\langle\langle sys \rangle\rangle \psi$. Since the 2EXPTIME lower bound holds already for LTL formulas with a fixed number of propositions, the size of S_Π is fixed, and we are done.

The lower bound for the structure complexity of the problem follows from Theorem 2, and the upper bound follows from fixing $|\psi|$ in the complexity analysis of the joint complexity above. □

6 Beyond ATL*

In this section we suggest two more formalisms for the specification of open systems. We compare the two formalisms with ATL and ATL* and consider their expressiveness and their model-checking complexity. Given two logics L_1 and L_2, we say that the logic L_1 is *as expressive* as the logic L_2 if for every formula φ_2 of L_2, there exists a formula φ_1 of L_1 such that φ_1 and φ_2 are equivalent (i.e., they are true in the same states of each ATS). The logic L_1 is *more expressive* than L_2 if L_1 is as expressive as L_2 and L_2 is not as expressive as L_1.

6.1 The Alternating-time μ-Calculus

The formulas of the logic AMC (*Alternating-time μ-Calculus*) are constructed from propositions, boolean connectives, the next operator $\bigcirc$, each occurrence parameterized by a set of agents, as well as the least fixed-point operator μ. Formally, given a set Π of propositions, a set V of propositional variables, and a set Σ of agents, an AMC formula is one of the following:

- p, for propositions $p \in \Pi$.
- X, for a propositional variables $X \in V$.
- $\neg\varphi$ or $\varphi_1 \vee \varphi_2$, where φ, φ_1, and φ_2 are AMC formulas.
- $\langle\langle A \rangle\rangle \bigcirc \varphi$, where $A \subseteq \Sigma$ is a set of agents and φ is an AMC formula.
- $\mu X.\varphi$, where φ is an AMC formula in which all free occurrences of X (i.e., those that do not occur in a subformula of φ starting with μX) fall under an even number of negations.

The logic AMC is similar to the μ-calculus of [Koz83], only that the next operator $\bigcirc$ is parameterized by sets of agents rather than by a universal or an existential path quantifier. Additional boolean connectives are defined from $\neg$ and $\vee$ in the usual manner. As with ATL, we use the dual $[\![A]\!]\bigcirc\varphi = \neg\langle\langle A \rangle\rangle \bigcirc \neg\varphi$, and the abbreviations $\exists = \langle\langle \Sigma \rangle\rangle$ and $\forall = [\![\Sigma]\!]$. As with the μ-calculus, we write $\nu X.\varphi$ to abbreviate $\neg\mu X.\neg\varphi$. Using both the greatest fixed-point operator ν, the dual next operator $[\![A]\!]\bigcirc$, and the connective $\wedge$, we can write every AMC formula in *positive normal form*, where all occurrences of $\neg$ are in front of propositions. An AMC formula φ is *alternation free* if when φ is written in positive normal form, there are no occurrences of ν (resp. μ) on any syntactic path from an occurrence of μX (resp. νX) to an occurrence of X. For example, the formula $\mu X.(p \vee$

$\mu Y.(X \vee \langle\langle a \rangle\rangle \bigcirc Y))$ is alternation free; the formula $\nu X.\mu Y.((p \wedge X) \vee \langle\langle a \rangle\rangle \bigcirc Y)$ is not. The *alternation-free fragment of AMC* contains only alternation-free formulas.

We now turn to the semantics of AMC. We first need some definitions and notations. Given an ATS $S = \langle \Pi, \Sigma, Q, \pi, \delta \rangle$, a *valuation* $\mathcal{V}$ is a function from the propositional variables V to subsets of Q. For a valuation $\mathcal{V}$, a propositional variable X, and a set $Q' \subseteq Q$ of states, we denote by $\mathcal{V}[X := Q']$ the valuation that maps X to Q' and agrees with $\mathcal{V}$ on all other variables. An AMC formula φ is interpreted as a mapping φ^S from valuations to state sets. Then, $\varphi^S(\mathcal{V})$ denotes the set of states that satisfy the AMC formula φ under the valuation $\mathcal{V}$. The mapping φ^S is defined inductively as follows:

- For a proposition $p \in \Pi$, we have $p^S(\mathcal{V}) = \{q \in Q \mid p \in \pi(q)\}$.
- For a propositional variable $X \in V$, we have $X^S(\mathcal{V}) = \mathcal{V}(X)$.
- $(\neg\varphi)^S(\mathcal{V}) = Q \setminus \varphi^S(\mathcal{V})$.
- $(\varphi_1 \vee \varphi_2)^S(\mathcal{V}) = \varphi_1^S(\mathcal{V}) \cup \varphi_2^S(\mathcal{V})$.
- $(\langle\langle A \rangle\rangle \bigcirc \varphi)^S(\mathcal{V}) = \{q \in Q \mid$ for each agent $a \in A$, there exists a set $Q_a \in \delta(q, a)$ such that for each state $q' \in \bigcap_{a \in A} Q_a$, if q' is a successor of q, then $q' \in \varphi^S(\mathcal{V})\}$.
- $(\mu X.\varphi)^S(\mathcal{V}) = \bigcap\{Q' \subseteq Q \mid \varphi^S(\mathcal{V}[X := Q']) \subseteq Q'\}$.

Consider an AMC formula of the form $\mu X.\varphi$. Then, given a valuation $\mathcal{V}$, the subformula φ can be viewed as a function $h^S_{\varphi,\mathcal{V}}$ that maps each state set $Q' \subseteq Q$ to the state set $\varphi^S(\mathcal{V}[X := Q'])$. Since all free occurrences of X fall under an even number of negations, the function $h^S_{\varphi,\mathcal{V}}$ is monotonic; that is, if $Q' \subseteq Q''$, then $h^S_{\varphi,\mathcal{V}}(Q) \subseteq h^S_{\varphi,\mathcal{V}}(Q'')$. Consequently, by standard fixed-point theory, the function $h^S_{\varphi,\mathcal{S}}$ has a least fixed-point, namely, $\bigcap\{Q' \subseteq Q \mid \varphi^S(\mathcal{V}[X := Q']) \subseteq Q'\}$. Furthermore, if each state has only finitely many successor states, the function $h^S_{\varphi,\mathcal{V}}$ is continuous, and the least fixed-point can be computed by iterative approximation starting from $X = [false]$:

$$(\mu X.\varphi)^S(\mathcal{V}) \;=\; \bigcap_{i \geq 0} (h^S_{\varphi,\mathcal{V}})^i([false]).$$

If the ATS S has only finitely many states, the intersection is finite, and the iterative approximation converges in a finite number of steps.

A *sentence* of AMC is a formula that contains no free occurrences of propositional variables. Sentences φ define the same mapping φ^S for any and all valuations. Therefore, for a state q of S and a sentence φ, we write $S, q \models \varphi$ ("state q satisfies formula φ in structure S") iff $q \in \varphi^S$. For example, the AMC formula $\mu X.(q \vee (p \wedge \langle\langle A \rangle\rangle \bigcirc X))$ is equivalent to the ATL formula $\langle\langle A \rangle\rangle p \,\mathcal{U}\, q$.

AMC expressiveness All temporal properties using the always and until operators can be defined as fixed points of next-time properties. For closed systems, this gives the μ-calculus as a generalization of temporal logics. It is known that the μ-calculus is more expressive than CTL*, and the alternation-free μ-calculus is more expressive than CTL. Similarly, and for the same reasons, AMC is more

expressive than ATL*, and its alternation-free fragment is more expressive than ATL.

Theorem 5. *AMC is more expressive than ATL*. The alternation-free fragment of AMC is more expressive than ATL.*

Proof: The translation from alternating-time temporal logics to AMC is very similar to the translation from branching-time temporal logics to μ-calculus [EL86], with $\langle\langle A\rangle\rangle\bigcirc$ replacing $\exists\bigcirc$. We describe here the translation of ATL formulas to the alternation-free fragment of AMC. For this, we present a function

$$g : \text{ATL formulas} \rightarrow \text{alternation-free AMC formulas}$$

such that for every ATL formula φ, the formulas φ and $g(\varphi)$ are equivalent. The function g is defined inductively as follows:

- For $p \in \Pi$, we have $g(p) = p$.
- $g(\neg\varphi) = \neg g(\varphi)$.
- $g(\varphi_1 \vee \varphi_2) = g(\varphi_1) \vee g(\varphi_2)$.
- $g(\langle\langle A\rangle\rangle\bigcirc\varphi) = \langle\langle A\rangle\rangle\bigcirc g(\varphi)$.
- $g(\langle\langle A\rangle\rangle\Box\varphi) = \nu X.(g(\varphi) \wedge \langle\langle A\rangle\rangle\bigcirc X)$.
- $g(\langle\langle A\rangle\rangle\varphi_1 \mathcal{U} \varphi_2) = \mu X.(f(\varphi_2) \vee (g(\varphi_1) \wedge \langle\langle A\rangle\rangle\bigcirc X))$.

To establish that AMC is more expressive than ATL*, and its alternation-free fragment is more expressive than ATL, note that for a single-agent ATS, (alternation-free) AMC is the same as the (alternation-free) μ-calculus, CTL* is the same as ATL*, and CTL is the same as ATL. □

The alternating-time μ-calculus, however, is not a natural and convenient specification language for reasoning about open systems. Writing and understanding formulas in the μ-calculus is hard already in the context of closed systems, and in practice, designers avoid the nonintuitive use of fixed points and prefer simple temporal operators (see [BBG$^+$94]). Using AMC as a specification language for open systems would require even more complicated formulas, with extra nesting of fixed points, making the μ-calculus even less appealing. So, just as CTL and CTL* capture useful and friendly subsets of the μ-calculus for the specification of closed system, ATL and ATL* capture useful and friendly subsets of AMC for the specification of open systems. This is because ATL and ATL* have as primitives parameterized path quantifiers, not just parameterized next-time operators.

AMC model checking Algorithms and tools for μ-calculus model checking can be easily modified to handle AMC. Indeed, the only difference between the μ-calculus and AMC is the definition of the next operator, which has a game-like interpretation in AMC. Hence, as in Section 4.1, the modification involves only the *Pre* function. Therefore, the complexity of the model-checking problem for the μ-calculus [EL86] implies the following.

Theorem 6. *The model-checking problem for the alternation-free fragment of AMC can be solved in time $O(m\ell)$ for an ATS with m transitions and a formula of size ℓ. The model-checking problem for AMC can be solved in time $O(m^{d+1})$ for an ATS with m transitions and an formula of alternation depth $d \geq 1$.*

AMC and propositional logic of games In [Par83], Parikh defines a propositional logic of games. Parikh's logic extends dynamic logics (e.g., PDL [FL79]) in a way similar to the way in which AMC extends the μ-calculus. The formulas in Parikh's logic are built with respect to a set of atomic games, which correspond to the choices of agents in an ATS. Cooperation between agents and fixed-point expressions are specified in Parikh's logic by the usual PDL operations, such as disjunction and iteration, on games. The alternation-free fragment of AMC can be embedded into Parikh's logic. For example, the AMC formula $\mu X.p \vee \langle\langle a, b\rangle\rangle \bigcirc X$ corresponds to the formula $\langle (a \vee b)* \rangle p$ in Parikh's logic. In [Par83], Parikh's logic is shown to be decidable and a complete set of axioms is given; the model-checking problem is not studied.

6.2 Game Logic

The parameterized path quantifier $\langle\langle A\rangle\rangle$ first stipulates the *existence* of strategies for the agents in A and then *universally* quantifies over the outcomes of the stipulated strategies. One may generalize ATL and ATL* by separating the two concerns into strategy quantifiers and path quantifiers, say, by writing $\exists A.\forall$ instead of $\langle\langle A\rangle\rangle$ (read $\exists A$ as "there exist strategies for the agents in A"). Then, for example, the formula $\hat{\varphi} = \exists A.(\exists\Box\varphi_1 \wedge \exists\Box\varphi_2)$ asserts that the agents in A have strategies such that for some behavior of the remaining agents, φ_1 is always true, and for some possibly different behavior of the remaining agents, φ_2 is always true.

We refer to the general logic with strategy quantifiers, path quantifiers, temporal operators, and boolean connectives as *game logic* (GL, for short). There are three types of formulas in GL: *state formulas*, whose satisfaction is related to a specific state of the given ATS S, *tree formulas*, whose satisfaction is related to a specific execution tree of S (for the definition of execution trees, recall Section 5.3), and *path formulas*, whose satisfaction is related to a specific computation of S. Formally, a GL state formula is one of the following:

(S1) p, for propositions $p \in \Pi$.
(S2) $\neg\varphi$ or $\varphi_1 \vee \varphi_2$, where φ, φ_1 and φ_2 are GL state formulas.
(S3) $\exists A.\theta$, where $A \subseteq \Sigma$ is a set of agents and θ is a GL tree formula.

A GL tree formula is one of the following:

(T1) φ, for a GL state formula φ.
(T2) $\neg\theta$ or $\theta_1 \vee \theta_2$, where θ, θ_1 and θ_2 are GL tree formulas.
(T3) $\exists\psi$, where ψ is a GL path formula.

A GL path formula is one of the following:

(P1) θ, for a GL tree formula θ.
(P2) $\neg\psi$ or $\psi_1 \vee \psi_2$, where ψ, ψ_1, and ψ_2 are GL path formulas.
(P3) $\bigcirc\psi$ or $\psi_1 \mathcal{U} \psi_2$, where ψ, ψ_1, and ψ_2 are GL path formulas.

The logic GL consists of the set of state formulas generated by the rules (S1–3). For instance, while the formula $\hat{\varphi}$ from above is a GL (state) formula, its subformula $\exists\Box\varphi_1 \wedge \exists\Box\varphi_2$ is a tree formula.

We now define the semantics of GL. We write $S, q \models \varphi$ to indicate that the state formula φ holds at state q of the structure S. We write $S, t \models \theta$ to indicate that the tree formula θ holds at execution tree t of the structure S. We write $S, t, \lambda \models \psi$ to indicate that the path formula ψ holds at infinite path λ of the execution tree t of the structure S (note that in this case, λ is a computation of S). If t is an execution tree of S, and λ is a node of t, we write $t(\lambda)$ for the subtree of t with root λ. The satisfaction relation $\models$ is defined inductively as follows:

- For formulas generated by the rules (S1–2), the definition is the same as for ATL. For formulas generated by the rules (T2) and (P2), the definition is obvious.
- $q \models \exists A.\theta$ iff there exists a set F_A of strategies, one for each agent in A, so that $exec(q, F_A) \models \theta$.
- $t \models \varphi$ for a state formula φ iff $q \models \varphi$, where q is the root of the execution tree t.
- $t \models \exists\psi$ for a path formula ψ iff there exists a rooted infinite path λ in t such that $t, \lambda \models \psi$.
- $t, \lambda \models \theta$ for a tree formula θ iff $t \models \theta$.
- $t, \lambda \models \bigcirc\psi$ iff $t(\lambda[0,1]), \lambda[1,\infty] \models \psi$,
- $t, \lambda \models \psi_1 \mathcal{U} \psi_2$ iff there exists a position $i \geq 0$ such that $t(\lambda[0,i]), \lambda[i,\infty] \models \psi_2$ and for all positions $0 \leq j < i$, we have $t(\lambda[0,j]), \lambda[j,\infty] \models \psi_1$.

GL expressiveness The logic ATL* is the syntactic fragment of GL that consists of all formulas in which every strategy quantifier is immediately followed by a path quantifier (note that $\exists A.\exists$ is equivalent to $\exists$). Since the formula $\exists A.(\exists\Box p \wedge \exists\Box q)$ is not equivalent to any ATL* formula, GL is more expressive than ATL*.

Another syntactic fragment of GL is studied in *module checking* [KV96]. There, one considers formulas of the form $\exists A.\theta$, with a single outermost strategy quantifier followed by a CTL or CTL* formula θ. Since the GL formula $\langle\langle A_1\rangle\rangle\Diamond\langle\langle A_2\rangle\rangle\Diamond p$ is not equivalent to any formula with a single outermost strategy quantifier, GL is more expressive than module checking. Furthermore, from an expressiveness viewpoint, alternating-time logics and module checking identify incomparable fragments of game logic. In [KV96], it is shown that the module-checking complexity is EXPTIME-complete for CTL and 2EXPTIME-complete

for CTL*, and the structure complexity of both problems is PTIME-complete. Hence, from a computational viewpoint, ATL is advantageous.

GL model checking The model-checking problem for CTL* can be solved by repeatedly applying, in a bottom-up fashion, an LTL model-checking procedure on subformulas [EL85]. The same technique can be used in order to solve the model-checking problem for GL by repeatedly applying the CTL* module-checking algorithm from [KV96]. The complexity of CTL* module checking then implies the following.

Theorem 7. *The model-checking problem for GL is 2EXPTIME-complete. The structure complexity of the problem is PTIME-complete.*

Thus, game logic is not more expensive than ATL*. We feel, however, that unlike state and path formulas, tree formulas are not natural specifications of reactive systems.

7 Incomplete Information

According to our definition of ATL, every agent has complete information about the state of an ATS. In certain modeling situations it may appropriate, however, to assume that an agent can observe only a subset of the propositions. Then, the strategy of the agent can depend only on the observable part of the history. In this section we study such agents with incomplete information. Using known results on multi-player games with incomplete information, we show that this setting is much more complex than the setting with complete information. Our main result is negative: we show that the ATL model-checking problem is undecidable for cooperating agents with incomplete information. We state this result for our weakest version of ATS, namely, turn-based synchronous ATS.

7.1 ATS with Incomplete Information

A *turn-based synchronous ATS with incomplete information* is a pair $\langle S, P\rangle$ consisting of a turn-based synchronous ATS $S = \langle \Pi, \Sigma, Q, \pi, \sigma, R\rangle$ and a vector $P = \{\Pi_a \mid a \in \Sigma\}$ that contains sets $\Pi_a \subseteq \Pi$ of propositions, one for each agent in Σ. The *observability vector* P defines for each agent a the set Π_a of propositions observable by a. Consider an agent $a \in \Sigma$. For a state $q \in Q$, we term $\pi(q) \cap \Pi_a$ the *a-view* of q. We write $Q_a = 2^{\Pi_a}$ for the set of possible a-views, and $\pi_a : Q \to Q_a$ for the function that maps each state to its a-view. The function π_a is extended to computations of S in the natural way: if $\lambda = q_0, q_1, q_2, \ldots$, then $\pi_a(\lambda) = \pi_a(q_0), \pi_a(q_1), \pi_a(q_2), \ldots$. Two states q and q' are *a-stable* if $\pi(q) \setminus \pi_a(q) = \pi(q') \setminus \pi_a(q')$; that is, q and q' agree on all propositions that a cannot observe. We require that the transition function of a can influence only propositions that a can observe and is independent of propositions that a cannot observe. Formally, we require that the following two conditions hold for all agents $a \in \Sigma$ and all states $q_1, q_1', q_2 \in Q$:

1. If $\sigma(q_1) = a$ and $R(q_1, q_1')$, then q_1 and q_1' are a-stable.
2. If $\sigma(q_1) = \sigma(q_2) = a$ and $\pi_a(q_1) = \pi_a(q_2)$ and $R(q_1, q_1')$, then for every state q_2' such that $\pi_a(q_1') = \pi_a(q_2')$ and q_2 and q_2' are a-stable, we have $R(q_2, q_2')$.

In other words, the transition function of agent a maps each a-view of a state in which a is scheduled into a set of a-views of possible successor states. Accordingly, we define the relation $R_a \subseteq Q_a \times Q_a$ such that $R_a(v, v')$ iff for any and all a-stable states q and q' with $\sigma(q) = a$ and $\pi_a(q) = v$ and $\pi_a(q') = v'$, we have $R(q_1, q_2)$.

7.2 ATL with Incomplete Information

When we specify properties of an ATS with incomplete information using ATL formulas, we restrict ourselves to a syntactic fragment of ATL. To see why, consider the ATL formula $\langle\langle a \rangle\rangle \Diamond p$ for $p \notin \Pi_a$. The formula requires agent a to have a strategy to eventually reach a state in which the proposition p, which a cannot observe, is true. Such a requirement does not make sense. Consequently, whenever a set of agents is supposed to attain a certain task, we require that each agent in the set can observe the propositions that are involved in the task (this includes all propositions that appear in the task as well as all propositions that are observable by agents appearing in the task). Formally, given an observability vector P, we define for each ATL formula φ the set $inv_P(\varphi) \subseteq \Pi$ of *involved propositions*. The definition proceeds by induction on the structure of the formula:

- For $p \in \Pi$, we have $inv_P(p) = \{p\}$.
- $inv_P(\neg\varphi) = arg(\varphi)$.
- $inv_P(\varphi_1 \vee \varphi_2) = inv_P(\varphi_1) \cup inv_P(\varphi_2)$.
- $inv_P(\langle\langle A \rangle\rangle \bigcirc \varphi) = inv_P(\varphi) \cup \bigcup_{a \in A} \Pi_a$.
- $inv_P(\langle\langle A \rangle\rangle \Box \varphi) = inv_P(\varphi) \cup \bigcup_{a \in A} \Pi_a$.
- $inv_P(\langle\langle A \rangle\rangle \varphi_1 \mathcal{U} \varphi_2) = inv_P(\varphi_2) \cup inv_P(\varphi_2) \cup \bigcup_{a \in A} \Pi_a$.

The ATL formula φ is *well-formed* with respect to the observability vector P if the following two conditions hold:

1. For every subformula of φ of the form $\langle\langle A \rangle\rangle \bigcirc \theta$ or $\langle\langle A \rangle\rangle \Box \theta$ and for every agent $a \in A$, we have $inv_P(\theta) \subseteq \Pi_a$.
2. For every subformula of φ of the form $\langle\langle A \rangle\rangle \theta_1 \mathcal{U} \theta_2$ and for every agent $a \in A$, we have $inv(\theta_1) \cup inv(\theta_2) \subseteq \Pi_a$.

Note that if the formula $\langle\langle A \rangle\rangle \psi$ is well-formed, then each agent in A can observe all propositions that are observable by agents appearing in ψ, but it may not be able to observe some propositions that are observable by other agents in A.

When we interpret an ATL formula φ over a turn-based synchronous ATS $\langle S, P \rangle$ with incomplete information, we require φ to be well-formed with respect to P. The definition of the satisfaction relation is as in the case of complete information (see Section 3.2), except for the following definitions of strategies

and outcomes. Now, a *strategy* for an agent $a \in \Sigma$ is a mapping $f_a : Q_a^+ \to Q_a$ such that for all $\chi \in (2^{\Pi_a})^*$ and $v, v' \in \Pi_a$, we have $R_a(v, v')$. Thus, the strategy f_a maps each a-view of a finite computation prefix to the a-view of a possible successor state. Given a state $q \in Q$, a set $A \subseteq \Sigma$ of agents, and a set $F_A = \{f_a \mid a \in A\}$ of strategies, one for each agent in A, a computation $\lambda = q_0, q_1, q_2, \ldots$ is in an *outcome* in $out(q, F_A)$ if $q_0 = q$ and for all positions $i \geq 0$, if $\sigma(q_i) \in A$, then $\pi_a(q_{i+1}) = f_a(\pi_a(\lambda[0, j]))$ for $a = \sigma(q_i)$. Thus, for example, $q \models \langle\langle A \rangle\rangle \bigcirc \varphi$ iff either $\sigma(q) \in A$ and there exists a $\sigma(q)$-view $v \subseteq \Pi_{\sigma(q)}$ such that for all states q' with $R(q, q')$ and $\pi_{\sigma(q)}(q') = v$, we have $q' \models \varphi$, or $\sigma(q) \notin A$ and for all states q' with $R(q, q')$, we have $q' \models \varphi$.

Theorem 8. *The model-checking problem for ATL with incomplete information is undecidable, even in the special case of turn-based synchronous ATS.*

Proof: The outcome problem for multi-player games with incomplete information has been proved undecidable by [Yan97]. This problem is identical to the model-checking problem for the ATL formula $\langle\langle A \rangle\rangle \Diamond p$ on a turn-based synchronous ATS with incomplete information. □

We note that for Fair-ATL, proving undecidability is easier, and follows from undecidability results on asynchronous multi-player games with incomplete information [PR79, PR90].

7.3 Single-agent ATL with Incomplete Information

Single-agent ATL is the fragment of ATL in which every path quantifier is parameterized by a singleton set of agents. In this case, where agents cannot cooperate, the model-checking problem is decidable also for incomplete information. There is an exponential price to be paid, however, over the setting with complete information.

Theorem 9. *The model-checking problem for single-agent ATL with incomplete information is EXPTIME-complete. The structure complexity of the problem is also EXPTIME-complete, even in the special case of turn-based synchronous ATS.*

Proof: We start with the upper bound. Given a turn-based synchronous ATS $\langle S, P \rangle$ and an ATL formula φ, well formed with respect to P, we label the states of S by subformulas of φ, starting as usual from the innermost subformulas. Since φ is well-formed with respect to P, for each subformula of the form $\langle\langle a \rangle\rangle \psi$, the agent a can observe all labels that correspond to subformulas of $\langle\langle a \rangle\rangle \psi$, and we refer to these labels as observable propositions. For subformulas generated by the rules (S1–2), the labeling procedure is straightforward. For subformulas generated by (S3), we proceed as follows. Given a state q of S, and a well-formed ATL formula φ' of the form $\langle\langle a \rangle\rangle \bigcirc p$, $\langle\langle a \rangle\rangle \Box p$, or $\langle\langle a \rangle\rangle p_1 \mathcal{U} p_2$, for an agent a and observable propositions p, p_1, p_2, we define a turn-based synchronous ATS S'

(with complete information) and a state q' of S' such that $\langle S, P\rangle, q \models \varphi'$ iff $S', q' \models \varphi'$. Let $S = \langle \Pi, \Sigma, Q, \pi, \sigma, R\rangle$, and let $\Pi_{\varphi'}$ be the set of observable propositions in φ' (then $\Pi_{\varphi'} \subseteq \Pi_a$). In order to define S', we need the following notations. First, we add to Π_a a special proposition p_a that indicates if agent a is scheduled to proceed; that is, for all states $q \in Q$, we have $q \models p_a$ iff $\sigma(q) = a$. Let $\Pi'_a = \Pi_a \cup \{p_a\}$. For a set $Q_1 \subseteq Q$, an agent $a \in \Sigma$, and an extended a-view $\upsilon \subseteq \Pi'_a$, we define the *$\upsilon$-successor* of Q_1 as the set

$$Q_2 = \{q_2 \in Q \mid \pi_a(q_2) = \upsilon \text{ and there exists a state } q_1 \in Q \text{ such that } R(q_1, q_2)\};$$

that is, Q_2 is the set of all states with extended a-view υ that are successors of some state in Q_1. Now, $S' = \langle \Pi_{\varphi'}, \{a, b\}, Q', \pi', \sigma', R'\rangle$ is defined as follows:

- $Q' \subseteq 2^Q$ is the smallest set satisfying (1) $\{q\} \in Q'$ and (2) for all sets $Q_1 \in Q'$ and all a-views $\upsilon \subseteq \Pi'_a$, the υ-successor of Q_1 is in Q'. Note that for all sets $Q_1 \in Q'$, if $q_1, q_2 \in Q_2$, then q_1 and q_2 have the same a-view, and either agent a is scheduled to proceed in both q_1 and q_2, or a is not scheduled to proceed in both q_1 and q_2. Hence, each state in Q' corresponds to a set of states in Q that are indistinguishable by the agent a.
- For all sets $Q_1 \in Q'$, if $\pi_a(q) = \upsilon$ for any and all $q \in Q_1$, then $\pi'(Q_1) = \upsilon$.
- For all sets $Q_1 \in Q'$, if $\sigma(q) = a$ for any and all $q \in Q_1$, then $\sigma'(Q_1) = a$; otherwise, $\sigma'(Q_1) = b$.
- For all sets $Q_1, Q_2 \in Q'$, we have $R'(Q_1, Q_2)$ iff $R_a(\pi'(Q_1), \pi'(Q_2))$.

It is easy to prove that for each of the three types of φ' we have $\langle S, P\rangle, q \models \varphi'$ iff $S', \{q\} \models \varphi'$. Since the size of S' is exponential in the size of S, membership in EXPTIME follows from Theorem 2.

For the lower bound, we observe that the model-checking problem for the ATL formula $\langle\langle a\rangle\rangle \Diamond p$ on a turn-based synchronous ATS with the two agents a and b and incomplete information is identical to the outcome problem for two-player games with incomplete information. The latter problem is known to be EXPTIME-hard [Rei84]. □

8 Conclusions

Methods for reasoning about closed systems are, in general, not applicable for reasoning about open systems. The verification problem for open systems, more than it corresponds to the model-checking problem for temporal logics, corresponds, in the case of linear time, to the *realizability* problem [ALW89, PR89a, PR89b], and in the case of branching time, to the *module-checking* problem [KV96]; that is, to a search for winning strategies. Indeed, existing methods for the verification of open systems could not circumvent the computational price caused by solving infinite games. The logic ATL introduced here identifies a class of verification problems for open systems for which it suffices to solve iterated finite games. The ensuing linear model-checking complexity for ATL shows that despite the pessimistic results achieved in this area area so far, there is still a

great deal of interesting reasoning about open systems that can be performed naturally and efficiently.

While closed systems are naturally modeled as labeled transition systems (Kripke structures), we model open systems as alternating transition systems. In the case of closed systems, ATL degenerates to CTL, Fair-ATL to Fair-CTL [CES86], and ATL* to CTL*. Our model-checking complexity results are summarized in Table 1. All complexities in the table denote tight bounds, where m is the size of the system and ℓ is the length of the formula.

	Closed System	Open System
ATL joint complexity	PTIME [CES86]	PTIME $O(m\ell)$
ATL structure complexity	NLOGSPACE [BVW94]	PTIME
Fair-ATL joint complexity	PTIME [CES86]	PTIME $O(m^2\ell^2)$
Fair-ATL structure complexity	NLOGSPACE [KV95]	PTIME
ATL* joint complexity	PSPACE [CES86]	2EXPTIME $m^{2^{O(\ell)}}$
ATL* sstructure complexity	NLOGSPACE [BVW94]	PTIME

Table 1. Model-checking complexity results

Acknowledgments. We thank Amir Pnueli, Moshe Vardi, and Mihalis Yannakakis for helpful discussions. We also thank Luca de Alfaro and Freddy Mang for their comments on a draft of this manuscript.

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 207–218, 1996.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 1–17, 1989.

[BBG+94] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In *Proc. 6th Conference on Computer-aided Verification*, volume 818 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 182–193, 1994.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Symposium on Logic in Computer Science*, pages 428–439, 1990.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proc. 6th Conference on Computer-aided Verification*, volume 818 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 142–155, 1994.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 52–71, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[Cle93] R. Cleaveland. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[Dil89] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 368–377, 1988.

[EL85] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[EL86] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *Proc. 1st Symposium on Logic in Computer Science*, pages 267–278, 1986.

[ES84] E.A. Emerson and A.P. Sistla. Deciding branching-time logic. In *Proc. 16th ACM Symposium on Theory of Computing*, 1984.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[GSSL94] R. Gawlick, R. Segala, J. Sogaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. In *Proc. 23rd Int. Colloquium on Automata, Languages, and Programming*, volume 820 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 166–177, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[Imm81] N. Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22(3):384–406, 1981.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KV95] O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conferance on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 408–422, 1995.

[KV96] O. Kupferman and M.Y. Vardi. Module checking. In *Proc. 8th Conference on Computer-aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 75–86, 1996.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Par83] R. Parikh. Propositional game logic. In *Proc. 24th IEEE Symposium on Foundation of Computer Science*, pages 195–200, 1983.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[PR79] G.L. Peterson and J.H. Reif. Multiple-person alternation. In *Proc. 20th IEEE Symposium on Foundation of Computer Science*, pages 348–363, 1979.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 652–671, 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symposium on Foundation of Computer Science*, pages 746–757, 1990.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 337–351, 1981.

[Rab70] M.O. Rabin. Weakly definable relations and special automata. In *Proc. Symposium on Mathematical Logic and Foundations of Set Theory*, pages 1–23. North Holland, 1970.

[Rei84] J.H. Reif. The complexity of two-player games of incomplete information. *Journal on Computer and System Sciences*, 29:274–301, 1984.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[RW89] P.J.G. Ramadge and W.M. Wonham. The control of descrete event systems. *IEEE Transactions on Control Theory*, 77:81–98, 1989.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Symposium on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, Springer-Verlag, pages 1–13, 1995.

[VW86a] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symposium on Logic in Computer*

Science, pages 322–331, 1986.

[VW86b] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):182–221, 1986.

[Yan97] M. Yannakakis. Synchronous multi-player games with incomplete information are undecidable. Personal communication, 1997.

Compositionality in dataflow synchronous languages: specification & code generation *

Albert Benveniste[1], Paul Le Guernic[1], and Pascal Aubry[2]

[1] Irisa/Inria, Campus de Beaulieu, 35042 Rennes cedex, France; email: name@irisa.fr
[2] Irisa/Ifsic, Campus de Beaulieu, 35042 Rennes cedex, France; email: name@ifsic.fr

Abstract. Modularity is advocated as a solution for the design of large systems, the mathematical translation of this concept is often that of *compositionality*. This paper is devoted the issues of compositionality aiming at modular code generation, for dataflow synchronous languages. As careless storing of object code for further reuse in systems design fails to work, we first concentrate on what are the additional features needed to abstract programs for the purpose of code generation: we show that a central notion is that of *scheduling specification* as resulting from a *causality analysis* of the given program. Then we study separate compilation for synchronous programs, and we discuss the issue of distributed implementation using an asynchronous medium of communication; for both topics we provide a complete formal study in the extended version [BlGA97] of this paper. Corresponding algorithms are currently under development in the framework of the *DC+ common format* for synchronous languages.
Keywords: synchronous languages, modularity, code generation, separate compilation, desynchronisation.

1 Motivations

Modularity is advocated as the ultimate solution for the design of large systems, and this holds in particular for embedded systems, and their software and architecture. Modularity allows the designer to scale down design problems, and facilitate reuse of predefined modules.

The mathematical translation of the concept of modularity is often that of *compositionality*. Paying attention to the composition of specifications [MP92] is central to any system model involving concurrency or parallelism. More recently, significant effort has been devoted toward introducing compositionality in verification with aiming at deriving proofs of large programs from partial proofs involving (abstractions of) components [MP95].

* This work is or has been supported in part by the following projects: Eureka-SYNCHRON, Esprit R&D -SACRES (Esprit project EP 20897), Esprit LTR-SYRF (Esprit project EP 22703). In addition to the listed authors, the following people have indirectly, but strongly, contributed to this work: the STS formalism has been shamelessly borrowed from Amir Pnueli, and the background on labelled partial orders is mostly ackowledged to Paul Caspi.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 61-80, 1998.
Springer-Verlag Berlin Heidelberg 1998

Compilation and code generation has been given less attention from the same point of view, however. This is unfortunate, as it is critical for the designer to scale down the design of large systems by 1/ storing modules like black-box "procedures" or "processes" with minimal interface description, and 2/ generating code only using these interface descriptions, while still guaranteing that final encapsulation of actual code within these black-boxes together with their composition, will maintain correctness of the design w.r.t. specification.

This paper is devoted the issues of compositionality aiming at modular code generation, for dataflow synchronous languages. As dataflow synchronous is rather a paradigm more than a few concrete languages or visual formalisms [BB91], it was desirable to abstract from such and such particular language. Thus we have chosen to work with a formalism proposed by Amir Pnueli, that of Symbolic Transition Systems (STS [Pnu97]), which is at the same time very lightweight, and fully general to capture the essence of synchronous paradigm.

Using this formalism, we study composition of specifications, a very trivial topic indeed. Most of our effort is then devoted to issues of compositionality that are critical to code generation. As careless storing of object code for further reuse in systems design fails to work, we first concentrate on what are the additional features needed to abstract programs for the purpose of code generation: we show that a central notion is that of scheduling specification as resulting from a causality analysis of the given program. Related issues of compositionality are investigated. Then we show that there is some appropriate level of "intermediate code", which at the same time allows us to scale down code generation for large systems, and still maintains correctness at the system integration phase. Finally we discuss the issue of distributed implementation using an asynchronous medium of communication.

Besides this, let us mention that Amir Pnueli & coworkers [Pnu97] have introduced a more elaborated version of our STS formalism, for the purpose of investigating issues of compositionality in proofs involving liveness properties.

This work was initiated within the context of the SIGNAL synchronous language by P. Le Guernic and its students B. Le Goff, O. Maffeis [MLG94], and P. Aubry [Aub97] who finally implemented these ideas.

2 The essentials of the synchronous paradigm

There has been several attempts to characterize the essentials of the synchronous paradigm [BB91] [Halb93]. With some experience and after attempts to address the issue of moving from synchrony to asynchrony (and back), we feel the following features are indeed essential for characterizing this paradigm:

1. Programs progress via an infinite sequence of *reactions*:

$$P = R^{\omega}$$

where R denotes the family of possible reactions[1].

2. Within a reaction, decisions can be taken on the basis of the *absence* of some events, as exemplified by the following typical statements, taken from ESTEREL, LUSTRE, and SIGNAL respectively:

```
present S else 'stat'
y = current x
y := u default v
```

The first statement is selfexplanatory. The "`current`" operator delivers the most recent value of `x` at the clock of the considered node, it thus has to test for absence of `x` before producing `y`. The "`default`" operator delivers its first argument when it is present, and otherwise its second argument.

3. When it is defined, parallel composition is always given by taking the conjunction of associated reactions:

$$P_1 \| P_2 \;=\; (R_1 \wedge R_2)^{\omega}$$

Typically, if specifying is the intention, then the above formula is a perfect definition of parallel composition. In contrast, if programming was the intention, then the need for this definition to be compatible with an operational semantics very much complicates the "when it is defined" prerequisite[2].

Of course, such a characterization of what the synchronous paradigm is makes the class of "synchrony–compliant" formalisms much larger than usually considered. But it has been our experience that these were the key features for the techniques we have developed so far to work.

Clearly, these remarks call for a common format implementing this paradigm, the DC/DC$_+$ format [DC96] has been proposed with this objective. Also, this calls for a simplest possible formalism with the above features, on which fundamental questions should be investigated (the purpose of this basic synchronous formalism would not be to allow better specification or programming, however): the STS formalism we describe next has this in its objectives.

3 Specification: Symbolic Transition Systems (STS) [Pnu97]

Symbolic Transition Systems (STS). We assume a vocabulary $\mathcal{V}$ which is a set of typed variables. All types are implicitly extended with a special element $\perp$ to be interpreted as "absent". Some of the types we consider are the type of *pure signals* with domain $\{\mathrm{T}\}$, and *booleans* with domain $\{\mathrm{T}, \mathrm{F}\}$ (recall both types are extended with the distinguished element $\perp$).

[1] In fact, "reaction" is a slightly restrictive term, as we shall see in the sequel that "reacting to the environment" is not the only possible kind of interaction a synchronous system may have with its environment.

[2] For instance, most of the effort related to the semantics of ESTEREL has been directed toward solving this issue satisfactorily [Ber95].

We define a *state* s to be a type-consistent interpretation of $\mathcal{V}$, assigning to each variable $u \in \mathcal{V}$ a value $s[u]$ over its domain. We denote by S the set of all states. For a subset of variables $V \subseteq \mathcal{V}$, we define a V-state to be a type-consistent interpretation of V.

Following [Pnu97][3], we define a *Symbolic Transition System* (STS) to be a system

$$\Phi = \langle V, \Theta, \rho \rangle$$

consisting of the following components:

- V is a finite set of typed *variables,*
- $\Theta(V)$ is an assertion characterizing *initial states.*
- $\rho = \rho(V^-, V)$ is the *transition relation* relating previous and current states s^- and s, by referring to both past[4] and current versions of variables V^- and V. For example the assertion $x = x^- + 1$ states that the value of x in s is greater by 1 than its value in s^-. If $\rho(s^-[V], s[V]) = \mathrm{T}$, we say that state s^- is a *ρ-predecessor* of state s.

A *run* $\sigma : s_0, s_1, s_2, \ldots$ is a sequence of states such that

$$s_0 \models \Theta(s_0) \quad \bigwedge \quad \forall i > 0 \ , \ (s_{i-1}, s_i) \models \rho(s_{i-1}, s_i) \tag{1}$$

The *composition* of two STS $\Phi = \Phi_1 \bigwedge \Phi_2$ is defined as follows:

$$\begin{aligned} V &= V_1 \cup V_2 \\ \Theta &= \Theta_1 \wedge \Theta_2 \\ \rho &= \rho_1 \wedge \rho_2 \ , \end{aligned}$$

the composition is thus the pairwise conjunction of initial and transition relations.

Notations for STS*:* we shall use the following generic notations in the sequel:

- $c, v, w, \ldots$ denote STS *variables.*
- for v a variable, $h_v \in \{\mathrm{T}, \bot\}$ denotes its *clock*:

$$[h_v \neq \bot] \Leftrightarrow [v \neq \bot]$$

- for v a variable, ξ_v denotes its associated *state variable,* defined by:

$$\begin{aligned} &\textbf{if } h_v \textbf{ then } \xi_v = v \\ &\quad \textbf{else } \ \xi_v = \xi_v^- \end{aligned} \tag{2}$$

[3] The talented reader will also notice some close relation to the TLA model of L. Lamport.

[4] Usually, variables and *primed* variables are used to refer to current and *next* states. This is equivalent to our present notation. We have preferred to consider s^- and s, just because the formulæ we shall write mostly involve current variables, rather than past ones. Using the standard notation would have resulted in a burden of primed variables in the formulæ.

with the convention that $s_0[\xi_v^-] = \perp$, i.e., ξ_v is absent before the 1st occurrence of v, and is always present after the 1st occurrence of v. State variable ξ_v is assumed to be *local,* i.e., is never shared among different STS, and thus state variables play no role for STS composition.

Examples of Transition Relations :

- a selector:

$$\textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \ .$$

  Note that the "**else**" part corresponds to the property "$[b = \text{F}] \vee [b = \perp]$".
- decrementing a register:

$$\textbf{if } h_z \textbf{ then } v = \xi_z^- \Leftrightarrow 1 \textbf{ else } v = \perp \ ,$$

  where ξ_z is the state variable associated with z as in (2), and ξ_z^- denotes its previous value. The more intuitive interpretation of this statement is: $(v_n = z_{n-1} \Leftrightarrow 1)$, where index "$n$" denotes the instants at which both v and z are present (their clocks are specified to be equal). The specification of a register would simply be:

$$\textbf{if } h_z \textbf{ then } v = \xi_z^- \textbf{ else } v = \perp \ .$$

  Note that both statements imply the equality of clocks:

$$h_z \equiv h_v \ .$$

- testing for a property:

$$\textbf{if } h_v \textbf{ then } b = (v \leq 0) \textbf{ else } b = \perp \ .$$

  Note that a consequence of this definition is, again,

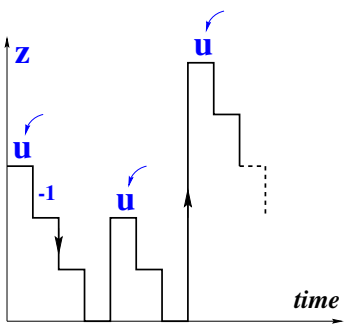
$$h_z \equiv h_v \ .$$

- a synchronization constraint:

$$(b = \text{T}) \equiv (h_u = \text{T}) \, ,$$

  meaning that the clock of u is the set of instants at which boolean variable b is true.

Putting things together yields the STS:



$$\begin{array}{l} \quad \textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \\ \bigwedge \textbf{ if } h_z \textbf{ then } v = \xi_z^- \Leftrightarrow 1 \textbf{ else } v = \perp \\ \bigwedge \textbf{ if } h_v \textbf{ then } b = (v \leq 0) \textbf{ else } b = \perp \\ \bigwedge \ h_v \equiv h_z \equiv h_b \\ \bigwedge \ (b = \text{T}) \equiv (h_u = \text{T}) \end{array}$$

A run of this STS for the z variable is depicted on the figure above. Each time u is received, z is reset and gets the value of u. Then z is decremented by one at each activation cycle of the STS, until it reaches the value 0. Immediately after this latter instant, a fresh u can be read, and so on. Note the schyzophrenic nature of the "inputs" of this STS. While the value carried by u is an input, the instant at which u is read is not : reading of the input is on demand-driven mode. This is reflected by the fact that inputs of this STS are the pair {activation clock h, value of u when it is present}.

Using these primitives, dataflow synchronous languages such as LUSTRE [Halb93] and SIGNAL [LG91] are easily encoded.

Open environments and modularity. As modularity is wanted, it is desirable that the pace of an STS is local to it rather than global. Since any STS is subject to further composition in some yet unknown environment, this makes the requirement of having a global pace quite inconvenient. This is why *we prohibit the use of clocks that are always present.* This has several consequences. First, it is not possible to consider the "complement of a clock" or the "negation of a clock" : this would require referring to the always present clock. Thus, as will be revealed by our examples, clocks will always be variables, and we shall be able to relate clocks only using $\wedge$ (intersection of instants of presence), $\vee$ (union of instants of presence), and $\setminus$ (set difference of instants of presence).

Stuttering. In the same vein, it should be permitted, for an STS, to do nothing while the environment is possibly working. This feature has been yet identified in the litterature and is known as *stuttering* invariance or robustness [Lam83a,Lam83b]. It is central to TLA, where it is understood that a transition with no event at all and no change of states is always legal.

For an STS Φ, stuttering invariance is defined as follows : if

$$\sigma \ : s_0, s_1, s_2, \ldots$$

is a run of Φ, so is

$$\sigma' : s_0, \underbrace{\bot_0, \ldots, \bot_0}_{0 \leq \#\{\bot_0\} < \infty}, s_1, \bot_1, \ldots, \bot_1, s_2, \bot_2, \ldots, \bot_2, \ldots \ , \tag{3}$$

where symbol $\bot_i$ denotes a *silent* state in which all state variables keep the value they had at state s_i, while other variables take the value $\bot$. The number of inserted silent states is ≥ 0 but finite. This models that the considered STS can do nothing for any arbitrary but finite "duration".

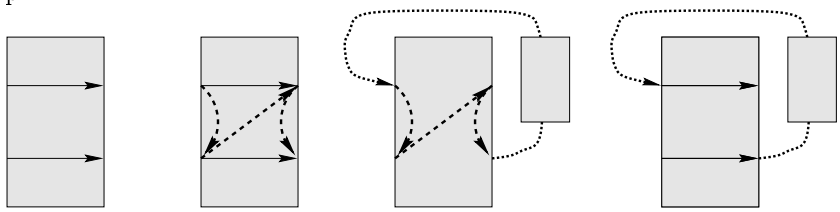
Stuttering invariance is not hardwired into our STS formalism, but a quick inspection of all the statements we have introduced in our example reveals that any composition of them is stuttering invariant. More generally, any STS not involving the always present clock (be it directly or indirectly) is stuttering invariant.

4 Compositional reasoning on causality and scheduling specifications

4.1 What is the problem?

Basically, the problem is twofold: 1/ bruteforce separate compilation can cause deadlocks, and 2/ generating distributed code is generally not compatible with maintaining strict compliance with the synchronous model of computation. We illustrate briefly these two issues next.

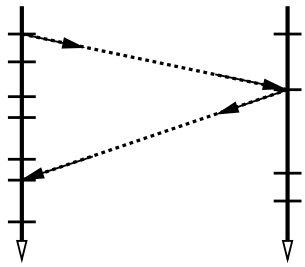
Naive separate compilation may be dangereous. This is illustrated in the following picture:



The first diagram depicts the "dependencies" associated with some STS specification: the 1st output needs the 1st input for its computation, and the 2nd output needs the 2nd input for its computation. The second diagram shows a possible scheduling, corresponding to the standard scheduling: 1/ read inputs, 2/ compute reaction, 3/ emit outputs. This gives a correct sequential execution of the STS. In the third diagram, an additional dependency is enforced by setting the considered STS in some environment which reacts with no delay to its inputs: a deadlock is created. In the last diagram, however, it is revealed that this additional dependency caused by the environment indeed was compatible with the original specification, and no deadlock resulted from applying it. Here, deadlock was caused by the actual implementation of the specification, not by the specification itself.

The traditional answer to this problem by the synchronous programming school has been to refuse considering separate compilation: modules for further reuse should be stored as source code, and combined as such before code generation. We shall however later see that this does not need to be the case, however.

Desynchronisation. This is illustrated in the following picture:



This figure depicts a communication scenario: two processors, modelled as sequential machines, exchange messages using an asynchronous medium for their communications. The natural structure of time is that of a *partial order,* as derived from the directed graph composed of 1/ linear time on each processor, and 2/ communications. This structure for time does not match the linear time corresponding to the infinite sequence of reactions which is the very basis of synchronous paradigm.

The need for reasoning about causality, schedulings, and communications. This need emerges from the above discussion. In the next subsection, we shall introduce a unique framework to handle these diverse aspects: the formalism of *scheduling specifications.*

4.2 Scheduling specifications

Preorders and partial orders to model causality relations, schedulings, and communications. Causality relations have been investigated for several years in the past in the area of models of distributed systems and computations. The classical approach considers a classical automaton, in which concurrency is modelled via an "independence" equivalence relation among the labels of the transitions. Since independence is generally not a symmetric relation (actions of writing and reading are not symmetric), the theory of traces [AR88] has been extended to so-called "semi-commutations" [CL87], and this technique has been recently applied to the implementation of reactive automata on distributed architectures [CCGJ97]. Causality preorder relations have also been used in a different way in [BCHlG94], from which we borrow the essentials of the present technique. In addition to modelling causality relations, preorders can be used to specify scheduling requirements, they can also be used to model send/receive type of communications.

We consider a set V of variables. A *preorder* on the set V is a relation (generically denoted by $\preceq$) which is reflexive ($x \preceq x$) and transitive ($x \preceq y$ and $y \preceq z$ imply $x \preceq z$). To $\preceq$ we associate the equivalence relation $\asymp$, defined by $x \asymp y$ iff $x \preceq y$ and $y \preceq x$. If equivalence classes of $\asymp$ are singletons, then $\preceq$ is a *partial order.*

The *conjunction* of two preorders is the minimal preorder which is an extension of the two considered conjuncts.

STS **with scheduling specifications.** Now we consider STS $\Phi = \langle V, \Theta, \rho \rangle$ as before, but with the following additional feature: as part of

- $\Theta(V)$ (relation defining initial states), and
- $\rho(V^-, V)$ (transition relation),

we have *preorders* denoted by

$$x \preceq y \text{ for } x, y \in V \ ,$$

specified via *(possibly cyclic)* directed graphs:

$$x \to y \ \text{ for } \ x, y \in V \ .$$

STS involving such type of preorder relation shall be called in the sequel STS *with scheduling specifications.* As preorders are just like any other relation, STS with scheduling specifications are just like any other STS, hence they inherit their properties, in particular *they can be composed.*

Notations for scheduling specifications : for b a variable of type $bool \cup \{\bot\}$, and u, v variables of any type,

$$\textbf{if } b \textbf{ then } u \longrightarrow v \ \text{ , resp. } \textbf{if } b \textbf{ else } u \longrightarrow v$$

is denoted by

$$u \xrightarrow{\ b\ } v \text{ resp. } u \xrightarrow{\ \overline{b}\ } v$$

Note the following:

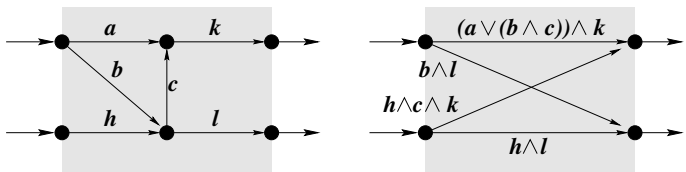
$$\underbrace{u \xrightarrow{\ \neg b\ } v}_{b=\text{false}} \neq \underbrace{u \xrightarrow{\ \overline{b}\ } v}_{b=\text{false} \ \vee \ b=\bot} \ \text{!!!}$$

In the extended version [BlGA97], Appendix A, it is shown that scheduling specifications have the following properties:

$$x \xrightarrow{\ b\ } y \bigwedge y \xrightarrow{\ c\ } z \Rightarrow x \xrightarrow{\ b \wedge c\ } z \tag{4}$$

$$x \xrightarrow{\ b\ } y \bigwedge x \xrightarrow{\ c\ } y \Rightarrow x \xrightarrow{\ b \vee c\ } y \tag{5}$$

Properties (4,5) can be used to compute input/output abstractions of scheduling specifications:



In this figure, the diagram on the left depicts a scheduling specification involving local variables. These are hidden in the diagram on the right, using rules (4,5).

Inferring scheduling specifications from causality analysis. The idea supporting causality analysis of an STS specification is quite simple. On the one hand, a transition relation involving only the types "pure" and "boolean" can be solved by unification and thus made executable. On the other hand, a transition relation involving arbitrary types is abstracted as term rewriting, encoded via directed graphs. For instance, relation $y = 2uv^2$ (involving, say, real types) is abstracted as $(u, v) \longrightarrow y$, since y can be substituted by expression $2uv^2$. On the other hand, the relation $w + uv^2 \geq 0$ (again involving real types) is abstracted as the full directed graph with vertices (u, v, w), as no term rewriting is possible. A difficulty arises from the hybrid nature of general STS, in which boolean variables can be computed from the evaluation of non-boolean expressions (e.g., $b = (x \geq 0)$), and then used as a control variable.

We now provide a technique for inferring schedulings from causality analysis for STS specified as conjunctions of the particular set of primitive statements we have introduced so far. In formulæ (6), each primitive statement has a scheduling specification associated with it, given on the corresponding right hand side of the table. Given an STS specified as the conjunction of a set of such statements, for each conjunct we add the corresponding scheduling specification to the considered STS. Since, in turn, scheduling specifications themselves have scheduling specifications associated with them, this mechanism of adding scheduling specifications must be applied until fixpoint is reached. Note that applying these rules until fixpoint is reached takes at most two successive passes. In formulæ (6), labels of schedulings are expressions involving variables in the domain $\{\bot, \mathrm{F}, \mathrm{T}\}$ ordered by $\{\bot < \mathrm{F} < \mathrm{T}\}$; with this in mind, expressions involving the symbols "$\wedge$" (min) and "$\vee$" (max) have a clear meaning.

(R-1) $$\forall u \quad h_u \longrightarrow u$$

(R-2) $$\textbf{if } b \textbf{ then } w = u \textbf{ else } w = v \Rightarrow \begin{cases} b \xrightarrow{h_b \wedge (h_u \vee h_v)} h_w \\ h_u \xrightarrow{b \wedge h_u} h_w \\ h_v \xrightarrow{\overline{b} \wedge h_v} h_w \\ u \xrightarrow{b \wedge h_u} w \\ v \xrightarrow{\overline{b} \wedge h_v} w \end{cases} \tag{6}$$

(R-3) $$u \xrightarrow{b} w \Rightarrow b \longrightarrow h_w$$

(R-4) $$\left.\begin{array}{r} w = f(u_1, \ldots, u_k) \\ h_w = h_{u_1} = \ldots = h_{u_k} \end{array}\right\} \Rightarrow u_i \xrightarrow{h_w} w$$

Rules (R-1,...,R-4), as well as the theorem to follow, are formally justified in [BlGA97], Appendix A, where a precise definition of "*deterministic*" is also given.

Theorem 1 (executable STS). *For* P *an* STS,

1. *Apply Rules (R-1,...,R-4) until fixpoint is reached: this yields an* STS *we call* **sched(P)**.
2. *A* sufficient *condition for* P *to have a unique deterministic run is:*
   (a) **sched(P)** *is provably circuitfree at each instant, meaning that* it is never true that

$$x_1 \xrightarrow{b_1} x_2 \xrightarrow{b_2} x_1$$

and

$$(b_1 \wedge b_2 = \mathrm{T})$$

   unless $x_1 = x_2$ *holds.*
   (b) **sched(P)** *has provably no multiple definition of variables at any instant, meaning that, whenever*

$$\begin{aligned} & \textbf{if } b_1 \textbf{ then } x = \exp_1 \\ \wedge\ & \textbf{if } b_2 \textbf{ then } x = \exp_2 \end{aligned}$$

   holds in P *and the* $\exp_1$ *and* $\exp_2$ *are different expressions, then*

$$b_1 \wedge b_2 = \mathrm{T}$$

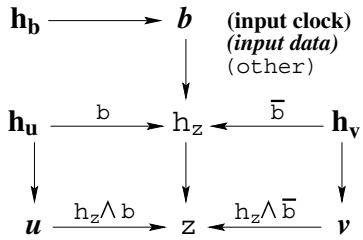
   never holds in P.

Then P *is said to be* executable, *and* **sched(P)** *provides (dynamic) scheduling specifications for this run.*

Examples. We show here some STS statements and their associated scheduling as derived from causality analysis. In the following figures, vertices in boldface denote input clocks, vertices in bold-italic denote input data, and vertices in courier denote other variables. It is of interest to split between these two different types of inputs, as input reading for an STS can occur with any combination of data- and demand-driven mode. Note that, for each vertex of the graph, the labels sitting on the incoming branches are evaluated prior to the considered vertex. Thus, when this vertex is to be evaluated, it is already known which other variables are needed for its evaluation. See the appendix for a formal support of this claim.

A data-driven statement:

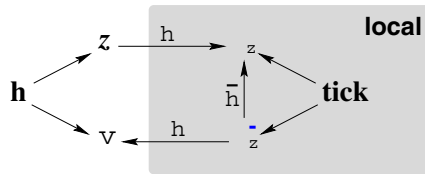
if b **then** $z = u$ **else** $z = v$



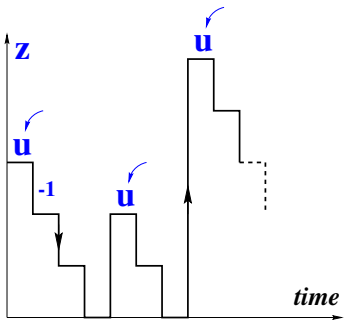
In the above example, input data are pairwise associated with corresponding input clocks: this STS reads its inputs on a purely data-driven mode, input patterns (u, v, b) are free to be present or absent, and, when they are present, their value is free also. We call it a "reactive" STS.

Decrementing a register:

if h_z **then** $v = \xi_z^- \Leftrightarrow 1$ **else** $v = \perp$

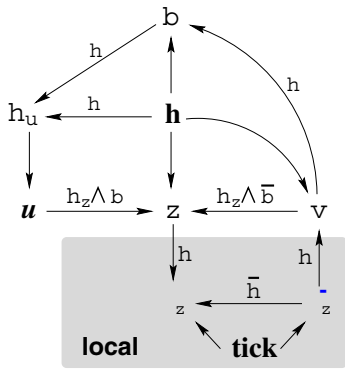


The full example:



if b **then** $z = u$ **else** $z = v$
$\bigwedge$ **if** h_z **then** $v = \xi_z^- \Leftrightarrow 1$ **else** $v = \perp$
$\bigwedge$ **if** h_v **then** $b = (v \leq 0)$ **else** $b = \perp$
$\bigwedge$ $h_v \equiv h_z \equiv h_b$
$\bigwedge$ $(b = \mathrm{T}) \equiv (h_u = \mathrm{T})$

Applying the rules (R-1,...,R-4) for inferring schedulings from causality, we get:



$$
\begin{array}{l}
\quad \textbf{if } b \textbf{ then } z = u \textbf{ else } z = v \\
\bigwedge \textbf{ if } h_z \textbf{ then } v = \xi_z^- \Leftrightarrow 1 \textbf{ else } v = \bot \\
\bigwedge \textbf{ if } h_v \textbf{ then } b = (v \leq 0) \textbf{ else } b = \bot \\
\bigwedge h_v \equiv h_z \equiv h_b \equiv_{\text{def}} h \\
\bigwedge (b = \text{T}) \equiv (h_u = \text{T})
\end{array}
$$

Note the change in control: { input clock, input data} habe been drastically modified from the " **if** b **then** $z = u$ **else** $z = v$" statement to the complete STS: inputs now consist of the pair $\{h, v_u\}$, where v_u refers to the value carried by u when present. Reading of the u occurs on demand, when condition b is true. Thus we call such an STS "proactive".

Summary. What do we get at this stage?

1. STS composition is just the conjunction of constraints.
2. Since preorders are just relations, scheduling specifications do compose as well.
3. As causality analysis is based on an abstraction, the rules (R-1,...,R-4) for inferring scheduling from causality are bound to the *syntax* of the STS conjuncts.
4. Hence, in order to maximize the chance of effectively recognizing that an STS P is executable, P is generally rewritten in a different but semantically equivalent syntax (runs remain the same) while causality analysis is performed[5]. But this latter operation is global and not compositional: here we reach the limits of bruteforce compositionality.

5 Modular and distributed code generation

Two major issues need to be considered:

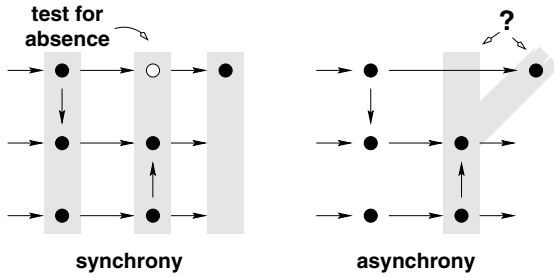
1. *Relaxing synchrony* is needed if distribution over possibly asynchronous media is desired without paying the price for maintaining the strong synchrony hypothesis via costly protocols.
2. *Designing modules equipped with proper interfaces for subsequent reuse,* and generating a correct scheduling and communication protocol for these modules, is the key to modularity.

We consider these two issues next.

[5] This is part of the job performed by the SIGNAL compiler's "clock calculus".

5.1 Relaxing synchrony

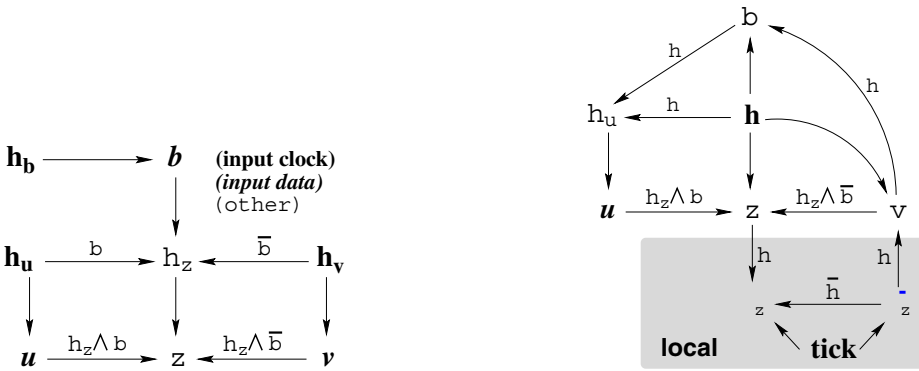
The major problem is that of testing for absence in an asynchronous environment! This is illustrated in the following picture in which the information of "which are the variables present in the considered instant" is lost when passing from left to right hand side, since explicit definition of the "instant" is not available any more:



The questionmark indicates that it is generally not possible, in an asynchronous environment, to decide upon presence/absence of a signal relatively to another one. The major problem is that of "testing for absence" as being part of the control. While this is perfectly sound in a synchronous paradigm, this is meaningless in an asynchronous one.

The solution consists in restricting ourselves to so-called *endochronous* STS. Endochronous STS are those for which the control depends only on 1/ the previous state, and 2/ the values possibily carried by environment signals, but not on the presence/absence status of these signals. An endochronous STS can work as a (synchronous) module working in a distributed execution using an asynchronous communication medium, provided that this medium satisfies the two requirements of 1/ not loosing messages, and 2/ not changing the order of messages.

An example of an STS which is "exochronous" (i.e., not endochronous) is the "reactive" STS given on the left hand side of the following picture, whereas the "proactive" STS shown on the right hand side is an endochronous STS:

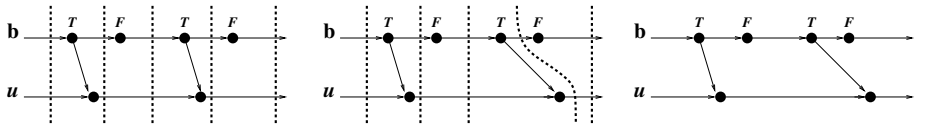


In the diagram on the left hand side, three different clocks are source nodes of the directed graph. This means that the primary decision in executing a re-

action consists in deciding upon relative presence/absence of these clocks. In contrast, in the diagram on the right hand side, only one clock, the activation clock h, is a source node of the graph. Hence no test for relative presence/absence is needed, and the control only depends on the value of the boolean variable b, which is computed internally.

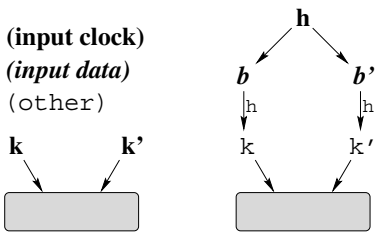
How endochrony allows us to desynchronize an STS is illustrated in an intuitive way on the following diagram, which depicts the scheduling specification associated with the (endochronous) pseudo-statement

" **if** b **then** get_u" :



In the left diagram, a history of this statement is depicted, showing the successive instants (or reactions) separated by thick dashed lines. In the middle, an instant has been twisted, and in the last one, thick dashed lines have been removed. Clearly, no information has been lost : we know that u should be got exactly when $b = \text{T}$, and thus it is only needed to wait for b in order to know whether u is to be waited for also. A formal study of desynchronization and endochrony is presented in [BlGA97], Appendix B.

Moving from exochronous to endochronous is easily performed, we only show one typical but simple example :



The idea is to add to the considered STS a monitor which delivers the information of presence/absence via the b, b' boolean variables with identical clock h, i.e., $\{k = \text{T}\} \equiv \{b = \text{T}\}$, and similarly for k', b'. The resulting STS is endochronous, since boolean variables b, b' are scrutinized at the pace of activation clock h. Other schemes are also possible.

5.2 Generating scheduling for separate modules

Relevant target architectures for embedded applications typically are 1/ purely sequential code (such as C-code), 2/ code using a threading or tasking mechanism provided by some kind of a real-time OS (here the threading mechanism offers some degree of concurrency), or 3/ DSP-type multiprocessor architectures with associated communication media.

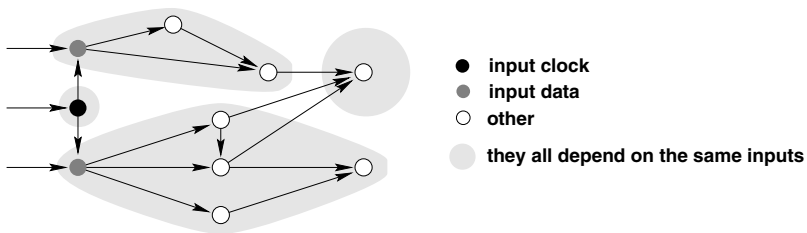
On the other hand, the scheduling specifications we derive from rules (R-1,...,R-4) of causality analysis still exhibit maximum concurrency. Actual imple-

mentations will have to conform to these scheduling specifications. In general, they will exhibit less (and even sometimes no) concurrency, meaning that further sequentialization has been performed to generate code.

Of course, this additional sequentialization can be the source of potential, otherwise unjustified, deadlock when the considered module is reused in the form of object code in some environment, this was illustrated in subsection 4.1. The traditional answer to this problem by the synchronous programming school has been to refuse considering separate compilation: modules for further reuse should be stored as source code, and combined as such before code generation.

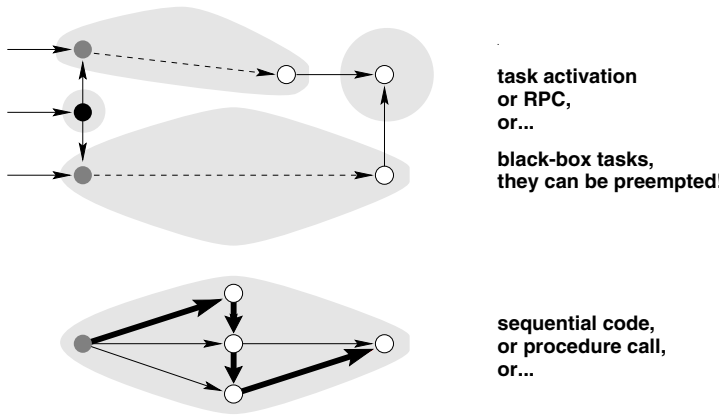
We shall however see that this does not need to be the case, however. Instead, a careful use of the scheduling specifications of an STS will allow us to decompose it into modules that can be stored as object code for further reuse, whatever the actual environment and implementation architecture will be.

The case of single-clocked STS. We first discuss the case of single-clocked STS, in which all variables have the same clock. The issue is illustrated in the following picture, in which the directed graph defining the circuitfree scheduling specification of some single-clocked STS is depicted:



In the above picture, the gray zones group all variables which depend on the same subset of inputs, let us call them "*tasks*". Tasks are not subject to the risk of creating fake deadlocks from implementation. In fact, as all variables belonging to the same task depend on the same inputs, each task can be executed according to the following scheme: 1/ collect inputs, 2/ execute task. The actual way the task is executed is arbitrary, provided it conforms the scheduling specification.

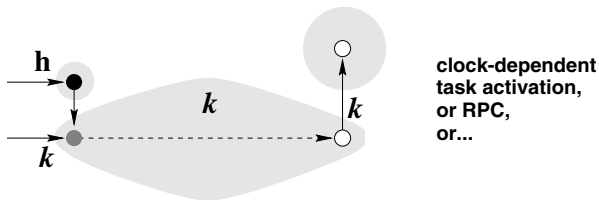
In the next picture, we show how the actual implementation will be prepared:



The thick arrows inside the task depicted on the bottom show one possible fully sequential scheduling of this task. Then, what should be really stored as source code for further reuse is only *the abstraction consisting of the task viewed as black-boxes, together with their associated interface scheduling specifications.*

In particular, if the supporting execution architecture involves a real-time tasking system implementing some preemption mechanism in order to dynamically optimize scheduling for best response time, tasks can be freely suspended/resumed by the real-time kernel, without impairing conformity of the object code to its specification.

The general case of multiple-clocked STS. The generalization is illustrated in the following picture:



The only new point is that all items discussed before are now clock-dependent. Thus the computations of the "tasks", their internal scheduling, and their abstraction, must be performed using scheduling specifications labelled by booleans. In the above example, the considered STS is activated by some clock h, which in turns defines the activation clock k of the depicted task. The bottom line is:

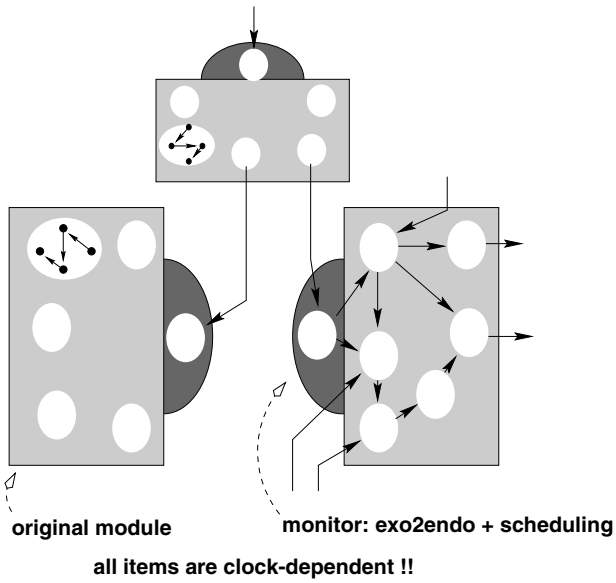
1. Different tasks can have different activation clocks.
2. For each task, the internal scheduling generally depends on some internal clocks, i.e., on some predicates involving internal states of the task. Thus the internal scheduling is dynamic, but can be precomputed at compile time.
3. Also, the task abstraction has a clock-dependent scheduling specification. This is another source of dynamic scheduling, also computed at compile time:

it is implemented in the form of a "software monitor" which opens/forbids the possibility of executing a given task at a given instant, depending on the current status of the STS clocks.

4. This software monitor can be combined with the interruption mechanism of the supporting real-time kernel intended to optimize response time of the execution, without impairing conformity of the object code to its specification.

5.3 The bottom line: modular and distributed code generation

The whole approach is summarized in the following diagram:



In this diagram, gray rectangles denote three modules $\mathtt{P}_1, \mathtt{P}_2, \mathtt{P}_3$ of the source STS specification, hence given by $\mathtt{P} = \mathtt{P}_1 \wedge \mathtt{P}_2 \wedge \mathtt{P}_3$. We assume here that this partitioning has been given by the designer, based on functional and architectural considerations. Note that the idempotence of STS composition ($\mathtt{Q} \wedge \mathtt{Q} = \mathtt{Q}$) allows us to duplicate source code at convenience while partitioning the specification.

Then, white bubbles inside the gray rectangles depict the structuration into tasks as discussed before.

Finally the black half-ellipses denote the monitors. Monitors are in charge of 1/ providing the additional control to make the considered module endochronous if asynchronous communication media are to be used, and 2/ specifying the scheduling of the abstract tasks.

In principle, communication media and real-time kernels do not need to be specified here, as they can be used freely provided they respect the send-receive abstract communication model and conform to the scheduling constraints set by the monitors.

6 Conclusion

The above approach is supported by the DC+ common format for synchronous languages [DC96]. The DC+ format is one possible concrete implementation of our STS model, including scheduling specifications. It serves as a basis for the code generation suite in the Esprit SACRES project. S. Machard and E. Rutten are currently working on generic code generation based on this work, with different real-time O.S. and architectures as targets.

References

[BlGA97] A. BENVENISTE AND P. LE GUERNIC AND P. AUBRY, "Compositionality in dataflow synchronous languages: specification & code generation", extended version of this paper, IRISA Research Report 1150, November 1997. `http://www.irisa.fr/EXTERNE/bibli/pi/1150/1150.html`

[AR88] I.J. AABELSBERG, AND G. ROZENBERG, *Theory of traces,* Theoretical Computer Science, 60, 1–82, 1988.

[Aub97] P. AUBRY, "Mises en œuvre distribuées de programmes synchrones", PhD Thesis, Univ. Rennes I, 1997.

[BB91] A. BENVENISTE, G. BERRY, "Real-Time systems design and programming", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1270–1282.

[BlGJ91] A. BENVENISTE, P. LE GUERNIC, AND C. JACQUEMOT. Synchronous programming with events and relations: the SIGNAL languages and its semantics. *Sci. Comp. Prog.*, 16:103–149, 1991.

[BCHlG94] A. BENVENISTE, P. CASPI, N. HALBWACHS, AND P. LE GUERNIC, Dataflow synchronous languages. In *A Decade of Concurrency, reflexions and perspectives, REX School/Symposium*, pages 1–45, LNCS Vol. 803, Springer Verlag, 1994.

[Ber95] GÉRARD BERRY, *The Constructive Semantics of Esterel,* Draft book, `http://www.inria.fr/meije/esterel`, December 1995.

[CCGJ97] B. CAILLAUD, P. CASPI, A. GIRAUD, AND C. JARD, "Distributing automata for asynchronous networks of processors", *European Journal on Automated Systems (JESA)*, Hermes, 31(3), 503–524, May 1997.

[CL87] M. CLERBOUT, AND M. LATTEUX, *Semi-commutations,* Information and Computation, 73, 59–74, 1987.

[DC96] SACRES CONSORTIUM, The Declarative Code DC+, Version 1.2, May 1996; Esprit project EP 20897: Sacres.

[Halb93] N. HALBWACHS, *Synchronous programming of reactive systems,*. Kluwer Academic Pub., 1993.

[KP96] Y. KESTEN AND A. PNUELI, An αSTS-based common semantics for SIGNAL and STATECHARTS, March 1996. Sacres Manuscript.

[Lam83a] L. LAMPORT, Specifying concurrent program modules, *ACM Trans. on Prog. Lang. and Sys.,* 5(2):190-222, 1983.

[Lam83b] L. LAMPORT, What good is temporal logic? In *Proc. IFIP 9th World Congress,* R.E.A. Mason (Ed.), North Holland, 657-668, 1983.

[LG91] P. LE GUERNIC, T. GAUTIER, M. LE BORGNE, C. LE MAIRE, "Programming real-time applications with SIGNAL", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1321–1336.

[MLG94] O. Maffeis and P. Le Guernic, "Distributed implementation of Signal: scheduling and graph clustering", *in: 3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems,* Lecture Notes in Computer Science 863, Springer Verlag, 149–169, Sept. 1994.

[MP92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[MP95] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Safety.* Springer-Verlag, New York, 1995.

[Pnu97] A. Pnueli, personal communication.

Compositional Reasoning in Model Checking *

Sergey Berezin[1] Sérgio Campos[2] Edmund M. Clarke[1]

[1] Carnegie Mellon University — USA
[2] Universidade Federal de Minas Gerais — Brasil

Abstract. The main problem in model checking that prevents it from being used for verification of large systems is the *state explosion problem*. This problem often arises from combining parallel processes together. Many techniques have been proposed to overcome this difficulty and, thus, increase the size of the systems that model checkers can handle. We describe several *compositional model checking* techniques used in practice and show a few examples demonstrating their performance.

1 Introduction

Symbolic model checking is a very successful method for verifying complex finite-state reactive systems [7]. It models a computer system as a state-transition graph. Efficient algorithms are used to traverse this graph and determine whether various properties are satisfied by the model. By using BDDs [5] it is possible to verify extremely large systems having as many as 10^{120} states. Several systems of industrial complexity have been verified using this technique. These systems include parts of the Futurebus+ standard [12, 19], the PCI local bus [10, 20], a robotics systems [8] and an aircraft controller [9].

In spite of such success, symbolic model checking has its limitations. In some cases the BDD representation can be exponential in the size of system description. This behavior is called the *state explosion problem*. The primary cause of this problem is *parallel composition* of interacting processes. The problem occurs because the number of states in the global model is exponential in the number of component processes. Explicit state verifiers suffer from the state explosion problem more severely than symbolic verifiers. However, the problem afflicts symbolic verification systems as well, preventing them from being applied to larger and more complex examples.

The state explosion can be alleviated using special techniques such as *compositional reasoning*. This method verifies each component of the system in isolation and allows global properties to be inferred about the entire system. Efficient

* This research is sponsored by the the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 81-102, 1998.
Springer-Verlag Berlin Heidelberg 1998

algorithms for compositional verification can extend the applicability of formal verification methods to much larger and more interesting examples. In this paper we describe several approaches to compositional reasoning. Some are automatic and are almost completely transparent to the user. Others require more user intervention but can achieve better results. Each is well suited for some applications while not so efficient for others.

For example, *partitioned transition relations* [6] and *lazy parallel composition* [11, 27] are automatic and, therefore, preferred in cases where user intervention is not desired (for example, when the user is not an expert). These techniques provide a way to compute the set of successors (or predecessors) of a state set without constructing the transition relation of the global system. Both use the transition relations of each component separately during traversal of the state graph. The individual results are combined later to give the set of states in the global graph that corresponds to the result of the operation being performed.

Another automatic technique is based on the use of *interface processes*. This technique attempts to minimize the global state transition graph by focusing on the communication among the component processes. The method considers the set of variables used in the interface between two components and minimizes the system by eliminating events that do not relate to the communication variables. In this way, properties that refer to the interface variables are preserved, but the model becomes smaller.

Assume-guarantee reasoning [17] is a manual technique that verifies each component separately. The behavior of each component depends on the behavior of the rest of the system, i.e., its environment. Because of this, the user must specify properties that the environment has to satisfy in order to guarantee the correctness of the component. These properties are *assumed*. If these assumptions are satisfied, the component will satisfy other properties, called *guarantees*. By combining the set of assume/guarantee properties in an appropriate way, it is possible to demonstrate the correctness of the entire system without constructing the global state graph.

All of these methods have been used to verify realistic systems. This shows that compositional reasoning is an effective method for increasing the applicability of model checking tools. Furthermore, it is a necessity for verification of many complex industrial systems.

The remainder of this paper is organized as follows: Section 2 introduces the formal model that we use for finite-state systems and the kinds of parallel composition we consider. Section 3 describes partitioned transition relations, and Section 4 discusses lazy parallel composition. Interface processes and assume-guarantee reasoning are described in Sections 5 and 6, respectively. Finally, the paper concludes in Section 7 with a summary and some directions for future research.

2 The Model

Given the description of the system to be verified, constructing its model involves two important steps. The first is constructing the model for the individual components. The second is composing these submodels into a global model. We start by showing how to represent each component symbolically given its state-transition graph. Then we describe the parallel composition algorithm used to create the global model.

2.1 Representing a Single Component

Representing a state-transition graph symbolically involves determining its set of states and deriving the transition relation of the graph that models the component. Consider a system with a set of variables V. For a synchronous circuit, the set V is typically the outputs of all the registers in the circuit together with the primary inputs. In the case of an asynchronous circuit, V is usually the set of all nodes. For a protocol or software system, V is the set of variables in the program. A state can be described by giving values to all the variables in V. Since the system is finite-state we can encode all states by boolean vectors. Throughout the paper we assume that this encoding has already been done and that all variables in V are boolean. Therefore, a state can be described by a valuation assigning either 0 or 1 to each variable. Given a valuation, we can also write a boolean expression which is true for exactly that valuation. For example, given $V = \{v_0, v_1, v_2\}$ and the valuation $\langle v_0 \leftarrow 1, v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$, we derive the boolean formula $v_0 \wedge v_1 \wedge \neg v_2$. This boolean formula can then be represented using a BDD.

In general, however, a boolean formula may be true for many valuations. If we adopt the convention that a formula represents the set of *all* valuations that make it true, then we can describe sets of states by boolean formulas and, hence, by BDDs. In practice, BDDs are often much more efficient than representing sets of states explicitly. We denote sets of states with the letter S and we denote the BDD representing the set S by $S(V)$, where V is the set of variables that the BDD may depend on. We also use f, g, ... for arbitrary boolean functions.

In addition to representing sets of states of a system, we must be able to represent the transitions that the system can make. To do this, we extend the idea used above. Instead of just representing a set of states using a BDD, we represent a set of ordered pairs of states. We cannot do this using just a single copy of the state variables, so we create a second set of variables V'. We think of the variables in V as *current state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variable in V', which we denote by v'. A valuation for the variables in V and V' can be viewed as an ordered pair of states, and we represent sets of these valuations using BDDs as above. We write a formula that is true iff there is a transition from the state represented by V to the state represented by V'. For example, if there is a transition from state $\langle v_0 \leftarrow 1, v_1 \leftarrow 1, v_2 \leftarrow 0\rangle$ to state $\langle v_0 \leftarrow 1, v_1 \leftarrow 0, v_2 \leftarrow 1\rangle$ we write the formula $v_0 \wedge v_1 \wedge \neg v_2 \wedge v_0' \wedge \neg v_1' \wedge v_2'$. The disjunction of all such

transitions is the transition relation of the model. If N is a transition relation, then we write $N(V, V')$ to denote the BDD that represents it.

2.2 Parallel Composition

The technique above shows how to construct the graph that models one component of the system. But usually systems are described by a set of components that execute concurrently. For synchronous or asynchronous circuits the components are the smaller circuits that are connected together to construct the bigger circuit. For protocols or programs the components are the processes that execute in parallel.

There are two major ways of composing processes or systems: synchronously and asynchronously. In synchronous composition all processes execute at the same time, one step in one process corresponds to exactly one step in all the other processes. In asynchronous composition, on the other hand, only one process executes at any point in time. When one process steps all the others remain unchanged. The choice of which process steps at any time is nondeterministic. There are different algorithms for composing synchronous and asynchronous systems.

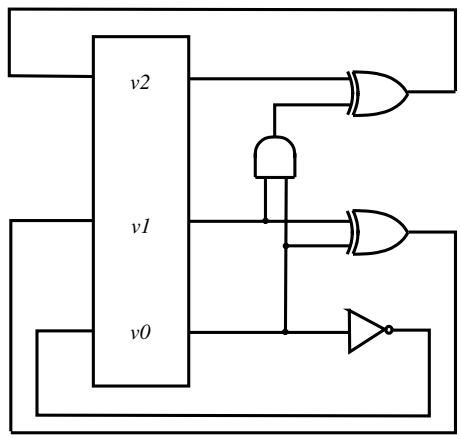


Fig. 1. A modulo 8 counter

Synchronous Systems The method for deriving the transition relation of a synchronous system can be illustrated using a small example. Consider the circuit of a modulo 8 counter on Fig. 1. Let $V = \{v_0, v_1, v_2\}$ be the set of state variables for this circuit, and let $V' = \{v_0', v_1', v_2'\}$ be another copy of the state variables. The transitions of the modulo 8 counter are given by

$$
\begin{aligned}
v_0' &= \neg v_0 \\
v_1' &= v_0 \oplus v_1 \\
v_2' &= (v_0 \wedge v_1) \oplus v_2
\end{aligned}
$$

The above equations can be used to define the relations

$$
\begin{aligned}
N_0(V, V') &= (v_0' \Leftrightarrow \neg v_0) \\
N_1(V, V') &= (v_1' \Leftrightarrow v_0 \oplus v_1) \\
N_2(V, V') &= (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)
\end{aligned}
$$

which describe the constraints each v_i' must satisfy in a legal transition. Each constraint can be seen as a separate component, and their composition generates the counter. These constraints can be combined by taking their conjunction to form the transition relation:

$$N(V, V') = N_0(V, V') \wedge N_1(V, V') \wedge N_2(V, V').$$

In the general case of a synchronous system with n components, we let $\{N_0, \cdots, N_{n-1}\}$ be the set of transition relations for each component. Each transition relation N_i determines the values of a subset of variables in V in the next state. Analogous to the modulo 8 counter, the conjunction of these relations forms the transition relation

$$N(V, V') = N_0(V, V') \wedge \cdots \wedge N_{n-1}(V, V').$$

Thus, the transition relation for a synchronous system can be expressed as a conjunction of relations.

Given a BDD for each transition relation N_i, it is possible to compute the BDD that represents N. We say that such a transition relation is *monolithic* because it is represented by a single BDD. Monolithic transition relations are the primary bottleneck for verification, because their size can be exponential in the number of equations used to define it.

Asynchronous Systems As with synchronous systems, the transition relation for an asynchronous system can be expressed as a conjunction of relations. Alternatively, it can be expressed as a disjunction. To simplify the description of how such transition relations are obtained, we assume that all the components of the system have exactly one output and have no internal state variables. In this case, it is possible to describe completely each component by a function $f_i(V)$. Given values for the present state variables v, the component drives its output to the value specified by $f_i(V)$. For some components, such as C-elements and flip-flops, the function $f_i(V)$ may depend on the current value of the output of the component, as well as the inputs. Extending the method to handle components with multiple outputs is straightforward.

In speed-independent asynchronous systems, there can be an arbitrary delay between when a transition is enabled and when it actually occurs. We can model this by allowing each component to choose nondeterministically whether to transition or not. This results in a conjunction of n parts, all of the form

$$T_i(V, V') = \big(v_i' \Leftrightarrow f_i(V)\big) \vee (v_i' \Leftrightarrow v_i).$$

This model is similar to the synchronous case discussed above, and allows *more* than one variable to transition concurrently.

Normally, we will use an *interleaving model* for asynchronous composition, in which only one variable is allowed to transition at a time. First, we apply the distributive law to the conjunction of the T_i's, giving a disjunction of 2^n terms:

$$\bigwedge_{i=1}^{n} T_i \equiv \bigvee_{b_1,\ldots b_n} \left(\bigwedge_{i=1}^{n} v_i' \Leftrightarrow g_i^{b_i}(V) \right),$$

where all b_i's are indices over $\{0, 1\}$ and

$$g_i^b(V) = \begin{cases} f_i(V), & \text{if } b = 1 \\ v_i, & \text{if } b = 0. \end{cases}$$

Each of these terms $\bigwedge_{i=1}^{n} v_i' \Leftrightarrow g_i^{b_i}(V)$ corresponds to the simultaneous transitioning of some subset of the n variables in the model for which $b_i = 1$. Second, we keep only those terms that correspond to exactly one variable being allowed to transition (that is, only those disjuncts for which the vector $b_1, \ldots, b_n$ contains exactly one 1). This results in a disjunction of the form

$$N(V, V') = N_0(V, V') \vee \cdots \vee N_{n-1}(V, V'),$$

where

$$N_i(V, V') = \big(v_i' \Leftrightarrow f_i(V)\big) \wedge \bigwedge_{j \neq i} (v_j' \Leftrightarrow v_j).$$

Notice, that using this method asynchronous systems are composed by disjuncting their components, while synchronous systems are composed by conjuncting their components.

3 Partitioned Transition Relations

Computing the image or pre-image of a set of states S under a transition relation N is the most important operation in model checking. A state t is a *successor* of s under N, if there is a transition from s to t or, in other words, $N(s, t)$ holds. The *image* of a set of states S is the set of all successors of S. If the set S and the transition relation N are given by boolean formulas, then the image of S is given by the following formula

$$\exists V \big[S(V) \wedge N(V, V')\big],$$

where $\exists V$ denotes existential quantification over all variables in V. This formula defines the set of successors in terms of free variables V'. Similarly, a state s is a *predecessor* of a state t under N iff $N(s,t)$ is true. The set of predecessors of a state set S is described by the formula

$$\exists V' \big[S(V') \wedge N(V,V') \big].$$

Formulas of this type are called *relational products.*

While it is possible to implement the relational product with one conjunction and a series of existential quantifications, in practice this would be fairly slow. In addition, the OBDD for $S(V') \wedge N(V,V')$ is often much larger than the OBDD for the final result, and we would like to avoid constructing it if possible. For these reasons, we use a special algorithm to compute the OBDD for the relational product in one step from the OBDDs for S and N. Figure 2 gives this algorithm for two arbitrary OBDDs f and g.

```
function RelProd(f, g: OBDD, E: set of variables): OBDD

if f = 0 ∨ g = 0
    return 0
else if f = 1 ∧ g = 1
    return 1
else if (f, g, E, r) is in the result cache
    return r
else
    let x be the top variable of f
    let y be the top variable of g
    let z be the topmost of x and y
    r0 := RelProd(f|z←0, g|z←0, E)
    r1 := RelProd(f|z←1, g|z←1, E)
    if z ∈ E
        r := Or(r0, r1)
            /* OBDD for r0 ∨ r1 */
    else
        r := BDDnode(z, r1, r0)
            /* OBDD for (z ∧ r1) ∨ (¬z ∧ r0) */
    endif
    insert (f, g, E, r) in the result cache
    return r
endif
```

Fig. 2. Relational product algorithm

Like many OBDD algorithms, *RelProd* uses a result cache. In this case, entries in the cache are of the form (f,g,E,r), where E is a set of variables that are quantified out and f, g and r are OBDDs. If such an entry is in the cache, it means that a previous call to $RelProd(f,g,E)$ returned r as its result.

Although the above algorithm works well in practice, it has exponential complexity in the worst case. Most of the situations where this complexity is observed are cases in which the OBDD for the result is exponentially larger than the OBDDs for the arguments $f(\bar{v})$ and $g(\bar{v})$. In such situations, any method of computing the relational product must have exponential complexity.

In the previous section we have described how to construct the global transition relation N from the individual transition relations N_i of the component processes. However, the size of N can be significantly larger than the sum of the sizes of all N_is. Our goal is to be able to compute relational products without constructing the global transition relation explicitly.

3.1 Disjunctive Partitioning

The global transition relation of an asynchronous system may be written as the disjunction of the transition relations for the individual components of the system. In this case, a relational product will have the form

$$\exists V' \left[S(V') \wedge (N_0(V,V') \vee \cdots \vee N_{n-1}(V,V')) \right].$$

In practice computing the value of a large formula with many quantifiers is usually very expensive. Since the existential quantifier distributes over disjunction we can shrink the scope of the quantifier to the individual components:

$$\exists V' \left[S(V') \wedge N_0(V,V') \right] \vee \cdots \vee$$

$$\exists V' \left[S(V') \wedge N_{n-1}(V,V') \right]$$

When this technique is used it is possible to compute relational products for much larger asynchronous systems.

3.2 Conjunctive Partitioning

For synchronous systems, a relational product will have the form

$$\exists V' \left[S(V') \wedge (N_0(V,V') \wedge \cdots \wedge N_{n-1}(V,V')) \right].$$

Unfortunately, existential quantification does not distribute over conjunction, so we can not directly apply the same transformation as in the asynchronous case. A simple counterexample is

$$\exists a[(a \vee b) \wedge (\neg a \vee c)] \not\equiv \exists a[a \vee b] \wedge \exists a[\neg a \vee c]$$

since it reduces to:

$$[b \vee c] \not\equiv true.$$

Nevertheless, we still can apply partitioning because systems often exhibit locality: most N_is depend only on a small number of variables in V and V'.

Subformulas can be moved outside of the scope of existential quantification if they do not depend on any of the variables being quantified:

$$\exists a \left[(a \vee b) \wedge (b \vee c) \right] \equiv \exists a \left[a \vee b \right] \wedge (b \vee c)$$

We can optimize the computation of a relational product by using *early variable elimination* for variables in each N_i. First, pick an order ρ for considering the partitions in the relational product. Then define D_i to be the set of variables process P_i depends on, and E_i to be a subset of D_i consisting of variables that *no* process later in the ordering depends on, i.e.,

$$E_{\rho(i)} = D_{\rho(i)} \Leftrightarrow \bigcup_{k=i+1}^{n-1} D_{\rho(k)}.$$

We will illustrate this with our example of the modulo 8 counter.

$$\begin{array}{ll} N_0 = (v_0' \Leftrightarrow \neg v_0) & \text{depends on } D_0 = \{v_0\} \\ N_1 = (v_1' \Leftrightarrow v_0 \oplus v_1) & \text{depends on } D_1 = \{v_0, v_1\} \\ N_2 = (v_2' \Leftrightarrow (v_0 \wedge v_1) \oplus v_2) & \text{depends on } D_2 = \{v_0, v_1, v_2\} \end{array}$$

If we choose the ordering $\rho = 2, 1, 0$, then $E_2 = \{v_2\}, E_1 = \{v_1\}$ and $E_0 = \{v_0\}$. We now can transform the relational product to:

$$\begin{aligned} S_1(V, V') &= \exists_{v \in E_{\rho(0)}} \left[S(V) \wedge N_{\rho(0)}(V, V') \right] \\ S_2(V, V') &= \exists_{v \in E_{\rho(1)}} \left[S_1(V, V') \wedge N_{\rho(1)}(V, V') \right] \\ &\vdots \\ S_n(V') &= \exists_{v \in E_{\rho(n-1)}} \left[S_{n-1}(V, V') \wedge N_{\rho(n-1)}(V, V') \right]. \end{aligned}$$

Or putting it all together,

$$\underbrace{\exists V_{\rho(n-1)} \left[\ldots \underbrace{\exists V_{\rho(1)} \left[\underbrace{\exists V_{\rho(0)} \left[S(V) \wedge N_{\rho(0)}(V, V') \right]}_{S_1} \wedge N_{\rho(1)}(V, V') \right]}_{S_2} \wedge \cdots \wedge N_{\rho(n-1)}(V, V') \right]}_{\vdots \; S_n}$$

The ordering ρ has a significant impact on how early in the computation state variables can be quantified out. This affects the size of the BDDs constructed and the efficiency of the verification procedure. Thus, it is important to choose ρ carefully, just as with the BDD variable ordering. For example, a badly chosen ordering $\rho = 0, 1, 2$ for the same modulo 8 counter yields $E_0 = \{\}, E_1 = \{\}$ and $E_2 = \{v_0, v_1, v_2\}$, which results in no optimization at all.

In practice, we have found it fairly easy to come up with orderings which give good results. We search for a good ordering ρ by using a greedy algorithm

to find a good ordering on the variables v_i to be eliminated. For each ordering on the variables, there is an obvious ordering on the relations N_i such that when this relation ordering is used, the variables can be eliminated in the order given by the greedy algorithm.

The algorithm on fig. 3 gives our basic greedy technique. We start with the set of variables V to be eliminated and a collection $\mathcal{C}$ of sets where every $D_i \in \mathcal{C}$ is the set of variables on which N_i depends. We then eliminate the variables one at a time by always choosing the variable with the least cost and then updating V and $\mathcal{C}$ appropriately.

```
while (V ≠ φ) do
begin
    For each v ∈ V compute the cost of eliminating v;
    Eliminate variable with lowest cost by updating C and V;
end;
```

Fig. 3. Algorithm for variable elimination.

All that remains is to determine the cost metric to use. We will consider three different cost measures. To simplify our discussion, we will use N_v to refer to the relation created when eliminating variable v by taking the conjunction of all the N_i that depend on v and then quantifying out v. We will use D_v to refer to the set of variables on which this N_v depends.

minimum size The cost of eliminating a variable v is simply $|D_v|$. With this cost function, we always try to insure that the new relation we create depends on the fewest number of variables.

minimum increase The cost of eliminating variable v is

$$|D_v| \Leftrightarrow \max_{A \in \mathcal{C}, v \in A} |A| + 1$$

which is the difference between the size of D_v and the size of the largest D_i containing v. The idea is that if we have a lot of small relations that all share one variable, then we do not want to eliminate that variable, since this may result in a big N_v. But this is what the previous heuristic would suggest. Instead, the minimum increase cost will favor eliminating variables that are shared by a small number of relations, thus, keeping the resulting relation smaller. In other words, we prefer to make a small increase in the size of an already large relation than to create a new large relation.

minimum sum The cost of eliminating variable v is

$$\sum_{A \in \mathcal{C}, v \in A} |A|$$

which is simply the sum of the sizes of all the D_i containing v. Since the cost of conjunction depends on the sizes of the arguments, we approximate this cost by the number of variables on which each of the argument N_i depends.

The overall goal is to minimize the size of the largest BDD created during the elimination process. In our abstraction, this translates to finding an ordering that minimizes the size of the largest set D_v created during the process. Always making a locally optimal choice does not guarantee an optimal solution and there are counterexamples for each of the three cost functions. In fact, the problem of finding an optimal ordering can be shown to be NP-complete. However, the minimum sum cost function seems to provide the best approximation of the cost of the actual BDD operations and in practice has the best performance on most examples.

4 Lazy Parallel Composition

Lazy parallel composition is an alternative method for compositional reasoning that can be related to partitioned transition relations. As in the case of the partitioned transition relations, the global transition relation is never constructed. However, in contrast to the previous method, a restricted transition relation for all processes is created. The restricted transition relation agrees with the global transition relation for 'important' states, but it may behave in a different way for other states. The advantage comes from the fact that in many cases it is possible to construct a restricted transition relation that is significantly smaller than the global transition relation.

There are many possible ways of constructing a restricted transition relation that would produce correct results. Given an original global transition relation N and a state set S, the computation of the set of successors of S can use any restricted transition relation N' that satisfies the following condition:

$$N'|_S = N|_S$$

The formula above means that N and N' agree on transitions that start from states in S. It is possible to represent N' with significantly fewer nodes than N in some cases by using the constrain operator from [14, 27]. For two boolean formulas f and g, $f' = constrain(f, g)$ is a formula that has the same truth value as f for variable assignments that satisfy g. If the variable assignment does not satisfy g, the value of f' can be arbitrary. In other words:

$$f'(x) = \begin{cases} f(x) & \text{if } g(x) \\ \text{don't care} & \text{otherwise} \end{cases}$$

In many cases the size of f' is significantly smaller than the size of f.

The lazy composition algorithm uses the constrain operator to simplify the transition relation of each process before generating the global restricted transition relation. When computing the set of successors of a state set S (represented by a boolean formula) the algorithm computes

$$N' = \bigwedge_{i=0..n} constrain(N_i, S).$$

Each transition $N_i' = constrain(N_i, S)$ agrees with N_i on transitions that start in S by the definition of the constrain operator. As a consequence, the transition relation N' agrees with the global transition relation N on transitions that start in S as well. Therefore, computing the set of successors of S using N' produces the same result as using N. The same method can be applied when computing the set of predecessors of a state set S. Only in this case the constrain operator has to maintain those transitions in N that *end* in S.

4.1 Partitioning vs. Lazy Composition

Lazy parallel composition is less sensitive to the order in which variables are eliminated than partitioned transition relations. This is because step i in the partitioned transition relation depends on step $i \Leftrightarrow 1$, as shown below

$$\underbrace{\exists v_1 \big[\underbrace{\exists v_0 \big[S(V') \wedge N_0(V,V')\big]}_{\text{step1}} \wedge N_1(V,V')\big].}_{\text{step2}}$$

As a consequence, the final degree of partitioning heavily depends on the order in which we quantify the variables out. We have already seen an example of such dependency in section 3.2.

The lazy parallel composition, on the other hand, processes each component independently, and thus, does not depend on the order in which the constrain operators are applied:

$$\exists V' \big[S(V') \wedge (\underbrace{N_1(V,V') \;|_S}_{\text{step1}} \wedge \underbrace{N_2(V,V') \;|_S}_{\text{step2}})\big].$$

We have implemented the lazy composition algorithm and obtained significant gains in both space and time. The verification of one example took 18 seconds and 1 MB of memory when lazy composition was used. The same example took about the same amount of time but twice as much memory when partitioned transition relations were used. If neither method was used, verification required more than 40 seconds and 12 MB. A significant part of the savings in both methods results from not constructing the global transition relation. However, lazy parallel composition often requires much less memory. The reason seems to be that partitioned transition relations are heavily influenced by the order in which partitions are processed, because this order determines which variables can or cannot be quantified out early. In lazy parallel composition this does not happen, since all of the variables are quantified out at the same time. This makes it less susceptible to the order in which partitions are processed, and more suitable to be used in the cases in which determining the processing order can be difficult. It also makes the new technique easier to automate.

5 Interface Processes

An important observation leads to another approach to compositional verification. The state explosion problem is usually most severe for *loosely coupled* processes which communicate using a small number of shared variables.

5.1 Cone of Influence Reduction

Suppose we are given a set of variables σ that we are interested in with respect to the process P. We can simplify the process P using the *cone of influence* reduction. Assume that the system is specified by a set of equations:

$$v_i' = f_i(V).$$

Define the cone of influence C_i of v_i for each variable v_i as the minimal set of variables such that

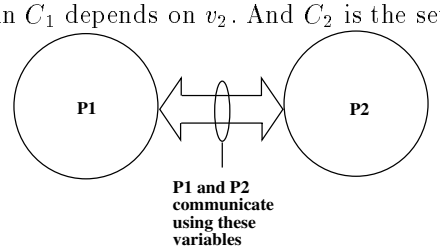
- $v_i \in C_i$,
- if for some $v_l \in C_i$ its f_l depends on v_j, then $v_j \in C_i$.

Construct a new (reduced) process P' from P by removing all the equations whose left hand side variables do not appear in any of the C_i's for $v_i \in \sigma$. It can be easily shown that $P \models \varphi$ iff $P' \models \varphi$, whenever φ contains only variables from σ.

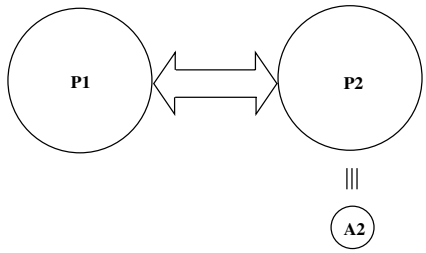
Again, consider our example of the modulo 8 counter (fig. 1). Its set of equations is

$$\begin{aligned} v_0' &= \neg v_0 \\ v_1' &= v_0 \oplus v_1 \\ v_2' &= (v_0 \wedge v_1) \oplus v_2 \end{aligned}$$

Clearly, $C_0 = \{v_0\}$, since f_0 does not depend on any variable other than v_0. We have $C_1 = \{v_0, v_1\}$, since f_1 depends on both of the variables, but $v_2 \notin C_1$ because no variable in C_1 depends on v_2. And C_2 is the set of all the variables.



Assume two processes P_1 and P_2 communicate using a set of variables σ. Then P_1 can only observe the behavior of P_2 through σ. It means that we can replace P_2 by any equivalent process A_2 which is indistinguishable from P_2 with respect to σ and this will completely preserve the behavior of P_1. The idea is to find a smaller process A_2 that hides all events irrelevant to σ.



The following *interface rule* guarantees the correctness of the abstraction A_2 with respect to P_1. Let $P|_\sigma$ be the restriction of P to the cone of influence of variables in σ, and $\mathcal{L}(\sigma)$ be the set of all CTL formulas with free variables from σ. The *interface rule* states that if the following conditions are satisfied:

- $P_2|_\sigma \equiv A_2$,
- $P_1||A_2 \models \varphi$,
- φ is a CTL formula such that $\varphi \in \mathcal{L}(\sigma)$,

then φ is also true in $P_1||P_2$. In fact, it is sufficient for φ to be in $\mathcal{L}(\Sigma_{P_1})$ for this rule to be sound, where Σ_{P_1} is the set of variables of P_1.

In the remainder of this section we describe how this strategy can be made precise and show how it can be used to reduce the state explosion problem for loosely coupled processes.

5.2 Soundness of the interface rule

In order for the interface rule to be sound we need to specify some properties that the process equivalence '$\equiv$' has to satisfy. For a process P let Σ_P be the set of atomic propositions (or state variables) in P, and let $\mathcal{L}(\Sigma)$ be the language of temporal formulas over the alphabet Σ. For any two processes P_1 and P_2 with sets of variables Σ_{P_1} and Σ_{P_2}, the following axioms have to be satisfied:

1. $P_1 \equiv P_2$ implies $\forall \varphi \in \mathcal{L}(\Sigma_{P_1})[P_1 \models \varphi \leftrightarrow P_2 \models \varphi]$
2. If $P_1 \equiv P_2$ then $P_1||Q \equiv P_2||Q$ and $Q||P_1 \equiv Q||P_2$
3. $(P_1||P_2)|_{\Sigma_{P_1}} \equiv P_1||(P_2|_{\Sigma_{P_1}})$ and $(P_1||P_2)|_{\Sigma_{P_2}} \equiv (P_1|_{\Sigma_{P_2}})||P_2$
4. If $\varphi \in \mathcal{L}(\Sigma_\varphi)$ and $\Sigma_\varphi \subseteq \Sigma_P$, then $P \models \varphi$ iff $P|_{\Sigma_\varphi} \models \varphi$

Theorem 1 (Soundness). *The Interface Rule is sound.*

To remind the reader, the interface rule states that

- $P_2|_{\Sigma_{P_1}} \equiv A_2$,
- $P_1||A_2 \models \varphi$,
- φ is a CTL formula such that $\varphi \in \mathcal{L}(\Sigma_{P_1})$,

imply $P_1||P_2 \models \varphi$. Notice, that restricting P_2 to Σ_{P_1} produces the same result as $P_2|_\sigma$, where $\sigma = \Sigma_{P_1} \cap \Sigma_{P_2}$.

Proof. Since $P_2|_{\Sigma_{P_1}} \equiv A_2$, then by 2 $P_1\|A_2 \equiv P_1\|(P_2|_{\Sigma_{P_1}})$. By 3, $P_1\|(P_2|_{\Sigma_{P_1}}) \equiv (P_1\|P_2)|_{\Sigma_{P_1}}$, hence we also have $P_1\|A_2 \equiv (P_1||P_2)|_{\Sigma_{P_1}}$. And since $P_1\|A_2 \models \varphi$ and $\varphi \in \mathcal{L}(\Sigma_{P_1})$, by 1 we derive $(P_1\|P_2)|_{\Sigma_{P_1}} \models \varphi$, and from 4 we immediately get $P_1||P_2 \models \varphi$ as required.

5.3 Equivalence of Processes

We define concrete equivalence relations over the processes that fulfil our requirements and are the most suitable in our framework. We use *bisimulation equivalence* and *stuttering equivalence* with synchronous parallel composition. We also give an "efficient" polynomial algorithm to determine bisimulation equivalence between processes and a sketch of the algorithm for stuttering equivalence.

Definition 1. *A* model *is a triple* $M = (S, N, L)$*, where* S *is a set of states,* $N \subseteq S \times S$ *is a transition relation and* L *is a* labeling function *mapping each state into a set of* atomic propositions *that are true in that state.*

Bisimulation Equivalence. Consider two models $M = (S, N, L)$ and $M' = (S', N', L')$ with the same set of atomic propositions.

Definition 2. *A binary relation* $E \subseteq S \times S'$ *is called a* bisimulation relation *if for any* $s \in S$ *and* $s' \in S'$*,* $E(s, s')$ *implies* $L(s) = L'(s')$ *and*

$$(i)\ \forall r \in S.N(s, r) \Rightarrow \exists r' \in S' : N'(s', r') \wedge E(r, r')$$

$$(ii)\ \forall r' \in S'.N'(s', r') \Rightarrow \exists r \in S : N(s, r) \wedge E(r, r').$$

Definition 3. *A* bisimulation equivalence *is the maximum bisimulation relation in the subset inclusion preorder.*

Notice that the definition of a bisimulation relation can be viewed as a fixpoint equation. Hence, the bisimulation equivalence is just the greatest fixpoint of that equation. This gives rise to a simple polynomial algorithm for computing the bisimulation equivalence using the well known iterative procedure. We compute a (decreasing) sequence of relations $E_0, E_1, \ldots$ until this sequence converges to a fixpoint at the n-th step. This convergence is guaranteed in finite-state case, since the subset inclusion preorder is well-founded in both directions. Choosing an appropriate E_0 guarantees that this fixpoint is the greatest fixpoint, therefore E_n is the required bisimulation equivalence. The sequence of relations is defined inductively as follows:

1. sE_0s' iff $L(s) = L'(s')$,
2. $sE_{n+1}s'$ iff $L(s) = L'(s')$ and
   - $\forall s_1[N(s, s_1)$ implies $\exists s'_1[N'(s', s'_1) \wedge s_1 E_n s'_1]]$
   - $\forall s'_1[N'(s', s'_1)$ implies $\exists s_1[N(s, s_1) \wedge s_1 E_n s'_1]]$

The complexity of this algorithm is $O(m^2)$, where m in the sum of the sizes of the transition relations. There are more efficient algorithms for computing bisimulation equivalence, for example the Paige-Tarjan algorithm [24]. It's complexity is $O(m \log n)$ in time and $O(m + n)$ in space, where n is the sum of the numbers of states in both models, and m in the sum of the sizes of the transition relations. However, it is unclear if this algorithm can employ BDDs as well.

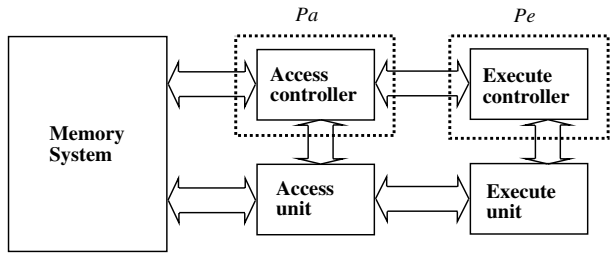


Fig. 4. A CPU controller.

Stuttering Equivalence. Unlike bisimulation, the *stuttering equivalence* [4, 16] is usually defined over the *computation paths* of the models. Intuitively, two paths π and π' are considered stuttering equivalent if they can be partitioned into finite blocks of repeated, or *stuttered* states, and corresponding blocks are equivalent in the two paths relative to the labeling functions L and L' of the models. Thus, we do not distinguish between two executions that differ only in the number of idle cycles between transitions. The stuttering equivalence also has a definition in terms of the greatest fixpoint.

Definition 4. *A binary relation $E \subseteq S \times S'$ is called a* stuttering relation *if for any $s \in S$ and $s' \in S'$, $s\ E\ s'$ implies $L(s) = L'(s')$ and*

$$(i)\ \forall r.\ N(s,r) \Rightarrow \exists s'_0, \ldots, s'_n\ (n \geq 0).\ s'_0 = s' \text{ and } r\ E\ s'_n \text{ and } \forall 0 \leq i < n.\ N'(s'_i, s'_{i+1}) \text{ and } s\ E\ s'_i;$$

$$(ii)\ \forall r'.\ N'(s',r') \Rightarrow \exists s_0, \ldots, s_m\ (m \geq 0).\ s_0 = s \text{ and } s_m\ E\ r' \text{ and } \forall 0 \leq i < m.\ N(s_i, s_{i+1}) \text{ and } s_i\ E\ s'.$$

Definition 5. *A* stuttering equivalence *is the maximum stuttering relation in the subset inclusion preorder.*

Stuttering equivalence preserves the truth of CTL* formulas that do not involve the next time operator **X** [4]. As in the case of bisimulation, we define inductively a sequence of relations $E_0, E_1, \ldots$ (that also converges in finite state case) and the stuttering equivalence is the intersection of all the E_i's. However, instead of computing the direct pre-image at each iteration as we did for bisimulation, we compute the set of states from which there is a path to the current state along which the current labeling $L(s)$ changes exactly once. This involves computing another least fixpoint. The details of the algorithm are described in [3]. A more efficient algorithm based on the Paige-Tarjan algorithm was found by Groote and Vaandrager [16] that runs in $O(mn)$ time. It is unknown, however, if this algorithm can use BDDs as well.

5.4 Interface Processes Example

As a simple example, we consider a model of the CPU controller [13] (fig. 4). The model comprises two parallel processes P_a and P_e called the access unit

and the execution unit. The access unit P_a fetches instructions and stores them in an instruction queue and maintains a cache of the top location of the CPU stack in a special register. The execution unit P_e pops out the instructions from the queue and interprets them. A major part of the temporal logic specification for CPU's controller defines correct behavior for the access unit and consists of formulas on the set of signals which are inputs or outputs of the unit. These signals constitute Σ_{P_a}. An example of such a formula is the following

$$\mathbf{AGAF}\, fetch$$

This formula is a liveness property which states that instructions are fetched from the access unit to the execution unit infinitely often. *Fetch* is actually a propositional formula defined in terms of request and acknowledge signals between the two units.

The parallel composition of the access unit and the execution unit in our design has approximately 1100 reachable states. However, by restricting the outputs of the execution unit to those in Σ_{P_a}, and then minimizing it, we obtain an interface process A_{P_e} such that $P_a \| A_{P_e}$ has only 196 reachable states. The reason for this reduction is that, while the execution unit interprets many different instructions, the memory accesses of these instructions fall into a few basic patterns.

6 Assume/Guarantee Reasoning

Assume-guarantee reasoning is a semi-automatic method that verifies each component separately. Ideally, compositional reasoning exploits the natural decomposition of a complex system into simpler components, handling one component at a time. In practice, however, when a component is verified it may be necessary to *assume* that the environment behaves in a certain manner. If the other components in the system *guarantee* this behavior, then we can conclude that the verified properties are true of the entire system. These properties can be used to deduce additional global properties of the system.

The *assume-guarantee paradigm* [17, 21, 23, 25] uses this method. Typically, a formula is a triple $\langle g \rangle M \langle f \rangle$ where g and f are temporal formulas and M is a program. The formula is true if whenever M is part of a system satisfying g, the system must also satisfy f. A typical proof shows that $\langle g \rangle M \langle f \rangle$ and $\langle true \rangle M' \langle g \rangle$ hold and concludes that $\langle true \rangle M \parallel M' \langle f \rangle$ is true. This proof strategy can also be expressed as an inference rule:

$$\frac{\langle true \rangle M' \langle g \rangle \quad \langle g \rangle M \langle f \rangle}{\langle true \rangle M \parallel M' \langle f \rangle}$$

The soundness of this simple assume-guarantee rule is straightforward.

In order to automate this approach, a model checker must be able to check that a property is true of *all* systems which can be built using a given component. More generally, it must be able to restrict to a given class of environments when

doing this check. An elegant way to obtain a system with this property is to provide a preorder $\preceq$ on the finite state models that captures the notion of "more behaviors" and to use a logic whose semantics is consistent with the preorder. The order relation should preserve satisfaction of formulas of the logic, i.e. if a formula is true for a model, it should also be true for any model which is smaller in the preorder. Additionally, composition should preserve the preorder, and a system should be smaller in the preorder than its individual components. Finally, satisfaction of a formula should correspond to being smaller than a particular model (a tableau for the formula) in the preorder.

Following Grumberg and Long [17], we use *synchronous process composition*, the *simulation preorder*, and the temporal logic *ACTL* (a subset of CTL without existential path quantifiers). This choice is motivated by the expressiveness of ACTL and the existence of a very efficient model checking algorithm for this logic. The simulation preorder is also a natural choice, since it is simple and intuitive as well as easily automated. We employ tableau construction methods for converting formulas into processes. Informally, a tableau for a formula φ is the greatest process A_φ (in the preorder) such that $A_\varphi \models \varphi$. In the remainder of this section we will not distinguish formulas and processes and will write, for example, $M \preceq \varphi$ to mean $M \preceq A_\varphi$.

It can be easily shown that our choice of formalisms meets all the requirements [17]. In particular, for all M and M' we have $M \| M' \preceq M$, and if $M' \preceq A$ then $M \| M' \preceq M \| A$, because synchronous composition can only restrict possible behaviors. Since M is greater than any system containing M, we can focus on proving properties of M in isolation. This insures that the same properties hold for an arbitrary system containing M.

Using the tableau construction we can verify $M \models \varphi$ by checking the relation $M \preceq \varphi$. In practice, however, we use classical model checking for verifying $M \models \varphi$ for a single component M if φ is given by a formula, and the simulation preorder if φ is an automaton, to increase the efficiency. Assumptions on the model correspond to composition. That is, a model M has the same set of behaviors under assumptions ψ as the model $M \| \psi$ without any assumptions. Thus, our triple $\langle \varphi \rangle M \langle \psi \rangle$ corresponds to $\varphi \| M \preceq \psi$. In other words, discharging assumptions corresponds to checking the preorder. Finally, the rule $M \preceq M \| M$ allows multiple levels of assume-guarantee reasoning.

Earlier we mentioned that the logic must preserve the preorder relation. Now we formalize and state the properties explicitly.

1. For all M, M' and φ, if $M \preceq M'$ and $M' \models \varphi$, then $M \models \varphi$ (removing behaviors cannot change a formula from true to false). Since $M \| M' \preceq M$, it is enough to check $M \models \varphi$ to know that any system containing M also satisfies φ.
2. For every φ, there is a structure T_φ such that $M \models \varphi$ if and only if $M \preceq T_\varphi$. This allows us to use φ as an assumption by composing M with T_φ.
3. Every model of φ is also a model of ψ if $T_\varphi \models \psi$

These lemmas are proved rigorously in [17] for synchronous composition of processes, the simulation preorder and the logic ACTL.

6.1 Implementation of Assume Guarantee Reasoning

Suppose we want to show that $M \| M' \models \psi$. That is, in terms of triples, we need to prove $\langle true \rangle M \| M' \langle \psi \rangle$. We verify that M satisfies some property ϑ by model checking. Next, using ϑ as an assumption, we show that M' satisfies some other auxiliary property φ. Finally, we show that M satisfies the required property ψ under the assumption φ. Since this extends to any system containing M, we are done. If the intermediate formulas (or processes) φ and ϑ are much smaller than M and M' respectively, then all the transition relations that must be constructed are significantly smaller than the one for $M \| M'$. This strategy for proving $M \| M' \models \psi$ can be summarized in the following assume-guarantee rule:

$$\frac{\langle true \rangle M \langle \vartheta \rangle \quad \langle \vartheta \rangle M' \langle \varphi \rangle \quad \langle \varphi \rangle M \langle \psi \rangle}{\langle true \rangle M \| M' \langle \psi \rangle}$$

In our framework, this corresponds to

$$\frac{M \preceq \vartheta \quad \vartheta \| M' \preceq \varphi \quad \varphi \| M \preceq \psi}{M \| M' \preceq \psi}$$

It is straightforward to show that this rule is sound by using the properties of preorder relation stated earlier.

Theorem 2. *The assume-guarantee rule is sound.*

Proof. Since $M \preceq \vartheta$, then $M \| M' \preceq \vartheta \| M'$. Since $\vartheta \| M' \preceq \varphi$, by transitivity $M \| M' \preceq \varphi$. Composing both sides with M we get $M \| M' \| M \preceq \varphi \| M$. Since parallel composition is commutative and associative, we can group the left hand side as $M \| M \| M'$. Then using $M \preceq M \| M$ and composing both sides with M' we obtain $M \| M' \preceq \varphi \| M$. Finally, from the last assumption $\varphi \| M \preceq \psi$ and transitivity we draw the conclusion of the rule $M \| M' \preceq \psi$.

So far, we have not discussed fairness. Both the preorder and the semantics of the logic should include some type of *fairness*. This is essential for modeling systems (hardware or communication protocols) at the appropriate level of abstraction. Moreover, fairness is necessary for the ACTL tableau construction.

Unfortunately, no efficient technique exists to check or compute *fair preorder* between models. In [17], Grumberg and Long suggest how to check the fair preorder only for a few trivial cases. Kupferman and Vardi showed that the general case is PSPACE-hard to compute [22]. Henzinger, Kupferman, and Rajamani [18] have proposed a new type of fair preorder that can be computed in polynomial time. However, it is not clear that this preorder is appropriate for compositional reasoning.

Example: The Futurebus+ Protocol. David Long has used this type of reasoning to verify safety and liveness properties for the Futurebus+ standard of cache coherence protocol [12, 19]. The whole design is divided into parallel

components that represent single modules like cache, memory, bus, etc. This example requires several levels of assumptions and guarantees.

The first stage of the verification was to check safety properties, since they can be verified using only forward reachability analysis and checking at each iteration that the current set of reachable states satisfies the property. Once a violation is found, the search is terminated immediately and an error trace is generated. The ability to terminate the search early was important since the BDD representing the set of reached states tended to become very large once an erroneous transition had occurred. As soon as all of the basic safety properties were satisfied, more complex formulas were checked in the state space restricted to the set of reachable states. Such a restriction also helped greatly in keeping the BDD from blowing up in size.

Using this technique he found specifications that were satisfied by a single bus configuration but not by multiple bus configurations. The details of the verification can be found in [12].

7 Conclusions

We describe several methods of dealing with the state explosion problem, which arises frequently due to parallel composition of processes. It is clear that compositional reasoning is critical in formal verification. Such techniques dramatically reduce the complexity of model checking and permit the verification of significantly larger systems. We have used compositional methods extensively to verify large complex systems such as the Futurebus+ [12] and the PCI bus [10,20] protocols.

This paper does not cover all of compositional proof techniques. There are a number of other compositional techniques that can also be successfully used. For example, *partial model checking* [1] encodes one of the processes into the formula, which is being checked, and simplifies the resulting formula. Similar method is described in [2]. Theorem proving techniques are also used to decompose and prove (manually) the property for each of the component [15,26].

In general, all of the compositional model checking techniques have their limitations and much work remains to be done. The most important problem is the trade-off between efficiency and automation. More powerful methods that can handle enormous complexity usually require an expert user and significant manual effort. These techniques usually rely on a powerful theorem prover under human guidance or careful choice of model checking parameters. On the other hand, completely automatic techniques frequently cannot handle extremely complex systems. The problem with automatic techniques is that they rely heavily on heuristics which may or may not work on different types of examples, and most of the intellectual work still has to be done by the user.

References

1. Henrik R. Andersen. Partial model checking (extended abstract). Technical Report ID-TR: 1994-148, Department of Computer Science, Technical University of

Denmark, October 1994. Accepted for LICS'95.
2. Henrik R. Andersen, Colin Stirling, and Glynn Winskel. A compositional proof system for the modal μ-calculus. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, France, 4–7 July 1994. IEEE Computer Society Press. Also as BRICS Report RS-94-34.
3. S. Berezin, E. Clarke, S. Jha, and W. Marrero. Model checking algorithms for the mu-calculus. Technical Report TR CMU-CS-96-180, Carnegie Mellon University, September 1996.
4. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2), July 1988.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
6. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI 91*, Edinburgh, Scotland, 1990.
7. Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.
8. S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: a tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995.
9. S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *IEEE Real-Time Systems Symposium*, 1994.
10. S. Campos, E. Clarke, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *Proceedings of the IEEE International Conference on Computer Design*, pages 73–79, 1995.
11. S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time System.* PhD thesis, SCS, Carnegie Mellon University, 1996.
12. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1993.
13. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, June 1989.
14. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
15. M. Dam. Compositional proof systems for model checking infinite state processes. In *Proceedings of CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag, 1995.
16. J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M. Paterson, editor, *Proceedings* 17^{th} *ICALP,* Warwick, volume 443 of *Lecture Notes in Computer Science*, pages 626–638. Springer-Verlag, July 1990.
17. Orna Grumberg and David Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

18. T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. In *Proc. of the 7th Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *LNCS*, Warsaw, July 1997.
19. IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, 1994. IEEE Standard 896.1, 1994 Edition.
20. Intel Corporation. *PCI Local Bus Specification*, 1993.
21. B. Josko. Verifying the correctness of AADL-modules using model checking. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400. Springer-Verlag, May 1989.
22. O. Kupferman and M. Y. Vardi. Module checking revisited. In O. Grumberg, editor, *Proc. of the 9th conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 36–47, Haifa, June 1997.
23. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4), July 1981.
24. R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal on Computing*, 16(6), Dec 1987.
25. A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F, Computer and system sciences.* Springer-Verlag, 1984.
26. C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–348, July 1987.
27. H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *IEEE Int. Conf. Computer-Aided Design*, pages 130–133, 1990.

Modeling Urgency in Timed Systems

Sébastien Bornot, Joseph Sifakis and Stavros Tripakis

Verimag
Centre Équation, 2, rue de Vignate, 38610, Gières, France
E-mail: {bornot,sifakis,tripakis}@imag.fr

1 Introduction

Timed systems can be modeled as automata (or, generally, discrete transition structures) extended with real-valued variables (*clocks*) measuring the time elapsed since their initialization. The following features are also common in the above models.

- States are associated with *time progress conditions* specifying how time can advance. Time can progress at a state by t only if all the intermediate states reached satisfy the associated time progress condition.
- At transitions, clock values can be tested and modified. This is usually done by associating with transitions guards (conditions on clocks) and assignments. If a guard is true from an automaton state and a given clock valuation, the corresponding transition can be executed by modifying clocks as specified by the corresponding assignment.

Time progress conditions can be used to specify urgency of transitions. Maximal urgency is achieved at a state if the corresponding time progress condition is equal to the negation of the disjunction of the guards of the transitions issued from this state. This implies that waiting at the state is allowed only if there is no enabled transition. As soon as a transition is enabled, time cannot progress anymore and the execution of the enabled transition(s) is enforced. Minimal urgency is achieved at a state when the corresponding time progress condition is true which implies that time can advance forever from this state and consequently indefinite waiting is allowed.

Choosing appropriate time progress conditions for complex system specifications is not a trivial problem as it is claimed in [SY96,BS97b,BS97a]. In many papers, time progress conditions have been defined as *invariants* that must be continuously true by clock valuations at the corresponding states. This implies that when a state is reached the associated invariant must be satisfied and makes modeling of absolute urgency sometimes difficult (for instance, in the case where a transition must be executed as soon as it is enabled).

The problem of the definition and use of time progress conditions has been tackled in [SY96,BS97b]. The purpose of this work is to show how the application of results presented in [BS97b] leads to a modeling methodology for timed systems. Emphasis is put on pragmatic and methodological issues. The basic ideas are the following.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 103-129, 1998.
© Springer-Verlag Berlin Heidelberg 1998

- A timed system can be specified as the composition of *timed transitions*. The latter are transitions labeled, as usual, with guards and assignments but also with deadlines, conditions on the clocks that characterize the states at which the transition is enforced by stopping time progress. We require that the deadline of a transition implies its guard, so that whenever time progress is stopped the transition is enabled.
- The guards and deadlines may contain formulas with past and future modalities concerning the evolution of clock values at a state. The use of such modalities does not increase the expressive power of the model but drastically enhances comfort in specification.

The paper is organized as follows. In section 2 we define Timed Automata with Deadlines (TAD) which are a class of Timed Automata [ACD93,HNSY94] where time progress conditions depend on deadlines associated with transitions. We show that using TAD makes urgency specification easier. In section 3 we present the model of Petri Nets with Deadlines (PND), which are (1-safe) Petri nets extended with clocks exactly as TAD are extensions of automata. We compare PND with different classes of Timed Petri Nets (TPNs) and show that safe TPNs can be modeled as PND. Section 4 presents some applications to modeling systems and in particular to modeling multimedia documents.

2 Timed Automata with Deadlines

2.1 Definitions

Definition 1 *(Timed Automaton with Deadlines (TAD))*
A TAD is :

- A discrete labeled transition system $(S, \rightarrow, A)$ where
  - S is a finite set of discrete states
  - A is a finite vocabulary of actions
  - $\rightarrow \subseteq S \times A \times S$ is a untimed transition relation
- A set $X = \{x_1, \ldots, x_m\}$ of real-valued variables called *clocks* with $dom(x_i) \in \mathbf{R}_+$.
- A labeling function h mapping *untimed transitions*, elements of $\rightarrow$, into *timed transitions*: $h(s, a, s') = (s, (a, g, d, r), s')$, where
  - g, d are respectively the *guard* and the *deadline* of the transition. Guards and deadlines are predicates p defined by the following grammar :

$$p ::= x \# c \mid x - y \# c \mid p \wedge p \mid \neg p$$

    where $x, y \in X$, c is an integer and $\# \in \{\leq, <\}$. We assume that $d \Rightarrow g$.
  - $r \subseteq X$ is a set of clocks to be reset.

Definition 2 *(Semantics of a TAD)*
A *state* of a TAD is a pair (s, v), where $s \in S$ is a discrete state and $v \in \mathbf{R}_+^m$

is a *clock valuation*. We associate with a TAD a transition relation $\rightarrow \subseteq (S \times \mathbf{R}_+^m) \times (A \cup \mathbf{R}_+) \times (S \times \mathbf{R}_+^m)$. Transitions labeled by elements of A correspond to *discrete state changes* while transitions labeled by non-negative reals correspond to *time steps*.

Given $s \in S$, if $\{(s, a_i, s_i)\}_{i \in I}$ is the set of all the transitions issued from s and $h(s, a_i, s_i) = (s, (a_i, g_i, d_i, r_i), si)$ then :

- $\forall i \in I \ \forall v \in \mathbf{R}_+ \ . \ (s, v) \xrightarrow{a_i} (s_i, v[r_i])$ if $g_i(v)$ where $v[r_i]$ is the variable valuation obtained from v when all the clocks in r_i are set at zero (and the others left unchanged).
- $(s, v) \xrightarrow{t} (s, v + t)$ if $\forall t' < t \ . \ c_s(v + t')$ where $c_s = \neg \bigvee_{i \in I} d_i$ and $v + t$ is the valuation obtained from v by increasing all the clock values by t.
  We call c_s the *Time Progress Condition* (TPC) associated with the discrete state s.

We consider TAD such that for any state s the TPC c_s is right-open.

2.2 About time-progress conditions

Notice that the simplest TAD is a single timed transition $(s, (a, g, d, r), s')$ with untimed transition (s, a, s'), guard g, deadline d and reset set r. The guard g characterizes the set of states from which the timed transition is possible while the deadline d characterizes the subset of these states where the timed transition is enforced by stopping time progress. The relative position of d with respect to g determines the *urgency* of the action. For a given g, the corresponding d may take two extreme values: first, $d = g$, meaning that the action is *eager* and, second, $d = false$, meaning that the action is *lazy*. A particularly interesting case is the one of a *delayable* action where d is the *falling edge* of a right-closed guard g (cannot be disabled without enforcing its execution). The above cases are illustrated in figure 1.

The condition $d \Rightarrow g$ guarantees that if time cannot progress at some state, then at least one action is enabled from this state. Restriction to right-open TPCs guarantees that deadlines can be reached by continuous time trajectories and permits to avoid deadlock situations in the case of eager transitions. (For instance, consider the case where $d = g = x > 2$, implying the TPC $x \leq 2$, which is not right-open. Then, if x is initially 2, time cannot progress by any delay t, according to definition 2.1 above. The guard g is not satisfied either, thus, the system is deadlocked.) The assumptions above ensure the property of *time reactivity*, that is, time can progress at any state unless a untimed transition is enabled.

Branching from a state s can be considered as a non-deterministic choice operator between all the timed transitions issued from this state. The resulting untimed transition relation is the union of the untimed transition relations of the combined timed transitions. The resulting time step relation is the intersection of the time step relations of the combined timed transitions.

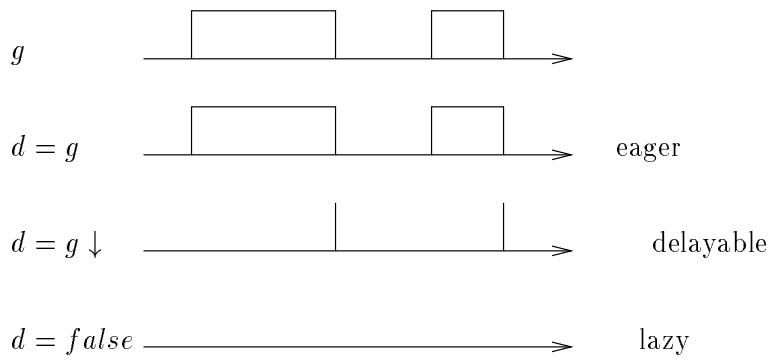


Fig. 1. Using deadlines to specify urgency.

Compared to the Timed Automata (TA) model [HNSY94], TAD differ in that TPCs are not given explicitly but rather derived from the deadlines which specify urgency of individual timed transitions. Thus, TAD are a subclass of TA that are time-reactive.

We believe that using deadlines rather than directly TPCs allows an easier modeling of urgency. Consider, for example, the TA in figure 2 which differ only in their TPCs. Clearly, the TA (1) and (2) specify the same behavior when s is reached with values $x \leq 5$. However, (1) does not satisfy the time reactivity requirement and cannot be obtained from a TAD, while (2) can be obtained by supposing that a is delayable ($d_1 = 5$) and b is eager or delayable ($d_2 = 5$). The case (3) corresponds to eager actions a and b and (4) to lazy actions.

Definition 3 *(Urgency types)*
For convenience, we replace explicit deadlines in transitions by the *urgency types* $\wr$, $\downarrow$, $\wr$, which are simply notations meaning that a transition is eager ($d = g$), delayable ($d = g \downarrow$), lazy ($d = false$), respectively.

Notice that any TAD can be transformed into an equivalent TAD with only eager and lazy transitions.

For complex systems, computation of TPCs from deadlines of transitions may be useful as shown by the following example. In table 2.2 we give the TPCs c_s associated with state s (figure 3) for different types of urgency ($\wr$=eager, $\downarrow$ = delayable, $\wr$=lazy) of the transitions (s, a_1, s_1) and (s, a_2, s_2).

δ_2 \ δ_1	$\wr$	$\downarrow$	$\wr$
$\wr$	$false$	$x = 5$	$2 \leq x \leq 5$
$\downarrow$	$y = 7$	$y = 7 \vee x = 5$	$y = 7 \vee 2 \leq x \leq 5$
$\wr$	$4 \leq y \leq 7$	$4 \leq y \leq 7 \vee x = 5$	$4 \leq y \leq 7 \vee 2 \leq x \leq 5$

Notice that the use of urgency types to induce deadlines could lead to right-closed TPCs (for example, consider the case where a transition is eager and has

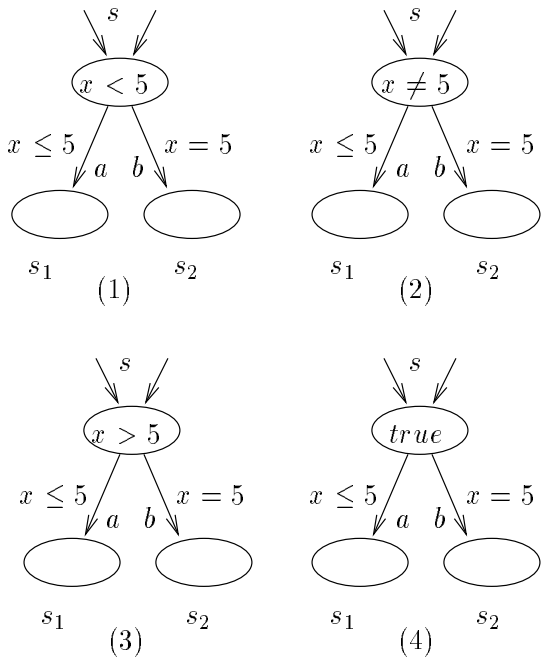


Fig. 2. Modeling urgency with TPCs.

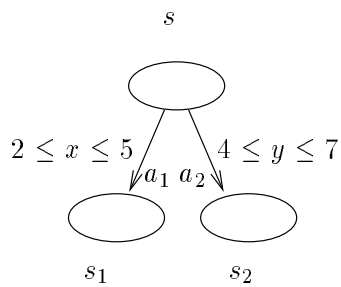


Fig. 3. Computing TPCs.

a left-open guard, say, $1 < x < 2$). This can be avoided by ensuring that eager transitions have always left-closed guards.

2.3 Priority Choice

It is often useful to consider that some priority is applied when from a state several timed transitions are enabled. This amounts to taking the non-deterministic choice between the considered transitions by adequately restricting the guards of the transitions with lower priority.

Consider, for example, two timed transitions $(s, (a_i, g_i, d_i, r_i), s_i)$ for $i = 1, 2$ with a common source state s. If a_1 has lower priority than a_2 in the resulting TAD the transition labeled by a_2 does not change while the transition labeled by a_1 becomes $(s, (a_1, g'_1, d'_1, r_1), s_1)$ where $g'_1 \Rightarrow g_1$ and $d'_1 = d_1 \wedge g'_1$.

Commonly, g'_1 is taken to be $g_1 \wedge \neg g_2$, which means that whenever a_1 and a_2 are simultaneously enabled, a_1 is disabled in the prioritized choice. However, for timed systems other ways to define g'_1 are possible. One may want to prevent action a_1 to be executed if it is established that a_2 will be eventually executed within a given delay.

For this reason we need the following notations.

Definition 4 *(Modal operators)*
Given a predicate p on X as in definition 2.1, we define the modal operators $\Diamond_{\leq k}\, p$ ("eventually p within k") and $\Diamond\!\!\!\!-_{\leq k}\, p$ ("once p since k"), for $k \in \mathbf{R}_+ \cup \{\infty\}$.

$$\Diamond_{\leq k}\ p\ (v) \text{ if } \exists t \in \mathbf{R}_+\ 0 \leq t \leq k.\ p(v+t)$$
$$\Diamond\!\!\!\!-_{\leq k}\ p\ (v) \text{ if } \exists t \in \mathbf{R}_+\ 0 \leq t \leq k.\ \exists v' \in V.\ v = v' + t \wedge p(v')$$

We write $\Diamond p$ and $\Diamond\!\!\!\!- p$ for $\Diamond_{\leq \infty}\ p$ and $\Diamond\!\!\!\!-_{\leq \infty}\ p$, respectively, and $\Box p$ and $\boxminus p$ for $\neg\Diamond\neg p$ and $\neg\Diamond\!\!\!\!-\neg p$, respectively.

Notice that modalities can be eliminated to obtain simple predicates without quantifiers. For example, $\Diamond(1 \leq x \leq 2)$ is equivalent to $x \leq 2$. For notational convenience, we shall be using in the sequel guards and deadlines with modalities.

Coming back to the example above, we can take $g'_1 = g_1 \wedge \neg\Diamond_{\leq k} g_2$ or even $g'_1 = g_1 \wedge \Box\neg g_2$. In the former case, a_1 gives priority up to a_2 if a_2 is eventually enabled within k time units. In the latter case, a_1 is enabled if a_2 is disabled forever.

It is shown in [BS97b] that for timed systems it is possible to define priority choice operators applicable to a set of timed transitions and parameterized by a priority relation $< \subseteq A \times \mathbf{R}_+ \times A$. If $(a_1, k, a_2) \in <$ (denoted $a_1 <_k a_2$) then the priority choice applied to a given set of timed transitions restricts the guard g_1 of a transition labeled by a_1 so as to disable a_1 whenever a_2 is to be enabled within k time units. In [BS97b] is is also shown that if the priority order satisfies some "transitivity conditions" then the corresponding priority choice preserves deadlock freedom in the following sense: If $\{g_i\}_{i\in I}$ are the guards of a set of timed transitions and $\{g'_i\}_{i\in I}$ are the modified guards obtained by application of the priority-choice operator then $\Diamond \bigvee_{i\in I} g_i \equiv \Diamond \bigvee_{i\in I} g'_i$ and $\Diamond g_i \Rightarrow \Diamond(g'_i \vee$

$\bigvee_{\exists k.\, a_i <_k a_j} g'_j$). The latter property says that if from a state the i-th transition is eventually enabled in the non-deterministic choice, then in the prioritized choice, either the i-th transition will be eventually enabled, or some transition of higher priority.

Let us illustrate the above ideas with an example. Consider the priority choice between two timed transitions with respective labels (a_i, g_i, d_i, r_i), $i = 1, 2$, such that a_1 has lower priority than a_2, where $g_1 = 0 \leq x \leq 4 \vee x \geq 6$ and $g_2 = 2 \leq x \leq 7$ for some x. We get the following decreasing values for ${g'}_1$ as the priority delay increases:

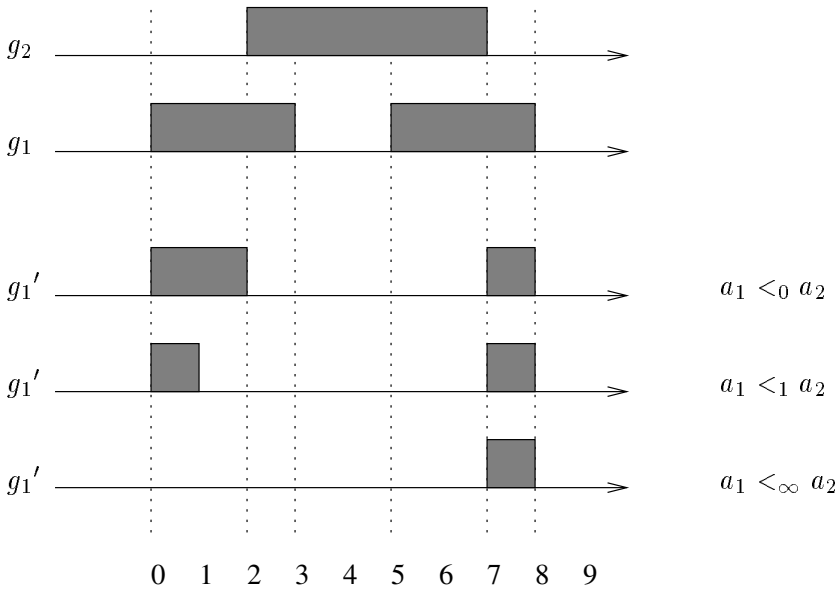


Fig. 4. Different priorities for a_2 over a_1.

$g'_1 = g_1 \wedge \neg g_2 = 0 \leq x < 2 \vee x > 7$ (immediate priority)
$g'_1 = g_1 \wedge \neg \Diamond_{\leq 1} g_2 = 0 \leq x < 1 \vee x > 7$ (priority within a delay of 1)
$g'_1 = g_1 \wedge \neg \Diamond g_2 = x > 7$ (priority within an infinite delay)

Figure 4 illustrates the above example. The first case corresponds to the "classical" priority choice, where a_1 is disabled whenever a_2 is enabled. The second case is stronger: a_1 is disabled also in case a_2 becomes enabled in at most 1 time unit. The third case is the strongest: a_1 is disabled whenever it is possible for a_2 to become enabled sometime in the future.

Finally, we should note that the use of negations to generate priority could lead to right-closed TPCs. When urgency types are used, this can be avoided by ensuring that a lazy transition never has higher priority over an eager transition.

3 Petri Nets with Deadlines

3.1 Definition

For the sake of simplicity, we consider the timed extensions of *1-safe* Petri nets.

Definition 5 *(Petri Net with Deadlines (PND))*
A PND consists of :

- A (1-safe) Petri net $(\mathcal{P}, \mathcal{T}, A)$ where :
  - $\mathcal{P}$ is a finite set of places.
  - A is a finite vocabulary of actions.
  - $\mathcal{T} \subseteq 2^{\mathcal{P}} \times A \times 2^{\mathcal{P}}$ is a transition relation.
- A set $X = \{x_1, \ldots, x_m\}$ of clocks.
- A labeling function h mapping untimed transitions elements of $\mathcal{T}$ into timed transitions : $h(P, a, P') = (P, (a, g, d, r), P')$, where $P, P' \subseteq \mathcal{P}$.

As usually, we represent a PND as a bipartite labeled graph with two types of nodes (places and transitions), see figure 5. The transitions are labeled with action names, guards, deadlines and resets.

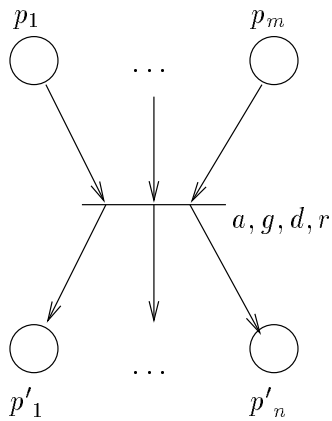


Fig. 5. The transition $(\{p_1, \ldots, p_m\}, (a, g, d, r), \{p'_1, \ldots, p'_n\})$.

We define the semantics of a PND in terms of a TAD.

Definition 6 *(TAD associated to a PND)*
A PND $(\mathcal{P}, \mathcal{T}, A, X, h)$ defines a TAD $(S, \rightarrow, A, X, h')$ such that :

- $S = 2^{\mathcal{P}}$
- $P \xrightarrow{a} P'$ if $(P, a, P') \in \mathcal{T}$
- $h'(P, a, P') = h(P, a, P')$.

The above definition simply means that a PND is a TAD where the discrete transition structure is the corresponding marking graph. The transitions of the marking graph are submitted to the same timing constraints as the transitions of the PND. So PND are extensions of PNs where transitions are submitted to timing constraints exactly as TAD are extension of automata.

By adopting standard PN terminology, we will say that there is a token in place p when p is an element of the current state in the marking graph. Places are local states of processes. A transition with several input places represents a synchronization of several processes. It is enabled only if its input places have a token and the associated timing constraints are satisfied.

An example of PND is given in figure 6.

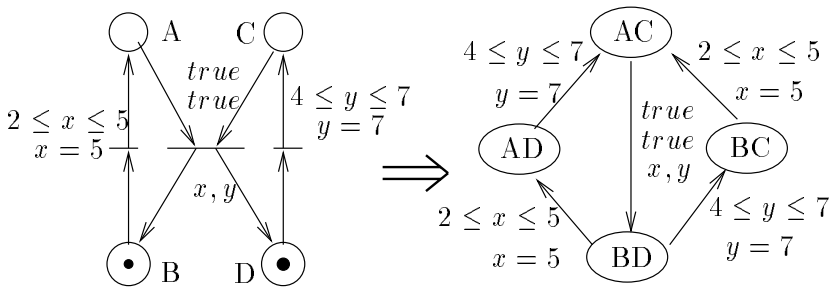


Fig. 6. A PND and its corresponding TAD.

3.2 Synchronization modes

We introduce some useful macro-notations that allow concise description of synchronization guards in terms of timing constraints about the synchronizing processes.

We first define three different synchronizing modes that correspond to different types of coordinations between processes. We suppose that, for a synchronization transition, are given "local guards" g_i expressing timing constraints about termination of each contributing process. We associate each guard g_i with an input arc of the synchronization transition (figure 7). A mode defines a way of composing the guards g_i to obtain the synchronization guard g.

AND-synchronization : The resulting guard g is the conjunction $g = \bigwedge_{i \in [1 \ldots n]} g_i$ of the input guards. This simply means that synchronization is possible only if all processes can terminate together. In the example of figure 8, we get $g = g_1 \wedge g_2 = 3 \leq x \leq 7$.

MAX-synchronization : Synchronization can take place only if all the contributing processes have terminated. This implies synchronization at times t

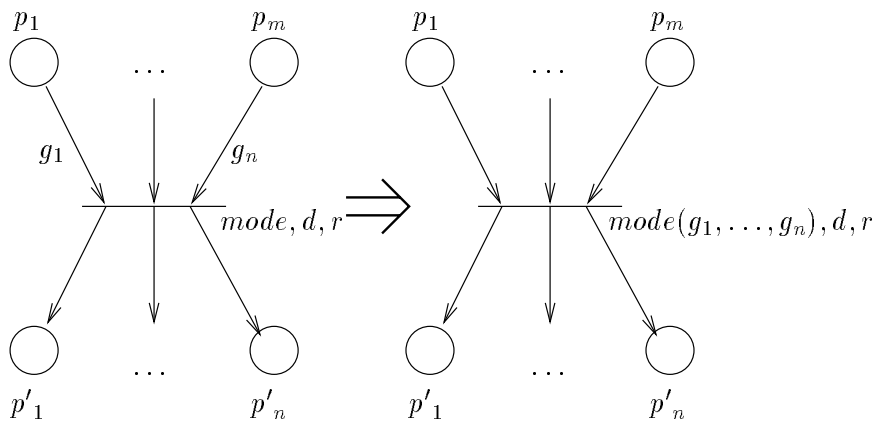


Fig. 7. Meaning of synchronization notation.

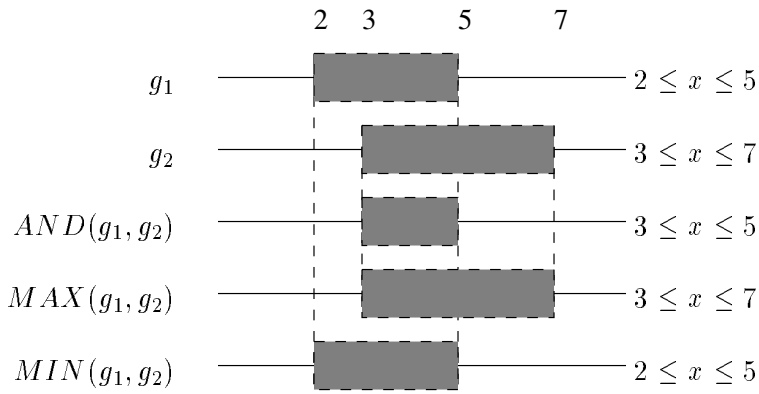


Fig. 8. Resulting guards for the three synchronization modes.

bounded by the maximum of the earliest termination times and the maximum of the latest termination times of the contributing processes.

For this synchronization mode, we take $g = \bigvee_{i\in[1\ldots n]} g_i \wedge \bigwedge_{j\neq i} \diamondsuit g_j$. The i-th term of the guard means that the i-th process can terminate (now) while the others have already terminated. This allows to specify synchronization with mutual waiting of all the contributing processes if $\diamondsuit g_i$ holds when the input place p_i is reached. Otherwise it may happen that before reaching an input place p_i the guard has been already satisfied but this does not correspond to termination of p_i.

In the example of figure 8, we get $g = g_1 \wedge (\diamondsuit g_2) \vee (\diamondsuit g_1) \wedge g_2 = 3 \leq x \leq 7$.

MIN-synchronization : Synchronization takes place when one of the contributing processes terminates and the others will eventually terminate. This corresponds to a kind of interrupt where the fastest process triggers the synchronization transition even though the other processes have not terminated. Notice that synchronization times t are bounded by the minimum of the earliest and the minimum of the latest termination time of the contributing processes.

We take $g = \bigvee_{i\in[1\ldots n]} g_i \wedge \bigwedge_{j\neq i} \diamondsuit g_j$. The i-th term of the guard means that the i-th process can terminate (now) and all the others will eventually terminate.

For the example of figure 8, we get $g = g_1 \wedge \diamondsuit g_2 \vee \diamondsuit g_1 \wedge g_2 = 2 \leq x \leq 5$.

3.3 Translating safe timed Petri nets into PND

Many different classes of timed Petri nets (TPNs) have been defined. An important difference between TPNs and PND is that in the former timing constraints are local and associated with tokens. A comparison of the two models in the general case of non-safe Petri nets is out of the scope of this paper and is the object of an ongoing work. Here, we restrict our attention to 1-safe TPNs.

Place-TPNs : [Sif77] In this class of TPNs, intervals $[l_i, u_i]$ are associated with places p_i. A token arriving at a place p_i cannot be used for firing an output transition for some time t, $l_i \leq t \leq u_i$. After this time it becomes available. A transition fires as soon as all its input places have available tokens.

The principle of a method for translating Place-TPNs to PND is illustrated in figure 9.

Transition-TPNs : [Mer74] In this class of TPNs, intervals $[l_i, u_i]$ are associated with transitions τ_i. A timed transition τ_i fires in times t, $l_i \leq t \leq u_i$ after the corresponding untimed transition becomes enabled.

The principle of a method for translating Transition-TPNs to PND is illustrated in figure 10.

Stream-TPNs : This type of TPNs is introduced in [SDdSS94]. Given a transition τ, an interval $[l_i, u_i]$ is associated with each input arc (p_i, τ) of τ. A token entering the input place p_i must wait for a time t, $l_i \leq t \leq u_i$, before becoming available for the transition τ.

Nine different synchronization modes for stream TPNs are defined in [SDdSS94] (see figure 11). For each input place p_i of the synchronization transition τ, two timers x_i and y_i are defined as follows: $x_i \stackrel{\text{def}}{=} max(l_i - t_i, 0)$

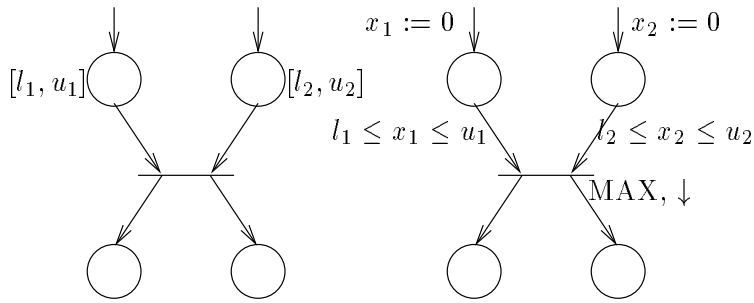


Fig. 9. From Place-TPNs to PND.

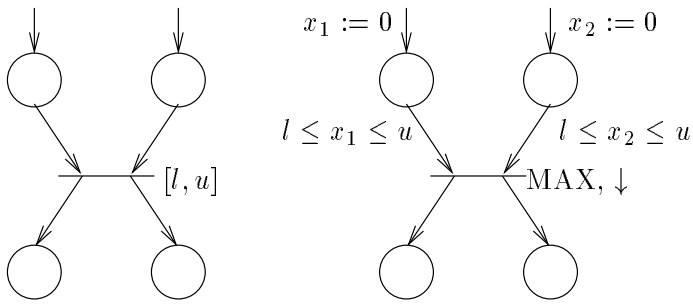


Fig. 10. From Transition-TPN to PND.

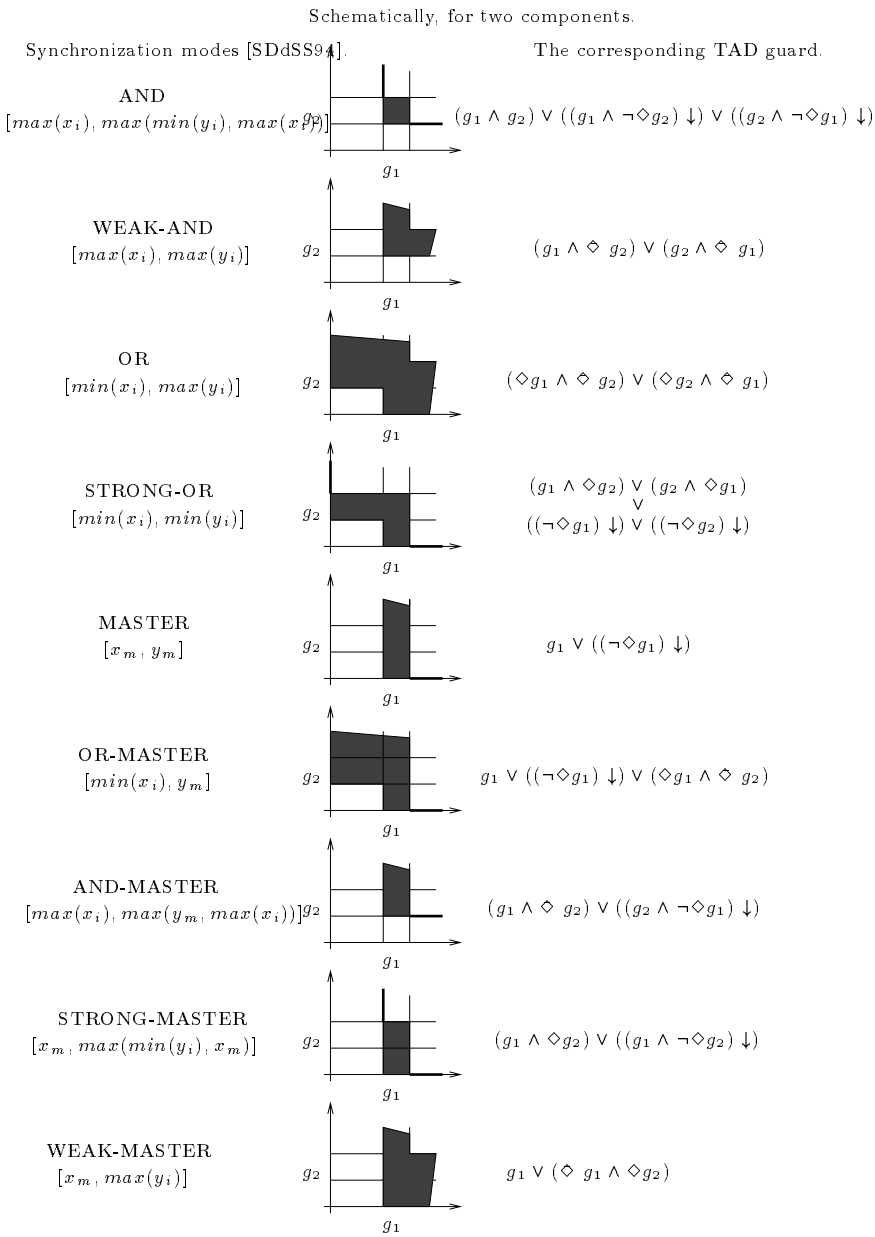


Fig. 11. Synchronization mode for Stream-TPNs.

and $y_i \stackrel{\text{def}}{=} max(u_i - t_i, 0)$, where t_i is the elapsed time since the arrival of the token at place p_i. Thus, x_i and y_i are taken to be the "current" lower and upper bounds for the enabledness of each input arc (p_i, τ).

The nine synchronization modes are shown in the leftmost column of figure 11. Notice that, in the figure, *max* and *min* denote the usual mathematical operators and are not to be confused with the MAX and MIN synchronization operators defined previously. Also, we write $max(x_i)$ as a shorthand for $max_{i=1,\ldots n}\{x_i\}$, and similarly for *min*. The middle column of the figure displays the guards induced by each of the synchronization modes, for $n = 2$, where $g_i = [l_i, u_i]$.

The model of stream-TPNs can also be translated into our model, as shown in the right-most column of figure 11.

4 Applications

4.1 Producer – Consumer

We show how PND can be used to model a system composed of a producer and a consumer communicating via a zero-length buffer. The producer takes between l_p and u_p time units to produce an item, which is then made available to the buffer after a delay between l'_p and u'_p time units. The consumer needs between l_c and u_c time units to consume an item and is ready for a new item after a delay between l'_c and u'_c time units. The above delays are measured using one clock per process, namely, x for the producer and y for the consumer. Figure 12(a) shows the two processes modeled as PND.

Whenever the buffer is full and the consumer is willing to take an item, the latter is exchanged between the two processes by an instantaneous *handshake*. The latter is represented by the synchronization transition of the PND corresponding to the composition of the two processes, shown in figure 12(b). The guard g of the handshake transition can be chosen to be either g' or g'', where:

$$\begin{aligned} g' &\equiv \text{AND}(l'_p \le x \le u'_p, l'_c \le y \le u'_c) \\ &\equiv l'_p \le x \le u'_p \ \wedge \ l'_c \le y \le u'_c \\ g'' &\equiv \text{MAX}(l'_p \le x \le u'_p, l'_c \le y \le u'_c) \\ &\equiv l'_p \le x \le u'_p \wedge l'_c \le y \ \vee \ l'_p \le x \wedge l'_c \le y \le u'_c \end{aligned}$$

In the first case, the temporal constraints are considered "hard", that is, it is required that both lower and upper bounds of the intervals $[l'_p, u'_p]$ and $[l'_c, u'_c]$ are respected in order for the handshake to take place. (An informal explanation of this choice could be that u'_p represents the "expiring date" of the item, while u'_c is the maximum time the consumer can wait, after which he/she "starves to death".) AND synchronization is commonly used in the composition of systems, however, it is a strict synchronization mechanism which often leads to deadlocks.

In the case of MAX synchronization, temporal constraints are "looser", that is, only one of the upper bounds is required to hold. (Informally, this might represent a more realistic situation, where the item never looses its value, while

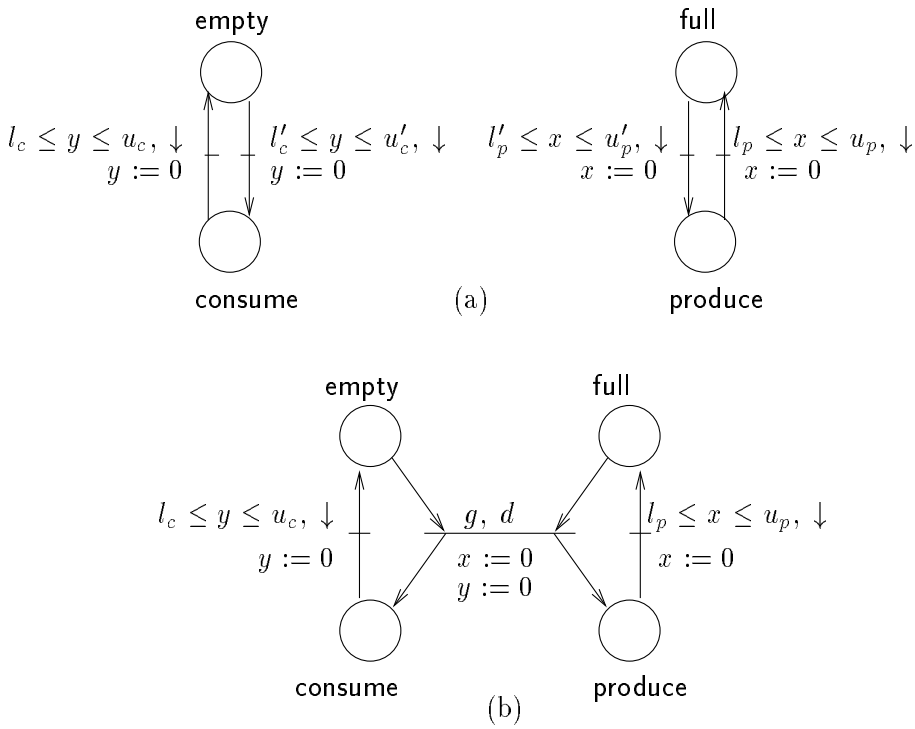


Fig. 12. Producer–Consumer system modeled as PND.

the consumer is willing to wait.) MAX synchronization guarantees the absence of deadlocks. Moreover, combined with appropriate deadlines, it can model synchronization with minimal or maximal waiting, as we show below.

Regarding the urgency type of the synchronization transition, in the case of AND synchronization it is reasonable to assume that the transition is delayable, which gives the deadline:

$$d' \equiv g'\downarrow \equiv x = u'_p \wedge l'_c \leq y \leq u'_c \vee y = u'_c \wedge l'_p \leq x \leq u'_p$$

In the case of MAX synchronization more than one possibilities are of interest, namely:

- The choice of delayble transition corresponds to a maximal-waiting policy:

$$d'' \equiv g''\downarrow \equiv x = u'_p \wedge y \geq u'_c \vee y = u'_c \wedge x \geq u'_p;$$

- The choice of eager transition corresponds to a minimal-waiting policy;
- The following choice corresponds to a "best-effort" synchronization scheme, where either no upper bound is violated if possible, or the transition is executed as soon as possible, in the case of violation:

$$d''' \equiv d' \vee x = l'_p \wedge y \geq u'_c \vee y = l'_c \wedge x \geq u'_p.$$

Notice that d''' cannot be obtained using any of the urgency types $\wr$, $\downarrow$ or $\wr$.

4.2 Variations on the theme of mutual exclusion

We consider the generic mutual-exclusion situation shown in figure 13. A resource is shared by two processes P_1 and $P2$ and can be used by at most one of them at any time. Each time it is used, the resource is again available after an amount of time which can vary in an interval I. Process P_i occupies the resource for an amount of time in an interval C_i, for $i = 1, 2$. From the moment it has finished using the resource, P_i is ready to use it again after some delay in an interval W_i. In the PND model shown in the figure, clocks x_1, x_2 and z are used for P_1, P_2 and the resource, respectively.

There are different policies of granting the resource to the processes, depending on how strict the temporal constraints of the problem are taken to be and also on whether an optimal utilization of the resource is sought. We examine some of these policies below, showing how they can be modeled by appropriately choosing the guards g_i and the urgency types δ_i shown in the figure, for $i = 1, 2$. We assume that $I = [l, u]$, $W_i = [l_i, u_i]$ and $C_i = [l'_i, u'_i]$, for $i = 1, 2$ (the analysis can be generalized to unbounded intervals).

- $g_i \equiv \mathrm{AND}(x_i \in W_i, z \in I)$. In this case the temporal constraints are hard. Then, if process P_i manages to get the resource, it is guaranteed to do so at most u_i time units after the time it has released it. On the other hand, the resource is guaranteed not to be left idle for more than u time units after it has been used for the last time. The problem of this method is that it can easily lead to deadlocks, either local (i.e., where one process starves) or global (i.e., where the whole system is blocked).

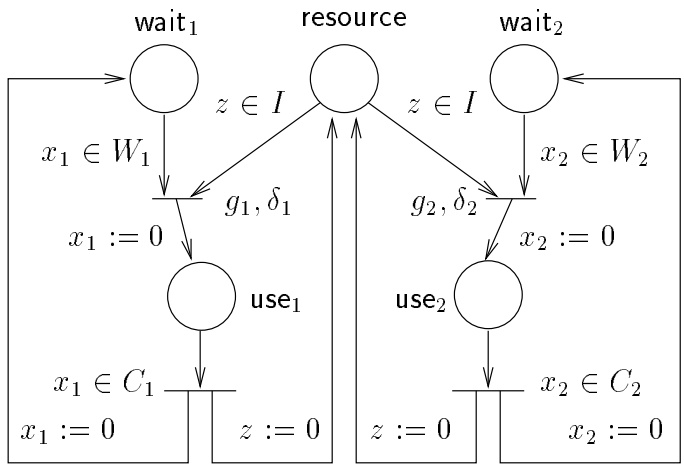


Fig. 13. Mutual exclusion modeled as PND.

- $g_i \equiv \mathrm{MAX}(x_i \in W_i, z \in I)$. In this case the temporal constraints are loose and the specification is deadlock-free for any (non-empty) intervals I, W_i and C_i.
- $g_i \equiv (x_i \in W_i) \wedge \diamondsuit\ (z \in I)$ or $g_i \equiv (\diamondsuit\ x_i \in W_i) \wedge (z \in I)$. These are intermediate choices, looser than AND synchronization, however, without avoiding deadlocks completely. In the first case, the upper bound of the resource's interval is ignored, while in the second case, the processes' upper bounds are ignored.

Regarding the urgency type of the synchronization transition, it can be chosen to be either eager or delayable (lazy synchronization is not meaningful in this case). Delayable is the less strict choice, minimizing the risk of deadlocks in the case MAX is not used. Eager implies that a better utilization of the resource (i.e., less idle time) is achieved. However, if MAX synchronization is not used, the risk of deadlocks is greater than in the delayable case, since the time non-determinism is reduced.

We finally consider the situation where process P_1 is given a higher priority with respect to process P_2. This is typically the case when P_1 demands the resource much less frequently than P_2 (for example, when P_1 is the process handling the keyboard, while P_2 is any batch process). We can model the different priorities by enforcing the guard g_2 into $g_2' = g_2 \wedge \neg\diamondsuit_{\leq u_2'} g_1$, where u_2' is the upper bound of interval C_2. The intention is to let P_2 have the resource only if it is guaranteed to finish before P_1 becomes ready.

4.3 Deadline-monotonic scheduling without preemption

We consider the following real-time scheduling problem. We are given a single processor and a set of *periodic* tasks $P_1, ..., P_n$ to be executed upon this processor.

Task P_i has a computation delay C_i and becomes ready for execution every T_i time units (the *period* of P_i). Furthermore, P_i needs to be completed at most D_i time units after the moment it becomes ready (the *deadline* of P_i). We assume that, for each $i = 1, ..., n$, we have $C_i \leq D_i \leq T_i$. The processor can execute only one process at a time and no *preemption* is allowed, that is, execution of a process cannot be interrupted and continued later on. See figure 14.

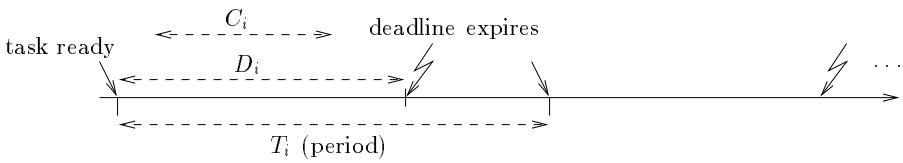


Fig. 14. Deadline-monotonic scheduling assumptions.

We show how Petri nets with deadlines can be used to model the so-called *deadline-monotonic* algorithm [ABRW91] which solves the above scheduling problem. [1] The algorithm is based on assigning *static priorities* to tasks according to their deadlines. In particular, higher priorities are assigned to tasks with shorter deadlines and no two tasks have the same priority (in case two tasks have equal deadlines, their relative ordering is chosen arbitrarily).

Figure 15(a) shows the PND modeling task P_i. The net has three places, namely, sleep_i (the task hasn't become ready yet), wait_i (the task is ready and waiting to be served) and use_i (the task is being served). Two clocks are used per task, namely, x_i and y_i: x_i counts the period T_i and also makes sure that the deadline D_i is not violated; y_i counts the computation delay C_i.

Figure 15(b) shows the PND modeling the deadline-monotonic scheduling algorithm for two tasks P_1 and P_2, assuming that the first one has higher priority (i.e., $D_1 \leq D_2$). The processor is modeled as a single place the token of which is necessary in order for a task to execute. Priority of P_1 over P_2 is ensured by placing the guard $x_1 < T_1$ in the transition $\mathsf{wait}_2 \rightarrow \mathsf{use}_2$. Transitions $\mathsf{wait}_i \rightarrow \mathsf{use}_i$ are both eager while all other transitions are delayable.

Using KRONOS, we test the schedulability of two tasks for various values of the parameters $C_i, D_i, T_i, i = 1, 2$. The test is performed as follows. We first replace the parameters by their values and generate the TAD corresponding to the resulting PND. Next, we translate this TAD to a classical TA with time-progress conditions by using extra clocks to specify the urgency of certain transitions. Finally, we test whether in the TA there exist reachable states which are zeno, that is, from which time can no longer progress. In fact, there are two cases: either all reachable states of this TA are zeno, meaning that the tasks are not schedu-

[1] We model a simplified version of the algorithm. Actually, deadline-monotonic scheduling uses preemption.

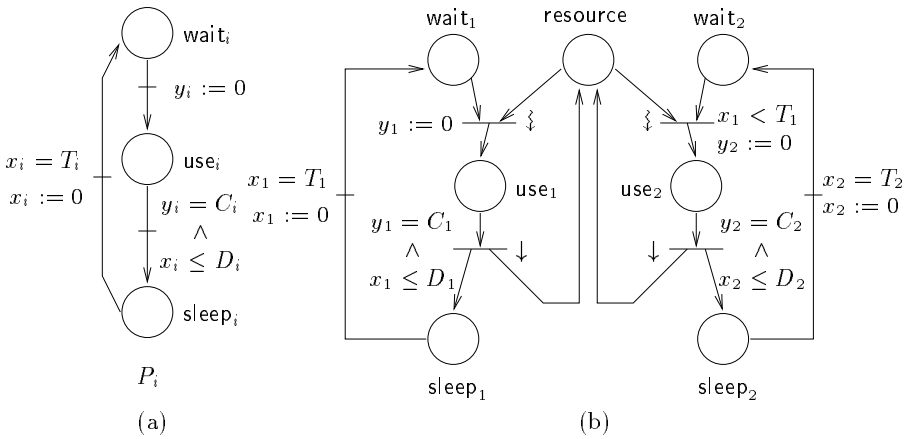


Fig. 15. Deadline-monotonic scheduling modeled as PND.

lable, or no zeno reachable states exist, which means that deadline-monotonic scheduling can be applied.

4.4 Specification and verification of multimedia documents

Description This application deals with modeling a multimedia document as a PND which can then be analyzed in order to check whether the document admits an execution scenario. More precisely, we consider (a simplified version of) MADEUS [JLSIR97] as the specification language of multimedia documents. This language combines operators from Allen's *interval temporal logic* [All83] with waiting and interruption operators.

The building blocks of a document are *media objects* representing a piece of information which has to be "played" continuously for a certain duration. The latter can be either fixed, or variable, in which case some flexibility is allowed in the presentation of the object. Let $\mathcal{O} = \{O_1, ..., O_n\}$ be the set of media objects. With each O_i we associate a *duration interval* I_i of one of the following types: $[l, u]$, $[l, u)$, $[l, \infty)$ or (l, ∞), where l, u are natural constants.

Documents are tree-like structures, built according to the following syntax:

$$\mathcal{D} ::= O \mid \mathcal{D}_1 \ \mathtt{op} \ \mathcal{D}_2$$

where $O \in \mathcal{O}$ and `op` is an operator among `meets`, `equals`, `overlaps`, `parmin`, `parmax`, and `parmaster`. We require that each object $O \in \mathcal{O}$ appears at most once in any document specification $\mathcal{D}$.

Each operator has a dual function: First, it builds a composite document from two simpler ones. Second, it imposes constraints on the order of the starting and finishing times of the component documents. These constraints can be trivial, as in the case of the `meets` operator, or more demanding, as in the case of `equals`, where consistency has to be ensured. Before giving the translation of a document

specification to a PND, let us present intuitively the meaning of the operators defined above.

- $\mathcal{D}_1$ `meets` $\mathcal{D}_2$ is the document starting when $\mathcal{D}_1$ starts, finishing when $\mathcal{D}_2$ finishes, and where the end of $\mathcal{D}_1$ coincides with the beginning of $\mathcal{D}_2$;
- $\mathcal{D}_1$ `equals` $\mathcal{D}_2$ is the document where $\mathcal{D}_1$ and $\mathcal{D}_2$ start and finish at the same time;
- $\mathcal{D}_1$ `overlaps` $\mathcal{D}_2$ is the document starting when $\mathcal{D}_1$ starts, finishing when $\mathcal{D}_2$ finishes, and where the beginning of $\mathcal{D}_2$ is strictly later than the beginning of $\mathcal{D}_1$, and the end of $\mathcal{D}_1$ is strictly later than the beginning of $\mathcal{D}_2$ and strictly earlier than the end of $\mathcal{D}_2$;
- $\mathcal{D}_1$ `parmin` $\mathcal{D}_2$ is the document where $\mathcal{D}_1$ and $\mathcal{D}_2$ start at the same time, and the one which finishes first terminates the document;
- $\mathcal{D}_1$ `parmax` $\mathcal{D}_2$ is the document where $\mathcal{D}_1$ and $\mathcal{D}_2$ start at the same time, and the one which finishes last terminates the document;
- $\mathcal{D}_1$ `parmaster` $\mathcal{D}_2$ is the document where $\mathcal{D}_1$ and $\mathcal{D}_2$ start at the same time, and the document finishes whenever $\mathcal{D}_1$ does.

Modeling With each media object $O_i, i = 1, ..., n$ we associate a clock x_i. Also, given a set of clocks X, we denote by $X := 0$ the resetting of each clock in X to zero.

We now define the translation of a document specification $\mathcal{D}$ to a PND $\mathcal{N}$. The definition is by induction on the syntax of $\mathcal{D}$. The media object O_i is translated to the net shown in figure 16.

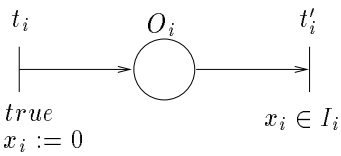


Fig. 16. The PND corresponding to the basic media object O_i.

In order to construct the PND for a document $\mathcal{D}_1$ `op` $\mathcal{D}_2$, we assume having already the PND $\mathcal{N}_1$ and $\mathcal{N}_2$ corresponding to $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. These nets have the general form shown in figure 17, that is, a single starting transition t_i, a single finishing transition t'_i guarded by g_i, and a body displayed as a dashed-line box in the figures. All the transitions are delayable, apart from the initial transition which is eager. Also, we assume that this is the case for all the PND resulting from the constructions shown in the sequel. It is easy to see that the PND of a basic object conforms to this general scheme. The constructions that are presented below preserve this general scheme.

Figures 18, 19 and 20 show the PND corresponding to $\mathcal{D}_1$ `meets` $\mathcal{D}_2$, $\mathcal{D}_1$ `equals` $\mathcal{D}_2$ and $\mathcal{D}_1$ `overlaps` $\mathcal{D}_2$, respectively. For the operators `parmin`,

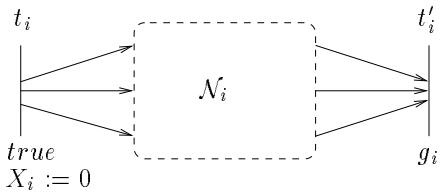


Fig. 17. The general form of PND $\mathcal{N}_i$ corresponding to a document $\mathcal{D}_i$.

`parmax` and `parmaster` the construction is identical to the one for `equals`, with the difference that the guard $g_1 \wedge g_2$ of the finishing transition t' is replaced by $\mathrm{MIN}(g_1, g_2)$, $\mathrm{MAX}(g_1, g_2)$ and $\mathrm{MASTER}(g_1, g_2)$, respectively.

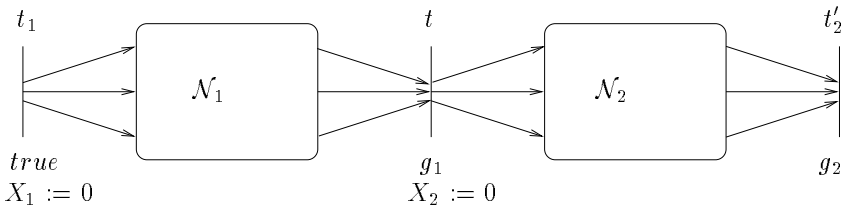


Fig. 18. The PND corresponding to document $\mathcal{D}_1$ `meets` $\mathcal{D}_2$.

Verification Given a document specification $\mathcal{D}$, we are interested in checking its *consistency*, that is, whether the temporal constraints imposed by the various operators and the duration intervals of each basic object are compatible. For instance, the specification O_1 `equals` O_2 is consistent if and only if the duration intervals I_1 and I_2 have non-empty intersection.

To check consistency, we proceed as follows. We first construct the net $\mathcal{N}$ corresponding to the specification $\mathcal{D}$. Next, we build the TAD $\mathcal{A}$ associated with $\mathcal{N}$ and add two extra locations `Begin` and `End` to $\mathcal{A}$. The former is the initial location, source of the (unique) edge of $\mathcal{A}$ corresponding to the starting transition of $\mathcal{N}$. `End` is the target location of the (unique) edge of $\mathcal{A}$ corresponding to the finishing transition of $\mathcal{N}$. `End` has no outgoing edges. Finally, we check whether `End` is reachable from `Begin`. If this is the case then $\mathcal{D}$ is consistent and we also obtain a sample execution scenario in the form of a run of the automaton $\mathcal{A}$. Otherwise, the specification is inconsistent. The reachability test is performed using the real-time verification tool KRONOS.

An example

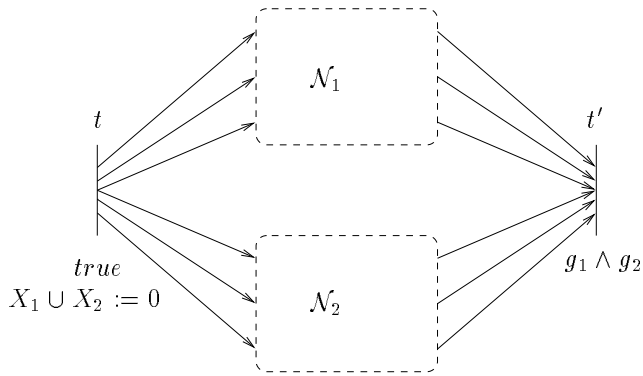


Fig. 19. The PND corresponding to document $\mathcal{D}_1$ `equals` $\mathcal{D}_2$.

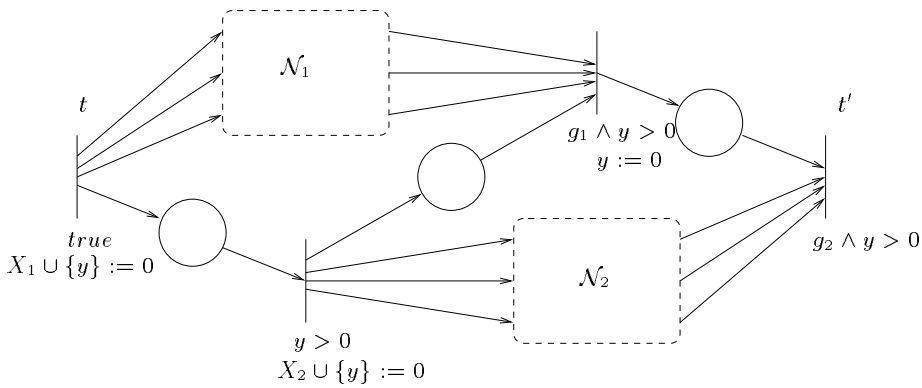


Fig. 20. The PND corresponding to document $\mathcal{D}_1$ `overlaps` $\mathcal{D}_2$.

Description. We consider the following document specification:

$$\begin{aligned}\mathcal{D} &\equiv \mathcal{D}_1 \texttt{ meets } \mathcal{D}_2\\ \mathcal{D}_1 &\equiv A \texttt{ equals } \big(B \texttt{ parmax } (C \texttt{ parmin } D)\big)\\ \mathcal{D}_2 &\equiv \big(E \texttt{ meets } (F \texttt{ equals } G)\big) \texttt{ parmaster } (H \texttt{ starts } O)\end{aligned}$$

where:

- $\mathcal{D}$ is a document composed of two "scenes", that is, two sub-documents $\mathcal{D}_1$ and $\mathcal{D}_2$.
- $\mathcal{D}_1$ is the introduction, composed of four media objects, namely, a video clip A, a sound clip B, a piece of music C and a user button D. The intention is that the video A is played in parallel with its sound B, while at the same time music is heard in the background. The user can stop the music by pressing the button.
- $\mathcal{D}_2$ is the body of the document, composed of five media objects, namely, a still picture E followed by a video clip F and its sound clip G, which determine the presentation of an animation H and a diagram O.
- The duration intervals of the objects are as follows:

$$\begin{aligned}&A : [15, 17]\ B : [14, 16]\ C : [9, 11]\ D : [10, 13]\\ &E : [5, 7] \quad F : [3, 6] \quad G : [4, 7] \quad H : [6, 12] \quad O : [11, \infty)\end{aligned}$$

- $\mathcal{D}'$ `starts` $\mathcal{D}''$ is a macro-notation for $(\mathcal{D}'$ `meets`$R)$ `equals` $\mathcal{D}''$, where R is a "dummy" media object of null content having an arbitrary duration in $(0, \infty)$.

Modeling. The specification $\mathcal{D}$ is modeled as the PND shown in figure 21. For clarity reasons, we have reduced the number of clocks in this example to the least possible. Indeed, since A, B, C, D start simultaneously, their respective clocks have the same value, thus, they can be replaced by the same clock, say, x. Clock x is re-used for E, H, O, while clock y is associated to F, G. Finally, a clock z is associated to the dummy object used for `starts`.

All transitions are delayable and their guards are as follows:

$$\begin{aligned}g_1 &\equiv (x \in I_A) \wedge \mathrm{MAX}\big((x \in I_B), \mathrm{MIN}\big((x \in I_C), (x \in I_D)\big)\big)\\ g_2 &\equiv (x \in I_E)\\ g_3 &\equiv (x \in I_H)\\ g_4 &\equiv \mathrm{MASTER}\big((y \in I_F \wedge y \in I_G), (z > 0 \wedge x \in I_O)\big)\end{aligned}$$

After replacing operators MIN, MAX, MASTER by their definitions and eliminating all existential quantifiers, we obtain:

$$\begin{aligned}g_1 &\equiv 15 \le x \le 16\\ g_2 &\equiv 5 \le x \le 7\\ g_3 &\equiv 6 \le x \le 12\\ g_4 &\equiv 4 \le y \le 6\end{aligned}$$

Therefore, we see that clock z is not needed.

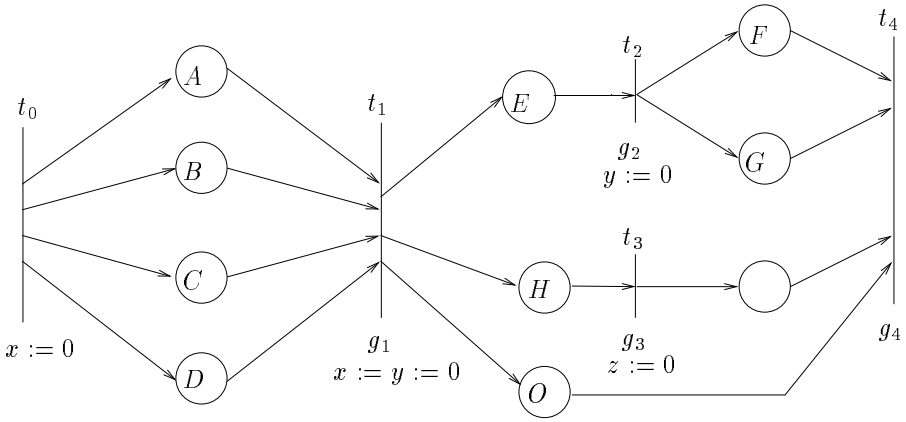


Fig. 21. Example multimedia specification translated into a PND.

Consistency analysis. In order to verify the consistency of the specification, we translate the PND of figure 21 to the TAD shown in figure 22. Then, using

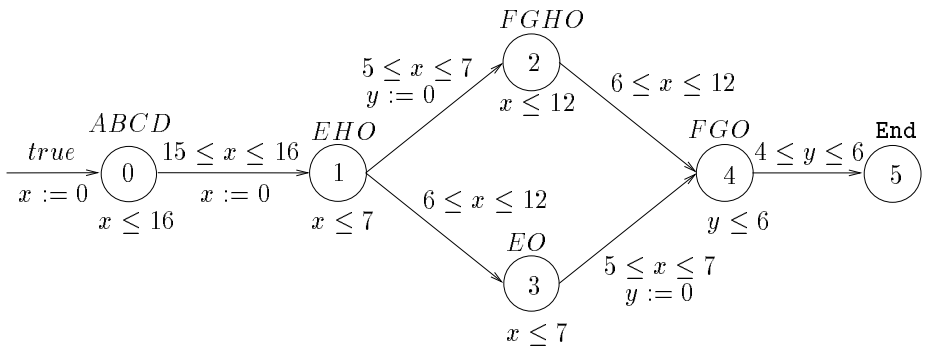


Fig. 22. The TAD corresponding to the PND of figure 21.

KRONOS, we find that the final location `End` is indeed reachable, and we are given the following sample execution scenario, in form of a *symbolic trail*. The latter is made of a sequence of *symbolic states*, that is, pairs of a control location and a clock guard. Each symbolic state is followed by its time successor, which

is in turn followed by an action successor.

$\langle\ 0,\ x = 0$ and $y = 0\ \rangle$
$\langle\ 0,\ 15 \leq x$ and $x \leq 16$ and $x = y\ \rangle$
$15 \leq x$ and $x \leq 16\ \Rightarrow$ end_ABCD; reset$\{x\}$; goto 1
$\langle\ 1,\ x = 0$ and $15 \leq y$ and $y \leq 16\ \rangle$
$\langle\ 1,\ 5 \leq x$ and $x \leq 7$ and $x + 15 \leq y$ and $x \leq y + 16\ \rangle$
$5 \leq x$ and $x \leq 7\ \Rightarrow$ end_E; reset$\{y\}$; goto 2
$\langle\ 2,\ 5 \leq x$ and $x \leq 7$ and $y = 0\ \rangle$
$\langle\ 2,\ 6 \leq x$ and $x \leq 12$ and $y \leq 6$ and $x \leq y + 7$ and $y + 5 \leq x\ \rangle$
$6 \leq x$ and $x \leq 12\ \Rightarrow$ end_H; reset$\{\}$; goto 4
$\langle\ 4,\ 6 \leq x$ and $x \leq 12$ and $y \leq 6$ and $x \leq y + 7$ and $y + 5 \leq x\ \rangle$
$\langle\ 4,\ 4 \leq y$ and $y \leq 6$ and $x \leq y + 7$ and $y + 5 \leq x\ \rangle$
$4 \leq y$ and $y \leq 6\ \Rightarrow$ end_FGO; reset$\{\}$; goto 5
$\langle\ 5,\ 4 \leq y$ and $y \leq 6$ and $x \leq y + 7$ and $y + 5 \leq x\ \rangle$
$\langle\ 5,\ 4 \leq y$ and $x \leq y + 7$ and $y + 5 \leq x\ \rangle$

5 Conclusions

The paper proposes a methodological framework for modeling urgency in timed systems. Urgency is an essential feature of timed systems and is related to their capability of waiting before executing actions. Compared to untimed systems where waiting is asynchronous (indefinite waiting of a process is usually allowed), waiting times are the same in all components of a timed system. Incompatibility of time progress requirements for the processes of a system may lead to inconsistency in specifications.

The main thesis of the paper is that many different ways of composing time progress conditions are useful in practice. Furthermore, time progress condition description should not be dissociated from action description. This leads to the definition of TAD which are timed automata composed of timed transitions, transitions specified in terms of two related conditions expressing respectively, possibility and forcing of execution by stopping time progress. The TAD are a subclass of timed automata that satisfy the time reactivity condition meaning that from any state as long as there are no actions enabled, time can progress.

The proposed methodology is based on the idea that complex timed systems can be obtained as the composition of elementary ones (timed transitions) by means of choice and synchronization operations. The latter allow to define the guard and the deadline of a synchronization action in terms of the guards and deadlines of the synchronizing actions. Apart from AND-synchronization that corresponds to the commonly used conjunctive synchronization, other synchronization modes are shown to be of practical interest as they have been introduced in timed models such as the timed extensions of Petri nets. These synchronization modes can be expressed in terms of AND-synchronization if auxiliary states (and transitions) are added to represent information encoded by modalities in the expression of synchronization guards. However, this may lead to complex

constructions and make specifications less legible. Thus, the different synchronization modes are at least an interesting macro-notation, especially for systems with loosely coupled components where coordination is realized by mechanisms seeking consensus and flexibility e.g., protocols. In fact, the modal formulas in synchronization guards can be considered as the abstract specifications of a protocol used to implement the described coordination.

The paper contributes to clarifying the notion of urgency and proposes the mechanisms that are necessary for a "natural" specification of timed systems. It shows amongst others, that for general timed systems specification a rich methodological framework is necessary that includes new concepts and constructs that are not applicable to untimed systems. In fact, compositional description of untimed specifications can be extended in many different manners to timed specifications, as shown by several examples. It is remarkable that the composition mechanisms defined initially for timed automata or process algebras are obtained by lifting directly the corresponding mechanisms for untimed systems (conjunction of guards and time progress conditions for synchronization) This contrasts with ad hoc flexible synchronization mechanisms added to Petri nets or to logical specification languages. We believe that our results allow to compare and better understand the relations between the existing timed formalisms and can be a basis of a framework for compositional specification of timed systems.

References

[ABRW91] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Deadline-monotonic scheduling. In *Proc. 8th IEEE Workshop on Real-time Operating Systems and Software*, 1991.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[All83] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[BS97a] S. Bornot and J. Sifakis. On the composition of hybrid systems (complete version). In *International NATO Summer School on "Verification of Digital and Hybrid Systems", Antalya, Turkey*, 1997.

[BS97b] S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. In *International Workshop, HART'97*, pages 286–300, Grenoble, France, March 1997. Lecture Notes in Computer Science 1201, Spinger-Verlag.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[JLSIR97] M. Jourdan, N. Layaïda, L. Sabry-Ismail, and C. Roisin. Authoring and presentation environment for interactive multimedia documents. In *Proc. of the 4th Conf. on Multimedia Modelling*, Singapore, November 1997. World Scientific Publishing.

[Mer74] P. Merlin. A study of the recoverability of computer systems. Master's thesis, University of California, Irvine, 1974.

[SDdSS94] P. Sénac, M. Diaz, and P. de Saqui-Sannes. Toward a formal specification of multimedia scenarios. *Annals of telecomunications*, 49(5-6):297–314, 1994.

[Sif77] J. Sifakis. Use of petri nets for performance evaluation. In H. Beilner and E. Gelenebe, editors, *Measuring, modelling and evaluating computer systems*, pages 75–93. North-Holland, 1977.

[SY96] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, pages 347–359, Grenoble, France, February 1996. Lecture Notes in Computer Science 1046, Spinger-Verlag.

Compositional Refinement of Interactive Systems Modelled by Relations*

Manfred Broy

Fakultät für Informatik, Technische Universität München, D-80290 München
e-mail: broy@informatik.tu-muenchen.de

Abstract. We introduce a mathematical model of components that can be used for the description of both hardware and software units forming distributed interactive systems. As part of a distributed system a component interacts with its environment by exchanging messages in a time frame. The interaction is performed by accepting input and by producing output messages on named channels. We describe forms of *composition* and three forms of *refinement*, namely *property refinement*, *glass box refinement*, and *interaction refinement*. Finally, we prove the compositionality of the mathematical model with respect to the introduced refinement relations.

1. Introduction

For a discipline of system development firmly based on a scientific theory we need a clear notion of components and ways to manipulate and to compose them. In this paper, we introduce a mathematical model of a component with the following characteristics:

- A component is *interactive*.
- It is connected with its environments by named and typed *channels*.
- It receives *input messages* from its environment on its *input* channels and generates *output messages* to its environment on its *output* channels.
- A component can be *nondeterministic*. This means that for a given input history there may exist several output histories that the component may produce.
- The interaction between the component and its environment takes place in a *global time* frame.

Throughout this paper we work with discrete time. Discrete time is a sufficient model for most of the typical applications. For an extension of our model to continuous time see [16].

Based on the ideas of an interactive component we can define forms of composition. We basically introduce only one form of composition, *namely parallel composition with feedback*. This form of composition allows us to model *concurrent*

* This work was partially sponsored by the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen", the BMBF-project KORSYS and the industrial research project SysLab sponsored by Siemens Nixdorf and by the DFG under the Leibniz program.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 130-149, 1998.
Springer-Verlag Berlin Heidelberg 1998

execution and *interaction*. We will show that other forms of composition can be introduced as special cases of parallel composition with feedback.

For the systematic stepwise development of components we introduce the concept of *refinement*. By refinement we can develop a given component in a stepwise manner. We study three refinement relations namely *property refinement*, *glass box refinement*, and *interaction refinement*. We claim that these notions of refinement are all what we need for a systematic top down system development.

Finally, we prove that our approach is *compositional*. This means that a refinement step for a composed system is obtained by refinement steps for its components. As a consequence, global reasoning can be structured into local reasoning on the components. Compositionality relates to *modularity* in systems engineering.

The new contribution of this paper the relational version of the stream processing approach as developed at the Technische Universität München (under the keyword FOCUS, see [11], [12]). Moreover, the paper aims at a survey over this approach.

We begin with the informal introduction of the concept of interactive components. This concept is based on communication histories called streams that are introduced in section 3. Then a mathematical notion of a component is introduced in section 4 and illustrated by simple examples. Section 5 treats operators for composing components to distributed systems. In section 6 we introduce three notions of refinements to develop systems and show the compositionality of these notions. All concepts are illustrated by simple examples.

2. Central Notion: Component

We introduce the mathematical notion of a component and on this basis a concept of component specification. A component specification is given by a description of the syntactic interface and a logical formula that relates input and output histories.

The notion of component is essential in systems engineering and software engineering. Especially in software engineering a lot of work is devoted to the concept of *software architecture* and to the idea of *componentware*. Componentware is a catchword in software engineering (see [15]) for a development method where software systems are composed from given components such that main parts of the systems do not have to be reprogrammed every time again but can be obtained by a new configuration of existing software solutions. A key for this approach are well designed *software architectures*. Software architectures mainly can be described as specifically structured systems, composed of components. In both cases a clean and clear concept of a component is needed.

In software engineering literature the following informal definition of a component is found:

> A *component* is a physical encapsulation of related services according to a published specification.

According to this definition we work with the idea of a component which encapsulates a local state or a distributed architecture. We provide a logical way to write a specification of component services. We will relate these notions to glass box views, to the derived black box views, and to component specifications.

A powerful semantic concept of a component interface is an essential ingredient for the following key issues in system development:

- modular program construction,
- software architecture,
- systems engineering.

In the following we introduce a mathematical concept of a component. We show how basic notions of development such as specification and refinement can be based on this concept.

3. Streams

A *stream* is a finite or infinite sequence of messages or of actions. Streams are used to represent communication histories or histories of activities. Let M be a given set of messages.

A stream over the set M is a finite or an infinite an sequence of elements from M.

We use the following notation:

M^*	denotes the finite sequences over M with the *empty* sequence $\diamond$,
M^∞	denotes the infinite sequences over M.

Throughout this paper we do not work with the simple concept of a stream as introduced so far but find it more appropriate to work with so called *timed streams*. A timed stream represents an infinite history of communications over a channel or of activities that are carried out in a discrete time frame. The discrete time frame represents time as an infinite chain of time intervals of equal length. In each time interval a finite number of messages can be communicated or a finite number of actions can be executed. Therefore we model a history of a system model with such a discrete time frame by an infinite sequence of finite sequences of messages or actions. By

$$M^\infty =_{def} (M^*)^\infty$$

we denote the set of timed streams. The k-th sequence in a timed stream represents the sequence of messages exchanged on the channel in the k-th time interval.

A timed stream over M is an infinite sequence of finite sequences of elements from M.

In general, in a system several communication streams occur. Therefore we work with *channels* to identify the individual communication streams. Hence, in our approach, a channel is nothing than an identifier in a system that is related to a stream in every execution of the system.

Throughout this paper we work with some simple forms of notation for streams that are listed in the following. We use the following notation for timed streams x:

$z^\frown x$	concatenation of a sequence z to a stream x,
$x \downarrow i$	sequence of the first i sequences in the stream x,
$S^\ast x$	stream obtained from x by deleting all messages that are not elements of the set S,
$\overline{x}$	finite or infinite stream that is the result of concatenating all sequences in x.

We can also consider timed streams of states to model the traces of state-based system models. In the following, we restrict ourselves to message passing systems, however.

4. Syntactic and Semantic Interfaces of Components

In this section we introduce a mathematical notion of components. We work with typed channels. Let a set S of *sorts* or *types* be given. By

C

we denote the set of typed channels. We assume that we have given a type assignment for the channels:

type: C S

Given a set C of typed channels we now can introduce what we call a *channel valuation* (let M be the set of all messages, by (s) we denote for a type its set of elements):

$\vec{C} = \{x: C \quad M : \quad c \quad C: x.c \quad (type(c)) \}$

A channel valuation x $\vec{C}$ associates a stream of elements of type type(c) with each channel c C.

Given a set of typed input channels I and a set of typed output channels O we introduce the notion of a *syntactic interface* of a component:

(I, O)	syntactic interface,
I	set of typed input channels and,
O	set of typed output channels.

In addition to the syntactic interface we need a concept for describing the *behaviour* of a component. We work with a very simple and straightforward notion of a behaviour. A behaviour is a *relation between input histories and output histories*. Input histories are represented by valuations of the input channels and output histories are represented by the valuations of output channels. To express that a component maps input onto output we do not describe a component by a relation but by a set valued function. Therefore we represent the semantic interface of a component F as follows:

F: $\vec{\ }$ ($\vec{\ }$)

Given x $\vec{\ }$, by F.x we denote the set of all output histories a component with behaviour F may produce on the input x.

Of course, a set valued function, as well known, is isomorphic to a relational definition. We call the function F an *I/O-function*.

Using logical means, such a function can be described by a formula relating input channels with output channels. Syntactically therefore such a formula uses channels as identifiers for streams.

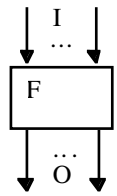


Fig 1 Graphical Representation of a Component F with Input Channels I and Output Channels O

A specification of a component defines:

- its syntactic interface,
- its behaviour by a specifying formula relating input and output channel valuations.

This way we obtain a specification technique that gives us a very powerful way to describe components.

Example. As examples of components we specify a merge component MRG and a fork component FRK as follows:

MRG

in x: T1, y: T2,
out z: T3,
$\overline{x}$ = T1 $\overline{z}$
$\overline{y}$ = T2 $\overline{z}$

Here let T1, T2, T3 be types (in our case we can see types simply as sets) where T1 and T2 are assumed to be disjoint and T3 is the union of T1 and T2.

FRK

in z: T3,
out x´: T1, y´: T2,
$\overline{x}$´ = T1 $\overline{z}$
$\overline{y}$´ = T2 $\overline{z}$

Note that the merge component as specified here is fair. Every input is finally

processed.

We use the following notation for a component F to refer to the constituents of its syntactic interface:

In(F)	the set of input channels I,
Out(F)	the set of output channels O.

I/O-functions can be classified by the following notions. These notions can be either added as properties to specifications explicitly or proved for certain specifications.

An I/O-function F: $\vec{\ }$ ($\vec{O}$) is called

- *properly timed*, if for all time points i $\mathbb{N}$ we have

  x i = z i F(x) i = F(z) i

- *time guarded* (or *causal*), if for all time points i $\mathbb{N}$ we have

  x i = z i F(x) i+1 = F(z) i+1

- *partial*, if F(x) = for some x $\vec{I}$.

- *realisable*, if for a time guarded function f: $\vec{I}$ $\vec{O}$, for all x: f.x F.x.

- *fully realisable*, if for all x: F.x = {f.x: f [[F]]}
  Here [[F]] denotes the set of time guarded functions f: $\vec{I}$ $\vec{O}$, where f.x F.x for all x.

- *time independent* (see [9]), if $\bar{x} = \bar{z}$ $\overline{F.x} = \overline{F.z}$

It is easy to show that both MRG and FRK are time independent. If we add time guardedness as a requirement then both are fully realisable.

We do not require that an I/O-function described by a specification has all the properties introduced above. We are much more liberal. We may add such properties to specifications freely whenever appropriate and therefore deal with all kinds of specifications of I/O-functions that do not have these properties.

A special case of I/O-functions are partial functions which are functions that for certain input histories may have an empty set of output histories. An extreme case is a function that maps every input history onto an empty set. Such functions are not very interesting when used for modelling the requirements for the implementation, since an implementation shows at least one output for each input. However, partial functions may be interesting as intermediate steps in the specification process, since based on these functions we can construct other functions that are more interesting for composition and implementation.

5. Composition Operators

In this section we introduce a notion of *composition* for components. We prefer to introduce a very general form and later define a number of special cases for it.

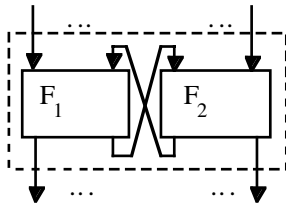


Fig 2 Parallel Composition with Feedback

Given two disjoint sets of channels C_1 and C_2 we define a join operation for the valuations x $\vec{C}_1$, y $\vec{C}_2$ by the following equations:

(x y).c = x.c if c C_1 and
(x y).c = y.c if c C_2

Given I/O-functions with disjoint sets of input channels (where O_1 O_2 =)

$F_1 : \vec{I}_1 \quad (\vec{O}_1), \qquad F_2 : \vec{I}_2 \quad (\vec{O}_2)$

we define the parallel composition with feedback by the I/O-function

$F_1 \quad F_2 : \vec{I} \quad (\vec{O})$

where $I = (I_1 \quad I_2) \setminus (O_1 \quad O_2)$, $O = (O_1 \quad O_2) \setminus (I_1 \quad I_2)$. The resulting function is specified by the equation (here y $\vec{C}$ where $C = I_1 \quad I_2 \quad O_1 \quad O_2$):

$(F_1 \quad F_2).x = \{ y | O : y|I = x|I \quad y|O_1 \quad F_1(y|I_1) \quad y|O_2 \quad F_2(y|I_2) \}$

By x|C we denote the restriction of the valuation x to the channels in C.

For this form of composition we can prove the following facts by rather simple straighforward proofs:

(1) if the F_i are *time guarded* for i = 1, 2, so is $F_1 \quad F_2$,

(2) if the F_i are *realisable* for i = 1, 2, so is $F_1 \quad F_2$,

(3) if the F_i are *fully realisable* for i = 1, 2, so is $F_1 \quad F_2$,

(4) if the F_i are *time independent* for i = 1, 2, so is $F_1 \quad F_2$.

If the F_i are total and properly timed for i = 1, 2, we cannot conclude that $F_1 \quad F_2$ is total. This shows that the composition works only in a modular way for well-chosen

subclasses of specifications.

Further forms of composition that can be defined (we do not give formal definitions for them, since these are quite straightforward):

- feedback without hiding: F
  let F: $\vec{I}$ ($\vec{O}$), then we define: F: $\vec{J}$ ($\vec{O}$) where J = I\O by the equation (here we assume y $\vec{C}$ where C = I O):

  (F).x = {y|O: y|I = x|I y|O F(y|I)}

- parallel composition: $F_1 \| F_2$

  if (I_1 I_2) (O_1 O_2) = we have $F_1 \| F_2 = F_1$ F_2

- logical connectors: F_1 F_2

- hiding: F \ {c}

- renaming of channels: F[c/c']

Given a component specification S we define by S[c/c'] the renaming of the channel c in S to c'. Finally, we also can work with input and output operations on components:

- input transition: F ‹ c:m

- output transition: c:m ‹ F

For a careful treatment of the last three operators see [14]. All the forms of compositions can be defined formally for our concept of components and in principle reduced to parallel composition with feedback.

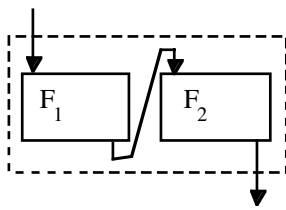


Fig 3 Sequential Composition as a Special Case of Composition

Sequential composition of the components F_1 and F_2 is denoted by

$F_1 ; F_2$

In the special case where $O_1 = I_2 = (O_1 \quad O_2) \quad (I_1 \quad I_2)$ we can reduce sequential composition to parallel composition with feedback along the lines illustrated in Fig. 3 as follows:

$$F_1 ; F_2 = F_1 \quad F_2$$

A simple example of sequential composition (where $O_1 = I_2$) is the composed component MRG;FRK as well as FRK[x/x´, y/y˺];MRG.

6. Refinement - the Basic Concept for System Development

Refinement relations (see [13]) are the key to formalize development steps (see [8]) and the development process. We work with the following basic ideas of refinement:

- *property refinement* - enhancing requirements - allows us to add properties to a specification,
- *glass box refinement* - designing implementations - allows us to decompose a component into a distributed system or to give a state transition description for a component specification,
- *interaction refinement* - relating levels of abstraction - allows us to change the granularity of the interaction, the number and types of the channels of a component (see [10]).

We claim that these notions of refinement are sufficient to describe all the steps needed in the idealistic view of a strict top down hierarchical system development. The three refinement concepts mentioned above are explained in detail in the following.

6.1 Property refinement

Property refinement allows us to replace an I/O-function by one with additional properties. A behaviour

$$F: \vec{I} \quad (\vec{O})$$

is refined by a behaviour

$$\hat{F}: \vec{I} \quad (\vec{O})$$

if

$$\hat{F} \quad F$$

This stands for the proposition

$$x \quad \vec{I}: \hat{F}(x) \quad F(x).$$

A property refinement is a basic refinement step as it is needed in requirements engineering. In the process of requirement engineering, typically the overall services of a system are specified. This, in general, is done by requiring more and more

sophisticated properties for components until a desired behaviour is specified.

Example. A specification of a component that transmits its input on its two input channels to its output channels (but does not necessarily observe the order) is specified as follows.

TM

in x: T1, y: T2,
out x´: T1, y´: T2,
m T1: {m} $\overline{x}$´ ={m} $\overline{x}$
m T2: {m} $\overline{y}$´ ={m} $\overline{y}$

We want to relate this specification to the simple specification of the time independent identity TII that reads as follows:

TII

in x: T1, y: T2,
out x´: T1, y´: T2,
$\overline{x}$´ = $\overline{x}$ $\overline{y}$´ = $\overline{y}$

Given these two specifications we immediately obtain that TII is a property refinement of TM.

TII TIM

This relation is straightforward to prove (see below).

The verification conditions for property refinement are obtained as follows. For given specifications S_1 and S_2 with specifying formulas E_1 and E_2, the specifications S_2 is a property refinement of S_1 if the syntactic interfaces of S_1 and S_2 coincide and if for the formulas E_1 and E_2 we have

$E_1 \quad E_2$

In our example the verification condition is easily obtained and reads as follows:

(m T1: {m} $\overline{x}$´ ={m} $\overline{x}$) $\overline{x}$´ = $\overline{x}$

(m T2: {m} $\overline{y}$´ ={m} $\overline{y}$) $\overline{y}$´ = $\overline{y}$

The proof of this condition is trivial.

Property refinement can also be used to relate composed components to given components. For instance, we obtain the refinement relation.

(MRG ; FRK) TII

Again the proof is quite straightforward.

Property refinement is used in requirements engineering. It is also used in the

design process where decisions are taken that introduce further properties for the components.

6.2 Compositionality of Property Refinement

In our case, the compositionality of property refinement is simple. This is a consequence of the simple definition of composition. The rule of compositional property refinement reads as follows:

$$\frac{\hat{F}_1 \quad F_1 \qquad \hat{F}_2 \quad F_2}{\hat{F}_1 \quad \hat{F}_2 \quad F_1 \quad F_2}$$

The proof of the soundness of this rule is straightforward by the monotonicity of the operarator with respect to set inclusion.

Example. For our example the application of the rule of compositionality reads as follows. Suppose we use a specific component MRG1 for merging two streams. It is defined by

MRG1

in x: T1, y: T2, **out** z: T3,
z = ‹›^f(x, y) **where** f(‹s›^x, ‹t›^y) = ‹s^t›^f(x, y)

Note that this merge component MRG1 is deterministic and time dependent. According to our rule of compositionality and transitivity of refinement, it is sufficient to prove

MRG1 MRG

to conclude

MRG1;FRK MRG;FRK

and by transitivity of the refinement relation

MRG1;FRK TII

This shows how local refinement steps and their proofs are schematically extended to global proofs.

The usage of the composition operator and the relation of property refinement leads to a design calculus for requirements engineering. It includes steps of decomposition and implementation that are treated more systematically in the following section.

6.3 Glass Box Refinement

Glass Box Refinement is a classical concept of refinement that we need and use in the design phase. In the design phase we typically decompose a system with a specified black box behaviour into a distributed system architecture or we represent this behaviour by a state transition machine. By this decomposition we are fixing the basic components of a system.

These components have to be specified and we have to prove that their composition leads to a system with the required functionality. In other words, a glass box refinement is a special case of a property refinement of the form

$F_1 \quad F_2 \quad ... \quad F_n \quad F$ *design of an architecture*

or of the form

$B(\ _0) \quad F$ *implementation by a state machine*

where the I/O-function $B(\ _0)$ is defined by a state machine (see [19]) and $_0$ is its initial state. In the case of the design of an architecture, its components F_1,..., F_n can be hierarchically decomposed into a distributed architecture again, until a granularity of components is obtained which should not be further decomposed into a distributed system but realised by a state machine.

As explained, in a glass box refinement we replace a component by a design which is given by

- a network of components $F_1 \quad F_2 \quad ... \quad F_n$ or
- a state machine $B(\ _0)$ - let be a set of states with an initial state $_0$ and a state transition function

  $: (\quad (I \quad M^*)) \quad (\quad (O \quad M^*))$

  which describes a function

  $B : \quad (\vec{I} \quad (\vec{O}))$

  where we define for each , $z \quad (I \quad M^*)$, $x \quad \vec{I}$ we specify B by the equation

  $B(\ ).(\langle z \rangle \hat{} x) = \{\langle t \rangle \hat{} y: \quad ' \quad : (\ ', t) \quad (\ , z) \quad y \quad B(\ ').x\}$

In our approach iterated glass box refinement leads to a hierarchical, top down refinement method.

It is not in the centre of our paper to describe in detail the design steps leading to distributed systems or to a state machine. Instead, we take a very puristic point of view. Since we have introduced a notion of composition we consider a system architecture as given by a term defining a system by composing a number of components. A state machine is given by a number of transition equations that define

the transitions of the machine.

Accordingly, a glass box refinement is a special case of property refinement where the refinement component has a special syntactic form. In the case of a glass box refinement that transforms a component into a network, this form is a term composed of a number of components.

Example. A very simple instance of such a glass box refinement is already shown by the proposition

MRG FRK TII

It allows us to replace the component TII by two components.

Hence, a glass box refinement works with the relation of property refinement and special terms representing the refined component.

Example. We describe a refinement of the specification TII by a state machine

: (({x, y} T3*)) (({x´, y´} T3*))

where the state space is given by the equation

= T1* T2*

and the state transition relation is specified by

((T1, T2), g) = {((g(x), g(y)), h)}

where h is specified by

h(x´) = T1 and h(y´) = T2

This defines a most trivial state machine implementing TII by buffering its input always exactly one time unit. We obtain a glass box refinement formalised as follows

F ((◇, ◇)) TII

In this case TII is refined into a state machine.

Of course we may also introduce a refinement concept for state machines explicitly in terms of relations between states leading to simulations or bisimulations (see [1], [2], [5], [6], and also [3]). We do not do this here explicitly. We call a relation between state machines with initial states and ´ and transition function and ´ a refinement if

F ·(´) F ( )

The compositionality of glass box refinement is a straightforward consequence of the

compositionality of property refinement.

6.4 Interaction Refinement

Interaction refinement is the refinement notion that we need for modelling development steps between levels of abstraction. Interaction refinement allows us to change

- the number and names of input and output channels,
- the granularity of the messages on the channels

of a component.

An *interaction refinement* requires two functions

A: $\vec{C}'$ ($\vec{C}$) R: $\vec{C}$ ($\vec{C}'$)

that relate the abstract with the concrete level of a development step from one level of abstraction to the next. Given an abstract history x $\vec{C}$ each y R(x) denotes a concrete history representing x. Calculating a representation for a given abstract history and then its abstraction yields the old abstract history again. This is expressed by the requirement:

R ; A = Id

Let Id denote the identity relation. A is called the *abstraction* and R is called the *representation*. R and A are called a *refinement pair*. For untimed components it is sufficient to require for the time independent identity TII (as a generalisation of the specification TII)

R ; A TII

Choosing the component MRG for R and FRK for A immediately gives a refinement pair for untimed components.

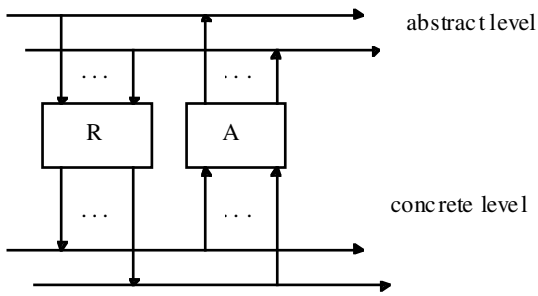


Fig 4 Communication History Refinement

Interaction refinement allows us to refine components, given appropriate refinement pairs for the input and output channels. The idea of a interaction refinement is visualised in Fig 5.

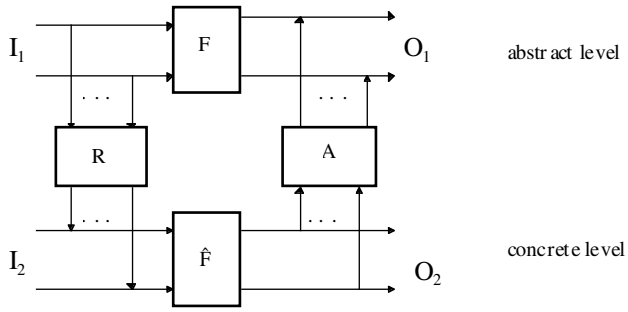


Fig 5 Interface Interaction Refinement (*U-simulation*)

Given interaction refinements

$$: \vec{I}_2 \quad (\vec{I}_1) \qquad\qquad R_I: \vec{I}_1 \quad (\vec{I}_2)$$
$$: \vec{O}_2 \quad (\vec{O}_1) \qquad\qquad R_O: \vec{O}_1 \quad (\vec{O}_2)$$

for the input and output channels we call the I/O-function

$$\hat{F}: \vec{I}_2 \quad (\vec{O}_2)$$

an *interaction refinement* of

$$F: \vec{I}_1 \quad (\vec{O}_1)$$

if one of the following proposition holds:

$\hat{F} \quad A_I ; F ; R_O$	*U^{-1}-simulation*
$R_I ; \hat{F} \quad F ; R_O$	Downward Simulation
$\hat{F} ; A_O \quad A_I ; F$	Upward Simulation
$R_I ; \hat{F} ; A_O \quad F$	*U-simulation*

These are different versions of useful relations between levels of abstractions. A more detailed discussion is found in [13].

Example. Looking at the time independent identity for messages of type T3 we obtain the component specification as follows:

TII3

in z: T3, **out** z´: T3,
$\overline{z} = \overline{z}'$

We obtain

$$MRG\ ;\ TII3\ ;\ FRK[z'/z] \quad TII$$

as a most simple example of interaction refinement by U-simulation. The proof is again straightforward.

6.5 Compositionality of U^{-1}-simulation

We concentrate on U^{-1}-simulation in the following and give the proof of compositionality only for that case. To keep the proof simple we do not give the proof for parallel composition with feedback but give the proof in two steps, first defining the compositionality for parallel composition without any interaction which is a simple straightforward exercise and then give a simplified proof for feedback.

For parallel composition without feedback the rule of compositional refinement reads as follows:

$$\frac{\hat{F}_1 \quad A_I^1\ ;\ F_1\ ;\ R_O^1 \qquad \hat{F}_2 \quad A_I^2\ ;\ F_2\ ;\ R_O^2}{\hat{F}_1 \parallel \hat{F}_2 \quad (A_I^1 \parallel A_I^2);\ (F_1 \parallel F_2)\ ;\ (R_O^1 \parallel R_O^2)}$$

where we require the following syntactic conditions:

$$O_1 \quad O_2 = \qquad \text{and} \quad I_1 \quad I_2 =$$

and analogous conditions for the channels of $\hat{F}_1$ and $\hat{F}_2$. These conditions make sure that there are no name clashes.

The proof of the soundness of this rule is straightforward since it only deals with parallel composition without interaction.

Example. If we replace in a property refinement the component TII3 by a new component TII3´ (for instance along the lines of the property refinement of TII into MRG;FRK) we get by the compositionality of property refinement

$$MRG\ ;\ TII3'\ ;\ FRK[z'/z] \quad TII$$

from the fact that TII3 is an interaction refinement of TII.

It remains to show compositionality of feedback. The general case reads as follows:

$$\frac{\hat{F} \quad (A_I \parallel A)\ ;\ F\ ;\ (R_O \parallel R)}{\hat{F} \quad A_I\ ; \quad F\ ;\ R_O}$$

where we require the syntactic conditions

$$In(A) = In(\hat{F}) \quad Out(\hat{F}),$$

$$In(R) = In(F) \quad Out(F),$$

For independent parallel composition the soundness proof of the compositional refinement rule is straightforward. We give only the proof of the feedback operator and only for the special case where the channels coming from the environment and leading to the environment are empty. This proof easily generalises without any difficulties to the general case. For simplicity, we consider the special case where

$$In(F) = Out(F)$$

In this special case the compositional refinement rule reads as follows:

$$\frac{\hat{F} \quad A ; F ; R}{\hat{F} \quad A ; \quad F ; R}$$

The proof of the soundness of this rule is shown as follows. Here we use the classical relational notation:

xFy

that stands for y F(x).

Proof. Soundness for the Rule of U^{-1}-Simulation

If we have:	$\hat{z} \quad \hat{F}$
then	$\hat{z}\ \hat{F}\ \hat{z}$
and by the hypothesis:	x, y: $\hat{z}$ Ax xFy yR $\hat{z}$
then by:	xRz zAy x = y
we obtain:	x, y: $\hat{z}$ Ax xFy yR $\hat{z}$ x = y
and thus:	x: $\hat{z}$ Ax xFx xR $\hat{z}$
and finally	$\hat{z}$ A; F ; R

The simplicity of the proof of our result comes from the fact that we have chosen such a straightforward model of component. In our model, in particular, input and output histories are represented explicitly. This allows us to apply classical ideas (see [17], [18]) of data refinement to communication histories. Roughly speaking: communication histories are nothing than data structures that can be manipulated and refined like other data structures.

Example. To demonstrate interaction refinement let us consider the specification of two delay components.

D3

in c, z: T3, **out** c´, z´: T3,
c´ = ‹›^z z´ = ‹›^c

D

in	x, c: T1, y, d: T2,
out	x´, c´: T1, y´, d´:T2,
c´ = ‹›^x, x´ = ‹›^c	
d´ = ‹›^y, y´ = ‹›^d	

We have

MRG MRG[c/x, d/y, c/z] ; D3 ; FRK FRK[c´/x, d´/y, c´/z] D

and in addition

D3[c/c´] TII3, D[c/c´, d/d´] TII

and so finally we obtain

MRG ; D3[c/c´] ; FRK D[c/c´, d/d´] TII

which is an instance of the compositionality rule for interaction refinement.

Our refinement calculus leads to a logical calculus for "programming in the large" where we can argue about software architectures.

7. Conclusions

What we have presented in the previous chapters is a comprehensive method for a system and software development which supports all steps of a hierarchical stepwise refinement development method. It is compositional and therefore supports all the modularity requirements that are generally needed.

What we have presented is a method that provides, in particular, the following ingredients:

- a proper notion of a syntactic and semantic interface of a component,
- a formal specification notation and method,
- a proper notion of composition,
- a proper notion of refinement and development,
- a compositional development method,
- a flexible concept of software architecture,
- concepts of time and the refinement of time (see [16]).

What we did not mention throughout the paper are concepts that are also available and helpful from a more practical point of view including

- combination with tables and diagrams,
- tool support in the form of AutoFocus (see [4]).

The simplicity of our results is a direct consequence of the specific choice of our semantic model. The introduction of time makes the model robust and expressive. The

fact that communication histories are explicit allows us to avoid all kinds of complications like prophecies or stuttering and leads to an abstract relational view of systems.

Of course, what we have presented is just the scientific kernel of the method. More pragmatic ways to describe specifications are needed. These more pragmatic specifications can be found in the work done in the SysLab-Project (see [7]) at the Technical University of Munich. For extensive explanations of the use of state transition diagrams, data flow diagrams and message sequence charts as well as several versions of data structure diagrams we refer to this work.

Acknowledgement

It is a pleasure to thank Max Breitling for a number of comments and helpful suggestions for improvement.

References

1. M. Abadi, L. Lamport: The Existence of Refinement Mappings. Digital Systems Research Center, SRC Report 29, August 1988
2. M. Abadi, L. Lamport: Composing Specifications. Digital Systems Research Center, SRC Report 66, October 1990
3. L. Aceto, M. Hennessy: Adding Action Refinement to a Finite Process Algebra. Proc. ICALP 91, Lecture Notes in Computer Science 510, (1991), 506-519
4. F. Huber, B. Schätz, G. Einert: Consistent Graphical Specification of Distributed Systems. In: J. Fitzgerald, C. B. Jones, P. Lucas (ed.): FME '97: 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, 1997, 122-141
5. R.J.R. Back: Refinement Calculus, Part I: Sequential Nondeterministic Programs. REX Workshop. In: J. W. deBakker, W.-P. deRoever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, 42-66
6. R.J.R. Back: Refinement Calculus, Part II: Parallel and Reactive Programs. REX Workshop. In: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, 67-93
7. R. Breu, R. Grosu, Franz Huber, B. Rumpe, W. Schwerin: Towards a Precise Semantics for Object-Oriented Modeling Techniques. In: H. Kilov, B. Rumpe (eds.): Proceedings ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques, 1997, Also: Technische Universität München, Institut für Informatik, TUM-I9725, 1997
8. M. Broy, B. Möller, P. Pepper, M. Wirsing: Algebraic Implementations Preserve Program Correctness. Science of Computer Programming 8 (1986), 1-19
9. M. Broy: Functional Specification of Time Sensitive Communicating Systems. REX Workshop. In: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds): Stepwise Refinement of Distributed Systems. Lecture Notes in Computer Science 430, 153-179

10. M. Broy: Compositional Refinement of Interactive Systems. Digital Systems Research Center, SRC Report 89, July 1992, To appear in JACM
11. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, R. Weber: The Design of Distributed Systems - An Introduction to Focus. Technische Universität München, Institut für Informatik, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9202, January 1992
12. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, R. Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems. Technische Universität München, Institut für Informatik, Sonderforschungsbereich 342: Methoden und Werkzeuge für die Nutzung paralleler Architekturen TUM-I9203, January 1992
13. M. Broy: Interaction Refinement – The Easy Way. In: M. Broy (ed.): Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118, 1993
14. M. Broy: Algebraic Specification of Reactive Systems. M. Nivat, M. Wirsing (eds): Algebraic Methodology and Software Technology. 5th International Conference, AMAST '96, Lecture Notes of Computer Science 1101, Heidelberg: Springer 1996, 487-503
15. M. Broy: Towards a Mathematical Concept of a Component and its Use. First Components' User Conference, Munich 1996. Revised version in: Software-Concepts and Tools 18, 1997, 137-148
16. M. Broy: Refinement of Time. M. Bertran, Th. Rus (eds.): Transformation-Based Reactive System Development. ARTS'97, Mallorca 1997. Lecture Notes in Computer Science 1231, 1997, 44-63, To appear in TCS
17. J. Coenen, W.P. deRoever, J. Zwiers: Assertional Data Reification Proofs: Survey and Perspective. Christian-Albrechts-Universität Kiel, Institut für Informatik und praktische Mathematik, Bericht Nr. 9106, Februar 1991.
18. C.A.R. Hoare: Proofs of Correctness of Data Representations. Acta Informatica 1, 1972, 271-281
19. N. Lynch, E. Stark: A Proof of the Kahn Principle for Input/Output Automata. Information and Computation 82, 1989, 81-92

Toward Parametric Verification of Open Distributed Systems*

Mads Dam, Lars-åke Fredlund, and Dilian Gurov

Swedish Institute of Computer Science**

Abstract. A logic and proof system is introduced for specifying and proving properties of open distributed systems. Key problems that are addressed include the verification of process networks with a changing interconnection structure, and where new processes can be continuously spawned. To demonstrate the results in a realistic setting we consider a core fragment of the Erlang programming language. Roughly this amounts to a first-order actor language with data types, buffered asynchronous communication, and dynamic process spawning. Our aim is to verify quite general properties of programs in this fragment. The specification logic extends the first-order μ-calculus with Erlang-specific primitives. For verification we use an approach which combines local model checking with facilities for compositional verification. We give a specification and verification example based on a billing agent which controls and charges for user access to a given resource.

1 Introduction

A central feature of open distributed systems as opposed to concurrent systems in general is their reliance on modularity. Open distributed systems must accommodate addition of new components, modification of interconnection structure, and replacement of existing components without affecting overall system behaviour adversely. To this effect it is important that component interfaces are clearly defined, and that systems can be dynamically put together relying only on component behaviour along these interfaces. That is, behaviour specification of open distributed systems, and hence also their verification, cannot be based on a fixed systems structure but needs to be parametric on the behaviour of components. Almost all prevailing approaches to verification of such systems rely on an assumption that process networks are static, or can safely be approximated as such, as this assumption opens up for the possibility of bounding the space of global system states. Clearly such assumptions square poorly with the dynamic and parametric nature of open distributed systems.

* Work partially supported by the Computer Science Laboratory of Ericsson Telecom AB, Stockholm, the Swedish National Board for Technical and Industrial Development (NUTEK) through the ASTEC competence centre, and a Swedish Foundation for Strategic Research Junior Individual Grant.

** Address: SICS, Box 1263, S-164 28 Kista, Sweden. Email: {mfd,fred,dilian}@sics.se.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 150-185, 1998.
Springer-Verlag Berlin Heidelberg 1998

Core Erlang Our aim in this paper is to demonstrate an approach to system specification and verification that has the potentiality of addressing open distributed systems in general. We study the issue in terms of a core fragment of Ericsson's Erlang programming language [AVWW96] which we call Core Erlang. Core Erlang is essentially a first-order actor language (cf. [AMST97]). The language has primitives for local computation: data types, first-order abstraction and pattern matching, and sequential composition. In addition to this Core Erlang has a collection of primitives for component (process) coordination: sending and receiving values between named components by means of ordered message queues, and for dynamically creating new components.

Specification Language We use a temporal logic based on a first-order extension of the modal μ-calculus for the specification of component behaviour. In this logic it is possible to describe a wide range of important system properties, ranging from type-like assertions to complex interdependent safety and liveness properties. The development of this logic is actually fairly uncontroversial: To adequately describe component behaviour it is certainly needed to express potentialities of actions across interfaces and the necessary and contingent effects of these actions, to express properties of data types and properties of components depending on values, to access component names, and to express properties of messages in transit.

Challenges The real challenge is to develop techniques that allow such temporal properties to be verified in a parametric fashion in face of the following basic difficulties:

1. Components can dynamically create other components.
2. Component names can be bound dynamically, thus dynamically changing component interconnection structure (similar to the case of the π-calculus [MPW92]).
3. Components are connected through unbounded message queues.
4. Through use of non-tail recursion components can give rise to local state spaces of unbounded size.
5. Basic data types such as natural numbers and lists are also unbounded.

We would expect some sort of uniformity in the answers to these difficulties. For instance, techniques for handling dynamic process creation are likely to be adaptable to non-tail recursive constructions quite generally, and similarly message queues is just another unbounded data type.

Approach In [Dam98] an answer to the question of dynamic process creation was suggested, cast in terms of CCS. Instead of closed correctness assertions of the shape $s : \phi$ (s is a system, ϕ its specification) which are the typical objects of state exploration based techniques, the paper considered more general *open correctness assertions* of the shape $\Gamma \vdash s : \phi$ where Γ expresses assumptions $S : \psi$ on components S of s. Thus the behaviour of s is specified parametrically upon the behaviour of its component S. To address verification, a sound and

weakly complete proof system was presented, consisting of proof rules to reduce complex proof goals to (hopefully) simpler ones, including a process cut rule by which properties of composite processes can be proved in terms of properties of its constituent parts. The key, however, is a loop detection mechanism, namely a *rule of discharge*, that can under certain circumstances be applied to discharge proof goals that are instances of proof goals that have already been encountered during proof construction. The combination of the rule of discharge with the process cut rule provides the power required to deal successfully with features such as dynamic process creation.

Our contribution in the present paper is to show how the approach of [Dam98] can be extended to address the difficulties enumerated above for a fragment of a real programming language, and to show the utility of our approach on a concrete example exhibiting some of those difficulties. In particular we have resolved a number of shortcomings of [Dam98] with respect to the rule of discharge.

Example We use a running example based on the following scenario: A user wants to access a resource paying for this using a given account. She therefore issues a request to a resource manager which responds by dynamically creating a billing agent process to act as an intermediary between the user, the resource, and the user's account. We view this scenario as quite typical of many security-critical mobile agent applications.

The user is clearly taking a risk by exposing her account to the resource manager and the billing agent. One of these parties might violate the trust put in him eg. by charging for services not provided, or by passing information to third parties that should be kept confidential. Equally the resource manager need to trust the billing agent (and to some minor extent the user). We show how the system can be represented in Core Erlang, how some critical properties can be expressed, and outline a proof of the desirable property of the billing agent that the number of transfers from the user account does not exceed the number of requests to use the resource.

Organisation The paper is organised as follows. Section 2 introduces the fragment of Erlang treated in the paper and presents an operational semantics for the language. The following section focuses on the variant of the μ-calculus used as the specification logic, providing examples as well as a formal semantics. Section 4 describes the local part of a proof system for verifying that an Erlang system satisfies a specification formalised in the μ-calculus, and contains proofs of soundness for some proof rules introduced in the section. The rule of discharge is motivated in terms of two simple examples in section 5, and it is formally stated and proved sound in section 6. In section 7 we put the proof system to work on the billing example, outlining parts of a correctness proof. Finally, the paper ends with a discussion in section 8 on directions for further work, and some concluding remarks.

2 Core Erlang

We introduce a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as natural numbers, lists, tuples, or process identifiers (pid's), using asynchronous, first-order call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as communication guards, exception handling, modules, distribution extensions, and a host of built-in functions.

Processes A Core Erlang system consists of a number of processes computing in parallel. Each process is named by a unique process identifier *pid* of which we assume an infinite supply. Pid's are created along with the processes themselves. Associated with a process is an Erlang *expression* e, i.e. the expression being evaluated, and a mailbox, or input *queue* q, assumed to be of unbounded capacity. Messages are sent by addressing a data value to a receiving process, identified through its pid.

Definition 1 (Processes, System States). *An Erlang* process *is a triple* $\langle e, pid, q\rangle$*, where* e *is an Erlang expression,* pid *is a process identifier, and* q *is a message queue. An Erlang* system state s *is a set of processes such that* $\langle e, pid, q\rangle, \langle e', pid', q'\rangle \in s$ *and* $\langle e, pid, q\rangle \neq \langle e', pid', q'\rangle$ *implies* $pid \neq pid'$*.* $\mathcal{S}$ *is the set of system states.*

We normally write system states using the grammar:

$$s ::= \langle e, pid, q\rangle \mid s \parallel s$$

understanding $\parallel$ as set union.

As the amount of different syntactical categories involved in the operational semantics and in the specification logic is quite large, the following notational convention is useful.

Convention 2. Corresponding small and capital letters are used to range over values, resp. variables over a given syntactical domain.

Thus, as e.g. e is used to range over Erlang expressions, E is used to range over variables taking Erlang expressions as values.

Erlang Expressions Besides expressions we operate with the syntactical categories of *matches* m, *patterns* p, and *values* v. The abstract syntax of Core Erlang expressions is summarised as follows:

$$\begin{aligned}
e ::= {} & V \mid \texttt{self} \mid op(e_1, \ldots, e_n) \mid \\
& e_1\ e_2 \mid e_1, e_2 \mid \texttt{case}\ e\ \texttt{of}\ m \mid \texttt{spawn}(e_1, e_2) \mid \\
& \texttt{receive}\ m\ \texttt{end} \mid e_1 ! e_2 \\
m ::= {} & p_1 \to e_1; \cdots; p_n \to e_n \\
p ::= {} & op(p_1, \ldots, p_n) \mid V \\
v ::= {} & op(v_1, \ldots, v_n)
\end{aligned}$$

Here *op* ranges over a set of primitive constants and operations including zero 0, successor $e+1$, tupling $\{e_1, e_2\}$, the empty list $[\,]$, list prefix $[e_1|e_2]$, pid constants ranged over by *pid*, and atom constants ranged over by a, f, and g. In addition we need constants and operations for message queues: A queue is a sequence of values $q = v_1 \cdot v_2 \cdot \ldots \cdot v_n$ where ϵ is the empty queue and $q_1 \cdot q_2$ is queue concatenation, assumed to be associative.

Atoms are used to name functions. We reserve f and g for this purpose. Expressions are interpreted relative to an environment of function definitions $f(V_1, \ldots, V_n) \to e$, syntactic sugar for $f \stackrel{\Delta}{=} \{V_1, \ldots, V_n\} \to e$. Each function atom f is assumed to be defined at most once in this environment.

Intuitive Semantics The intuitive meaning of the Erlang operators, given in the context of a pid *pid* and a queue q, should be not too surprising:

- `self` evaluates to the pid *pid* of the process.
- *op* is a data type constructor: To evaluate $op(e_1, ..., e_n)$, e_1 to e_n are evaluated in left-to-right order.
- $e_1\ e_2$ is application: First e_1 is evaluated to a function atom[1] f, then e_2 is evaluated to a value v, and finally the function definition of f is looked up and matched to v.
- e_1, e_2 is sequential composition: First e_1 is evaluated (for its side-effect only), and then evaluation proceeds with e_2 whose value is actually returned.
- `case` e `of` m is evaluated by first evaluating e to a value v, then matching v using m. If several patterns in m match, the first one is chosen. Matching a pattern p_i of m against v can cause unbound variables to become bound in e_i[2]. In a function definition all free variables are considered as unbound.
- `spawn`(e_1, e_2) is the language construct for creating new processes. First e_1 is evaluated to a function atom f, then e_2 to a value v, a new pid pid' is generated, and a process $\langle (f\ v), pid', \epsilon \rangle$ with that pid and an initially empty queue is spawned evaluating $f\ v$. The value of the spawn expression itself is the pid pid' of the newly spawned process.
- `receive` m `end` inspects the process mailbox q and retrieves (and removes) the first element in q that matches any pattern of m. Once such an element v has been found, evaluation proceeds analogously to `case` v `of` m.
- $e_1!e_2$ is sending: e_1 is evaluated to a pid pid', then e_2 to a value v, then v is sent to pid', resulting in v as the value of the send expression.

Example: Billing Agents In the introduction we gave a scenario for accessing private resources. As an example Core Erlang program, a function for managing such accesses (a resource manager) is shown below. Erlang variables are upper-case, while atoms are lower-case. Atoms are used to name functions (*rm*, *lookup* and *billagent*), but also as constant values for identifying the "type" of a particular message (*contract*, *contract_ok*, etc.), or for other synchronisation purposes (*lookup_ok* and *lookup_nok*).

[1] There is no construct for lambda-abstraction in Erlang.

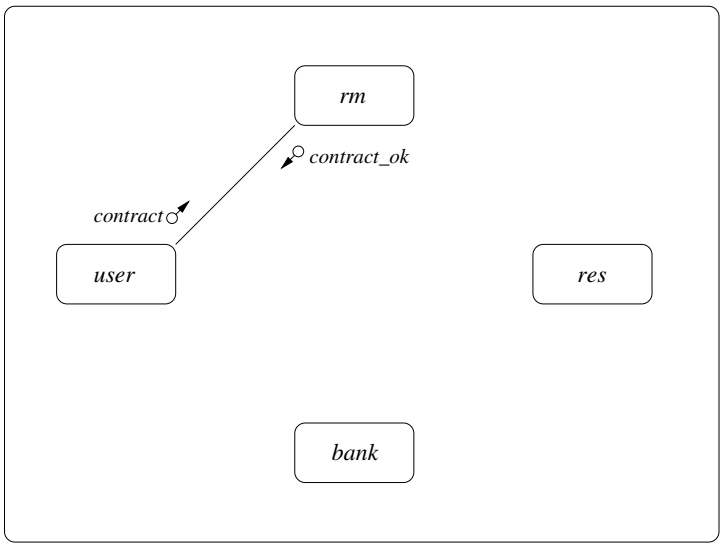
[2] This is not quite the binding the convention of Erlang proper: There the first occurrence of V in (case e_1 of $V \to e_2$), V can bind the second.

```
rm(ResList,BankPid,RAcc) →
  receive
    {contract,{Pu, UAcc}, UserPid} →
      case lookup(Pu, ResList)) of
        {lookup_ok, Pr} →
          UserPid!{contract_ok,spawn(billagent,(Pr, BankPid, RAcc, UAcc))};
        lookup_nok →
          UserPid!contract_nok
      end
  end,
  rm(ResList, BankPid, RAcc).
```

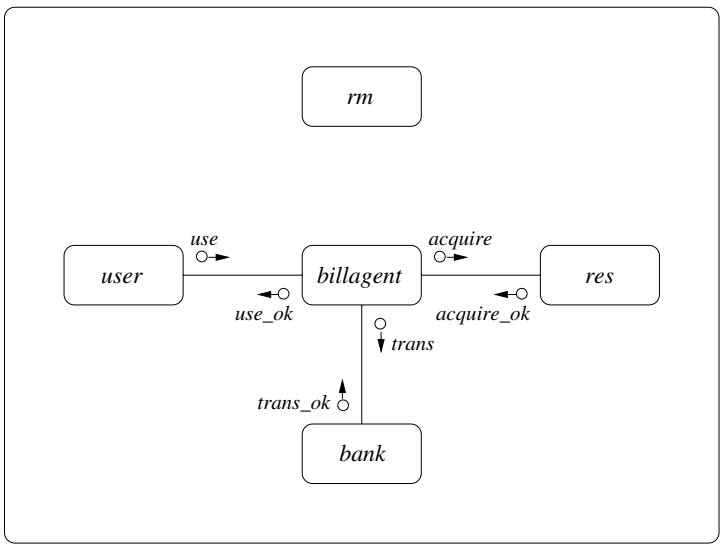
The resource manager *rm* accepts as arguments a resource list, the pid of a trusted bank agent, and a private account. The resource list uses pairs to implement a map from public to private resource "names"; given a public name Pu, a function $lookup(Pu, ResList)$ is used to extract the corresponding private name Pr. The resource manager, after receiving a contract offer (identifying the paying account $UAcc$), tries to obtain the private name of the requested resource, and if this succeeds, a billing agent is spawned to mediate between the user, the bank and the resource, and the name (i.e. pid) of the billing agent is made known to the user. Figure 1 shows the system configuration before and after the creation of the billing agent.

```
billagent(ResPid, BankPid, RAcc, UAcc) →
  receive
    {use, UserPid} →
      Res!{acquire,self},
      receive
        {acquire_ok, Value} →
          BankPid!{trans,{UAcc, RAcc},self},
          receive
            {trans_ok,{UAcc, RAcc}} → UserPid!{use_ok, Value};
            {trans_nok,{UAcc, RAcc}} → UserPid!use_nok
          end;
        acquire_nok → UserPid!use_nok
      end
  end,
  billagent(ResPid, BankPid, RAcc, UAcc).
```

The billing agent coordinates access to the resource with withdrawals from the account. Upon receiving a request for the resource $\{use, UserPid\}$, it attempts to acquire the resource, and if this succeeds (resulting in a response $Value$ being received from the resource), it attempts to transfer money from the user account to the resource manager account, and then sends the value to the user.



(a)



(b)

Fig. 1. The system configuration: (a) before, and (b) after spawning the billing agent

Matching The definition of the operational semantics requires an ancillary, entirely standard, definition to be made concerning pattern matching between a value v and a pattern p. A *term environment* is a partial function τ of variables V to values v. The totally undefined term environment is $[\,]$. The *most general unifier*, $mgu(v, p, \tau)$, of v and p in term environment τ is a term environment defined as follows:

$$\begin{aligned} mgu(v, V, \tau) &\triangleq \tau[V \mapsto v] \text{ if } V \notin dom(\tau) \\ mgu(v, V, \tau) &\triangleq \tau \qquad \text{if } V \in dom(\tau) \text{ and } \tau(V) = v \\ mgu(op(v_1, ..., v_n), op(p_1, ..., p_n), \tau) & \\ &\triangleq mgu(v_1, p_1, mgu(v_2, p_2, ..., mgu(v_n, p_n, \tau)...)) \end{aligned}$$

The equations should be understood as Kleene equations: One side of the equation is defined if and only if the other side is, and if they are defined, the two sides are equal. It is not hard to show that the definition is independent of order of unification in the third clause of this definition. Term environments are lifted from functions on variables to functions on arbitrary terms in the usual way. $\tau(e)$ is τ applied to the term e. Now define the relation v *matches* p to hold just in case $mgu(v, p, [\,])$ is defined. If $\{V_1, \ldots, V_n\}$ are the free variables in p then v *matches* p can be expressed as the first-order formula $\exists V_1, \ldots, V_n . p = v$.

Given a value v, a queue $q = v_1 \cdot v_2 \cdot \ldots \cdot v_m$ and a match $m = p_1 \rightarrow e_1 \,;\, \cdots \,;\, p_n \rightarrow e_n$, we define two matching operations to be used below:

$$\begin{aligned} &\textit{val-match}(v, m) \triangleq mgu(v, p_i, [\,])(e_i) \\ &\qquad \text{if } v \textit{ matches } p_i \text{ and } \forall j < i.\ \textit{not}(v \textit{ matches } p_j) \\ &\textit{queue-match}(q, m) \triangleq mgu(v_m, p_i, [\,])(e_i) \\ &\qquad \text{if } v_m \textit{ matches } p_i \text{ and } \forall j < m.\ \forall k.\ \textit{not}(v_j \textit{ matches } p_k) \\ &\qquad \text{and } \forall k < i.\ \textit{not}(v_m \textit{ matches } p_k). \end{aligned}$$

Operational Semantics A *reduction context* $r[\cdot]$ is an Erlang expression with a "hole" in it. The reduction context defines the evaluation order of subexpressions in language constructs; here from left to right[3]. The result of placing e in (the hole of) a context $r[\cdot]$ is denoted $r[e]$. Contexts are defined by the grammar:

$$\begin{aligned} r[\cdot] ::= \quad & \cdot \\ \mid\ & op(v_1, \ldots, v_{i-1}, r[\cdot], e_{i+1}, \ldots, e_n) \qquad \text{for all } i : 1 \leq i \leq n \\ \mid\ & r[\cdot]\,,\, e \\ \mid\ & r[\cdot]\ e \mid f\ r[\cdot] \\ \mid\ & \texttt{case}\ r[\cdot]\ \texttt{of}\ m\ \texttt{end} \\ \mid\ & r[\cdot]!e \mid pid!r[\cdot] \\ \mid\ & \texttt{spawn}(r[\cdot], e) \mid \texttt{spawn}(f, r[\cdot]) \end{aligned}$$

[3] An arbitrary simplification. The full Erlang language does not specify any evaluation ordering.

Definition 3 (Operational semantics). *The operational semantics of Core Erlang is defined by table 1.*

The rules defining the operational semantics of Core Erlang are separated into three classes: the *local rules* concern classical, side-effect free, computation steps of an Erlang expression, the *process rules* define the actions of a single process (an Erlang expression with an associated pid and queue), and the *system rules* give the semantics of the parallel composition operator. In the rules "*pid* fresh" requires *pid* to be a new pid, and $foreign(pid)(s)$ states that no process in the system state s has *pid* as its pid.

3 The Property Specification Logic

In this section we introduce a specification logic for Core Erlang. The logic is based on a first-order μ-calculus, corresponding, roughly, to Park's μ-calculus [Par76], extended with Erlang-specific features. Thus the logic is based on the first-order language of equality, extended with modalities reflecting the transition capabilities of processes and process configurations, least and greatest fixpoints, along with a few additional primitives.

Syntax Abstract formula syntax is determined by the following grammar where Z ranges over predicate variables parametrised by value vectors.

$$
\begin{aligned}
\phi ::= {} & p = p \mid p \neq p \mid \mathtt{term}(p) = p \mid \mathtt{queue}(p) = p \mid \\
& \mathtt{local}(p) \mid \mathtt{foreign}(p) \mid \mathtt{atom}(p) \mid \mathtt{unevaluated}(p) \mid \\
& \phi \wedge \phi \mid \phi \vee \phi \mid \forall V.\phi \mid \exists V.\phi \mid \\
& <>\phi \mid <p?p>\phi \mid <p!p>\phi \mid [\,]\phi \mid [p?p]\phi \mid [p!p]\phi \mid \\
& Z(p_1, \ldots, p_n) \mid \\
& (\mu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n) \mid (\nu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n)
\end{aligned}
$$

It is important to note that patterns p and value variables V in the case of formulas range not only over Erlang values, but also over message queues.

Intuitive Semantics The intuitive meaning of formulas is given as follows:

- Equality, inequality, the Boolean connectives and the quantifiers take their usual meanings.
- The purposes of $\mathtt{term}(p_1) = p_2$ and $\mathtt{queue}(p_1) = p_2$ are to pick up the values of terms and queues associated with a given pid p_1. $\mathtt{term}(p_1) = p_2$ requires p_1 to equal the pid of a process which is part of the system state being predicated, and the Erlang expression associated with that pid to be identical to p_2. Similarly $\mathtt{queue}(p_1) = p_2$ holds if the queue associated with p_1 is equal to p_2.
- $\mathtt{atom}(p)$ holds if p is equal to an atom.

Local rules

SEQ : $v\,,\, e \longrightarrow e$

CASE : $\texttt{case}\ v\ \texttt{of}\ m\ \texttt{end} \longrightarrow \mathit{val\text{-}match}(v, m)$ if $\mathit{val\text{-}match}(v, m)$ is defined.

FUN : $f\ v \longrightarrow \texttt{case}\ v\ \texttt{of}\ m\ \texttt{end}$ if $f \stackrel{\Delta}{=} m$.

Process rules

LOCAL : $\langle r[e], \mathit{pid}, q\rangle \longrightarrow \langle r[e'], \mathit{pid}, q\rangle$ if $e \longrightarrow e'$.

SELF : $\langle r[\texttt{self}], \mathit{pid}, q\rangle \longrightarrow \langle r[\mathit{pid}], \mathit{pid}, q\rangle$

SPAWN : $\langle r[\texttt{spawn}(f, v)], \mathit{pid}, q\rangle \longrightarrow \langle r[\mathit{pid}'], \mathit{pid}, q\rangle \parallel \langle f\ v, \mathit{pid}', \epsilon\rangle$ if pid' fresh.

SEND : $\langle r[\mathit{pid}'!v], \mathit{pid}, q\rangle \xrightarrow{\mathit{pid}'!v} \langle r[v], \mathit{pid}, q\rangle$ if $\mathit{pid} \neq \mathit{pid}'$.

SELF–SEND : $\langle r[\mathit{pid}!v], \mathit{pid}, q\rangle \longrightarrow \langle r[v], \mathit{pid}, q \cdot v\rangle$

RECEIVE : $\langle r[\texttt{receive}\ m\ \texttt{end}], \mathit{pid}, q' \cdot v \cdot q''\rangle \longrightarrow \langle r[\mathit{queue\text{-}match}(q' \cdot v, m)], \mathit{pid}, q' \cdot q''\rangle$
if $\mathit{queue\text{-}match}(q' \cdot v, m)$ is defined.

INPUT : $\langle e, \mathit{pid}, q\rangle \xrightarrow{\mathit{pid}?v} \langle e, \mathit{pid}, q \cdot v\rangle$ for all values v.

System rules

COM : $s_1 \parallel s_2 \longrightarrow s_1' \parallel s_2'$ if $s_1 \xrightarrow{\mathit{pid}!v} s_1'$ and $s_2 \xrightarrow{\mathit{pid}?v} s_2'$.

INTERLEAVE1 : $s_1 \parallel s_2 \longrightarrow s_1' \parallel s_2$ if $s_1 \longrightarrow s_1'$.

INTERLEAVE2 : $s_1 \parallel s_2 \xrightarrow{\mathit{pid}?v} s_1' \parallel s_2$ if $s_1 \xrightarrow{\mathit{pid}?v} s_1'$.

INTERLEAVE3 : $s_1 \parallel s_2 \xrightarrow{\mathit{pid}!v} s_1' \parallel s_2$ if $s_1 \xrightarrow{\mathit{pid}!v} s_1'$ and $\mathit{foreign}(\mathit{pid})(s_2)$.

Table 1. Operational semantics of Core Erlang

- `local`(p) holds if p is equal to the pid of a process in the system state being predicated, and analogously `foreign`(p) holds if p is equal to a pid and there is no process with pid p in the predicated system state.
- `unevaluated`(p) holds if `local`(p) does and the Erlang expression associated with p is not a ground value.
- $<>\phi$ holds if an internal transition is enabled to a system state satisfying ϕ. $[\,]\phi$ is the dual of $<>\phi$ (i.e. all states following an internal transition satisfy ϕ). $<p_1!p_2>\phi$ holds if an output transition with appropriate parameters is enabled to a state satisfying ϕ, and $<p_1?p_2>\phi$ is used similarly for input transitions.
- $\mu Z(V_1,\ldots,V_n).\phi$ is the least inclusive predicate Z satisfying the equation $\phi = Z(V_1,\ldots,V_n)$, while $\nu Z(V_1,\ldots,V_n).\phi$ is the most inclusive such predicate.

As is by now well known, monotone recursive definitions in a complete Boolean lattice have least and greatest solutions. This is what motivates the existence of predicates Z above. Greatest solutions are used, typically, for safety (i.e. invariant) properties, while least solutions are used for liveness (i.e. eventuality) properties. For readability we often prefer the notation $f(V_1,\ldots,V_n) \Leftarrow \phi$ for defining explicitly a formula atom f which is to be applied to pattern vectors $(p_1,\ldots,p_n)$, instead of the standard notation for least fixpoints, and similarly $f(V_1,\ldots,V_n) \Rightarrow \phi$ for greatest ones. In such definitions all value variables occurring freely in ϕ have to be in $V_1,\ldots,V_n$.

We use standard abbreviations like *true*, *false* and $\forall V_1,\ldots,V_n.\phi$. It is possible to define a de Morganised negation `not`, by the standard clauses along with clauses for the Erlang specific atomic propositions `term`$(p_1) = p_2$, `queue`$(p_1) = p_2$, `local`(p), `foreign`(p), or `unevaluated`(p). This is an easy exercise given the formal semantics in table 2 below.

Definition 4 (Boolean Formula). *A formula ϕ is* boolean *if it has no occurrences of modal operators, neither occurrences of any of the Erlang-specific atomic propositions* `term`$(p_1) = p_2$, `queue`$(p_1) = p_2$, `local`(p), `foreign`(p), *or* `unevaluated`(p).

Boolean formulas are important as they either hold globally or not at all.

The formal semantics of formulas is given as a set $\parallel \phi \parallel \eta \subseteq \mathcal{S}$, where η is a *valuation* providing interpretations for variables (value variables, or predicate variables). We use the standard notation $\eta\{v/V\}$ for *updating* η so that the new environment maps V to v and acts otherwise as η. Predicate maps $f : (v_1,\ldots,v_n) \mapsto A \subseteq \mathcal{S}$ are ordered by $\leq$ defined as subset containment lifted pointwise.

Definition 5 (Formula semantics). *The semantics of formulas is defined by table 2, where in the last two defining equations,* $M(f)(v_1,\ldots,v_n) \stackrel{\Delta}{=} \parallel \phi \parallel \eta\{f/Z, v_1/V_1,\ldots,v_n/V_n\}$.

$$
\begin{aligned}
\| p_1 = p_2 \|\eta &\triangleq \{s \mid p_1\eta = p_2\eta\} \\
\| p_1 \neq p_2 \|\eta &\triangleq \{s \mid p_1\eta \neq p_2\eta\} \\
\| \texttt{term}(p_1) = p_2 \|\eta &\triangleq \{\langle p_2\eta, p_1\eta, q\rangle \parallel s \mid p_1\eta \text{ is a pid}\} \\
\| \texttt{queue}(p_1) = p_2 \|\eta &\triangleq \{\langle e, p_1\eta, p_2\eta\rangle \parallel s \mid p_1\eta \text{ is a pid}\} \\
\| \texttt{local}(p) \|\eta &\triangleq \{\langle e, p\eta, q\rangle \parallel s \mid p\eta \text{ is a pid}\} \\
\| \texttt{foreign}(p) \|\eta &\triangleq \{s \mid p\eta \text{ is a pid that does not belong to a process in } s\} \\
\| \texttt{atom}(p) \|\eta &\triangleq \{s \mid p\eta \text{ is an atom}\} \\
\| \texttt{unevaluated}(p) \|\eta &\triangleq \{\langle e, p\eta, q\rangle \parallel s \mid e \text{ is not a ground value, } p\eta \text{ is a pid}\} \\
\| \phi_1 \wedge \phi_2 \|\eta &\triangleq \| \phi_1 \|\eta \cap \| \phi_2 \|\eta \\
\| \phi_1 \vee \phi_2 \|\eta &\triangleq \| \phi_1 \|\eta \cup \| \phi_2 \|\eta \\
\| \forall V.\phi \|\eta &\triangleq \bigcap \{\| \phi \|\eta\{v/V\} \mid v \text{ a value}\} \\
\| \exists V.\phi \|\eta &\triangleq \bigcup \{\| \phi \|\eta\{v/V\} \mid v \text{ a value}\} \\
\| <>\phi \|\eta &\triangleq \{s \mid \exists\, s' \in \| \phi \|\eta.\ s \longrightarrow s'\} \\
\| <p_1?p_2>\phi \|\eta &\triangleq \{s \mid \exists\, s' \in \| \phi \|\eta.\ s \xrightarrow{p_1\eta?p_2\eta} s'\} \\
\| <p_1!p_2>\phi \|\eta &\triangleq \{s \mid \exists\, s' \in \| \phi \|\eta.\ s \xrightarrow{p_1\eta!p_2\eta} s'\} \\
\| [\,]\phi \|\eta &\triangleq \{s \mid \forall\, s'.\ s \longrightarrow s' \text{ implies } s' \in \| \phi \|\eta\} \\
\| [p_1?p_2]\phi \|\eta &\triangleq \{s \mid \forall\, s'.\ s \xrightarrow{p_1\eta?p_2\eta} s' \text{ implies } s' \in \| \phi \|\eta\} \\
\| [p_1!p_2]\phi \|\eta &\triangleq \{s \mid \forall\, s'.\ s \xrightarrow{p_1\eta!p_2\eta} s' \text{ implies } s' \in \| \phi \|\eta\} \\
\| Z(p_1, \ldots, p_n) \|\eta &\triangleq (Z\eta)(p_1\eta, \ldots, p_n\eta) \\
\| (\mu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n) \|\eta & \\
&\triangleq \left(\bigcap \{f \mid M(f) \leq f\}\right)(p_1\eta, \ldots, p_n\eta) \\
\| (\nu Z(V_1, \ldots, V_n).\phi)(p_1, \ldots, p_n) \|\eta & \\
&\triangleq \left(\bigcup \{f \mid f \leq M(f)\}\right)(p_1\eta, \ldots, p_n\eta)
\end{aligned}
$$

Table 2. Formula Semantics

Example Formulas The combination of recursive definitions with data types makes the logic very expressive. For instance, the type of natural numbers is the least set containing zero and closed under successor. The property of being a natural number can hence be defined as a least fixpoint:

$$nat(P) \Leftarrow P = 0 \vee \exists V.(nat(V) \wedge P = V + 1) \tag{1}$$

Using this idea quite general flat data types can be defined. One can also define "weak" modalities that are insensitive to the specific number of internal transitions in the following style:

$$[[\,]]\phi \Rightarrow \phi \wedge [\,][[\,]]\phi \qquad [[V!V']]\phi \stackrel{\Delta}{=} [[\,]][V!V'][[\,]]\phi$$
$$\ll\gg\phi \Leftarrow \phi \vee <><\ll\gg\phi \qquad \ll V!V'\gg\phi \stackrel{\Delta}{=} \ll\gg<V!V'><\ll\gg\phi$$

Observe the use of formula parameters, and the use of $\stackrel{\Delta}{=}$ for non-recursive definitions. A temporal property like *always* is also easily defined:

$$\begin{aligned} &always(\phi) \Rightarrow \\ &\quad \phi \wedge [\,]always(\phi) \wedge \forall V, V'.[V?V']always(\phi) \wedge \forall V, V'.[V!V']always(\phi) \end{aligned} \tag{2}$$

The definition of *always* illustrates the typical shape of behavioural properties: one mixes state properties (ϕ) with properties concerning program transitions, separated into distinct cases for internal transitions (i.e. $[\,]always(\phi)$), input transitions and output transitions.

Eventuality operators are more delicate as progress is in general only made when internal or output transitions are taken. This can be handled, though, by nesting of minimal and maximal definitions.

Billing Agents: Specification We enumerate some desired properties of the billing agent system introduced in the previous section.

Disallowing spontaneous account withdrawals. The first correctness requirement forbids spontaneous withdrawals from the user account by the billing agent. This implies the invariant property that the number of attempts for transfers from the user account should be less than or equal to the number of requests for using the resource:

$$\begin{aligned} &safe(Ag, BankPid, UAcc, N) \quad \Rightarrow \\ &[\,]safe(Ag, BankPid, UAcc, N) \\ &\wedge\ \forall P, V.[P?V] \\ &\quad \left(\begin{array}{l} isuse(P, V, Ag) \wedge safe(Ag, BankPid, UAcc, N+1) \\ \vee\ \mathtt{not}(isuse(P, V, Ag)) \wedge safe(Ag, BankPid, UAcc, N) \\ \vee\ contains(V, UAcc) \end{array} \right) \\ &\wedge\ \forall P, V.[P!V] \\ &\quad \left(\begin{array}{l} istrans(P, V, BankPid, UAcc) \wedge N > 0 \wedge safe(Ag, BankPid, UAcc, N-1) \\ \vee\ \mathtt{not}(istrans(P, V, BankPid, UAcc)) \wedge safe(Ag, BankPid, UAcc, N) \end{array} \right) \end{aligned}$$

where the predicates *isuse* and *istrans* recognise resource requests and money transfers:

$$\begin{aligned} &isuse(P, V, Ag) \triangleq P = Ag \wedge \exists Pid.V = \{use, Pid\} \\ &istrans(P, V, BankPid, UAcc) \triangleq \\ &\quad P = BankPid \wedge (\exists Pid, Acc.\, V = \{trans, \{UAcc, Acc\}, Pid) \end{aligned}$$

So, a billing agent with pid Ag, pid of a trusted bank agent $BankPid$, and user account $UAcc$ is defined to be *safe* if the difference N between the number of requests for using the resource (messages of type $\{use, Pid\}$ received in the process mailbox) and the number of attempts for transfers from the user account (messages of type $\{trans, \{UAcc, Acc\}, pid\}$ sent to $BankPid$) is always non-negative. Since this difference is initially equal to zero, we expect $billagent(ResPid, BankPid, RAcc, UAcc)$ to satisfy $safe(Ag, BankPid, UAcc, 0)$. The predicate $contains(v, v')$ is defined via structural induction over an Erlang value (or queue) v and holds if v' is a component of v (such as v being a tuple $v = \{v_1, v_2\}$ and either $v = v'$ or $contains(v_1, v')$ or $contains(v_2, v')$). We omit the easy definition.

Expected Service is Received. Other interesting properties concern facts like: Denial of service responses correspond to failed money transfers, and returning the resource to the proper user. These sorts of properties are not hard to formalise in a style similar to the first example.

Preventing Abuse by a Third Party. The payment scheme presented here depends crucially on the non-communication of private names. For instance, even if we can prove that a resource manager or billing agent does not make illegal withdrawals nothing stops the resource manager from communicating the user account key to a third party, that can then access the account in non-approved ways.

Thus we need to prove at least that the system communicates neither the user account key nor the agent process identifier. Perhaps the service user also requests that her identity not be known outside of the system, in such a case the return process identifiers may not be communicated either. As an example, the property that the system does not communicate the user account key is captured by $notrans(UAcc)$ given the definition below.

$$\begin{aligned} notrans(A) \Rightarrow\ & [\,]\ notrans(A) \\ & \wedge\ \forall V, V'.[V?V']\,(contains(V', A) \vee notrans(A)) \\ & \wedge\ \forall V, V'.[V!V']\,(\mathtt{not}(contains(V', A)) \wedge notrans(A)) \end{aligned}$$

4 Proof System, Local Rules

In this section we give the formal semantics of sequents, present the most important rules of the proof system and establish the soundness of the proof rules. Consideration of fixed points and discharge is delayed until later.

Sequents We start by introducing the syntax and semantics of sequents.

Definition 6 (Sequent, Sequent Semantics).

1. *An* assertion *is a pair* $s : \phi$. *An* assumption *is an assertion of the shape either* $S : \phi$ *or* $\langle E, P, Q\rangle : \phi$.
2. *A valuation* η *is said to* validate *an assertion* $s : \phi$ *when* $s\eta \in \| \phi \|\, \eta$.
3. *A* sequent *is an expression of the shape* $\Gamma \vdash \Delta$ *where* Γ *is a multiset of assumptions, and* Δ *is a multiset of assertions.*
4. *The sequent* $\Gamma \vdash \Delta$ *is* valid, *written* $\Gamma \models \Delta$, *provided for all valuations* η, *if all assumptions in* Γ *are validated by* η, *then some assertion in* Δ *is validated by* η.

We use standard notation for sequents. For instance we use comma for multiset union, and identify singleton sets with their member. Thus, for instance, $\Gamma, S : \phi$ is the same multiset as $\Gamma \cup \{S : \phi\}$.

Boolean formulas are independent of the system state being predicated. Thus, when ϕ is boolean we often abbreviate an assertion like $s : \phi$ as just ϕ.

Proof Rules The Gentzen-type proof system comes in a number of installments.

1. The *structural rules* govern the introduction, elimination, and use of assertions.
2. The *logical rules* introduce the logical connectives to the left and to the right of the turnstile.
3. The *equality* rules account for equations and inequations.
4. The *atomic formula rules* control the Erlang-specific atomic formulas such as the term and queue extractors, predicates such as `local` and `foreign`, etc.
5. Finally the *modal rules* account for the modal operators.

It should be noted that, as we for now lack completeness results, the selection of rules which we present is to some extent arbitrary. Indeed some rules have deliberately been left out of this presentation for reasons of space. This is the case, in particular, for the groups (4) and (5). We comment more on this below. Moreover, because of the lack of completeness or syntactical cut-elimination results, we know little about admissibility of several rules such as Cut below.

Structural Rules

$$\mathsf{Id}\ \frac{}{\Gamma, s : \phi \vdash s : \phi, \Delta} \qquad \mathsf{Bool\ Id}\ \frac{}{\Gamma, \phi \vdash \phi, \Delta}\ \phi\ \textit{Boolean}$$

$$\mathsf{WeakL}\ \frac{\Gamma \vdash \Delta}{\Gamma, s : \phi \vdash \Delta} \qquad \mathsf{WeakR}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash s : \phi, \Delta}$$

$$\mathsf{ContrL}\ \frac{\Gamma, s : \phi, s : \phi \vdash \Delta}{\Gamma, s : \phi \vdash \Delta} \qquad \mathsf{ContrR}\ \frac{\Gamma \vdash s : \phi, s : \phi, \Delta}{\Gamma \vdash s : \phi, \Delta}$$

$$\mathsf{Cut}\ \frac{\Gamma \vdash s : \phi, \Delta \qquad \Gamma, s : \phi \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\mathsf{ProcCut}\ \frac{\Gamma \vdash s_1 : \psi, \Delta \qquad \Gamma, S : \psi \vdash s_2 : \phi, \Delta}{\Gamma \vdash s_2\{s_1/S\} : \phi, \Delta}\ S\ \mathit{fresh}$$

Observe that the standard cut rule is not derivable from the process cut rule as in Cut the system state s might appear in Γ or Δ.

Logical Rules

$$\mathsf{AndL}\ \frac{\Gamma, s : \phi_1, s : \phi_2 \vdash \Delta}{\Gamma, s : \phi_1 \wedge \phi_2 \vdash \Delta} \quad \mathsf{AndR}\ \frac{\Gamma \vdash s : \phi_1, \Delta \qquad \Gamma \vdash s : \phi_2, \Delta}{\Gamma \vdash s : \phi_1 \wedge \phi_2, \Delta}$$

$$\mathsf{OrL}\ \frac{\Gamma, s : \phi_1 \vdash \Delta \qquad \Gamma, s : \phi_2 \vdash \Delta}{\Gamma, s : \phi_1 \vee \phi_2 \vdash \Delta} \quad \mathsf{OrR}\ \frac{\Gamma \vdash s : \phi_1, s : \phi_2, \Delta}{\Gamma \vdash s : \phi_1 \vee \phi_2, \Delta}$$

$$\mathsf{All L}\ \frac{\Gamma, s : \phi\{v/V\} \vdash \Delta}{\Gamma, s : \forall V.\phi \vdash \Delta} \quad \mathsf{All R}\ \frac{\Gamma \vdash s : \phi, \Delta}{\Gamma \vdash s : \forall V.\phi, \Delta}\ V\ \mathit{fresh}$$

$$\mathsf{ExL}\ \frac{\Gamma, s : \phi \vdash \Delta}{\Gamma, s : \exists V.\phi \vdash \Delta}\ V\ \mathit{fresh} \quad \mathsf{ExR}\ \frac{\Gamma \vdash s : \phi\{v/V\}, \Delta}{\Gamma \vdash s : \exists V.\phi, \Delta}$$

We claim that for the full proof system the left and right `not` introduction rules

$$\mathsf{NotL}\ \frac{\Gamma, s : \mathtt{not}\ \phi \vdash \Delta}{\Gamma \vdash s : \phi, \Delta} \quad \mathsf{NotR}\ \frac{\Gamma \vdash s : \mathtt{not}\ \phi, \Delta}{\Gamma, s : \phi \vdash \Delta}$$

are derivable. Given this a number of other useful rules such as the rule of contradiction

$$\mathsf{Contrad}\ \frac{\Gamma \vdash s : \phi, \Delta \qquad \Gamma \vdash s : \mathtt{not}\ \phi, \Delta}{\Gamma \vdash \Delta}$$

become easily derivable as well.

Equality Rules For the rules which follow recall that op ranges over data type constructors like zero, successor, unit, binary tupling, etc.

$$\mathsf{Refl}\ \frac{}{\Gamma \vdash p = p, \Delta}$$

$$\mathsf{Subst}\ \frac{\Gamma\{p_1/V\} \vdash \Delta\{p_1/V\}}{\Gamma\{p_2/V\}, p_1 = p_2 \vdash \Delta\{p_2/V\}}$$

$$\mathsf{ConstrIneq}\ \frac{op \neq op'}{\Gamma, op(p_1, \ldots, p_n) = op'(p'_1, \ldots, p'_{n'}) \vdash \Delta}$$

$$\mathsf{ConstrEqL}\ \frac{\Gamma, p_i = p'_i \vdash \Delta}{\Gamma, op(p_1, \ldots, p_n) = op(p'_1, \ldots, p'_n) \vdash \Delta}$$

$$\mathsf{ConstrEqR}\ \frac{\Gamma \vdash p_1 = p'_1, \Delta \quad \cdots \quad \Gamma \vdash p_n = p'_n, \Delta}{\Gamma \vdash op(p_1, \ldots, p_n) = op(p'_1, \ldots, p'_n), \Delta}$$

Atomic Formula Rules For the Erlang-specific primitives we give only a sample here, of rules governing the term and queue extraction constructs.

$$\mathsf{TermL}\ \frac{\Gamma, E = p \vdash \Delta}{\Gamma, \langle E, Pid, Q\rangle : \mathtt{term}(Pid) = p \vdash \Delta}$$

$$\mathsf{TermR}\ \frac{\Gamma \vdash p_1 = p_2, \Delta}{\Gamma \vdash \langle p_1, pid, q\rangle : \mathtt{term}(pid) = p_2, \Delta}$$

$$\mathsf{ParTerm}\ \frac{\Gamma \vdash s_1 : \mathtt{term}(p_1) = p_2, \Delta}{\Gamma \vdash s_1 \parallel s_2 : \mathtt{term}(p_1) = p_2, \Delta}$$

$$\mathsf{QueueL}\ \frac{\Gamma, Q = p \vdash \Delta}{\Gamma, \langle E, Pid, Q\rangle : \mathtt{queue}(Pid) = p \vdash \Delta}$$

$$\mathsf{QueueR}\ \frac{\Gamma \vdash p_2 = p_3, \Delta}{\Gamma \vdash \langle p_1, pid, p_2\rangle : \mathtt{queue}(pid) = p_3, \Delta}$$

$$\mathsf{ParQueue}\ \frac{\Gamma \vdash s_1 : \mathtt{queue}(p_1) = p_2, \Delta}{\Gamma \vdash s_1 \parallel s_2 : \mathtt{queue}(p_1) = p_2, \Delta}$$

Modal Rules The rules governing the modal operators come in three flavours: *Monotonicity rules* that capture the monotonicity of the modal operators as unary functions on sets, *transition rules* ("model checking-like rules") that prove modal properties for closed system states by exploring the operational semantics transition relations, and *compositional rules* that decompose modal properties of composite system states in terms of modal properties of their components.

Monotonicity Rules The monotonicity rules are similar to well-known rules from standard Gentzen-type accounts of modal logic. Let α stand for either no label or for a label of the form $p_1?p_2$ or $p_1!p_2$.

$$\mathsf{Mon1}\ \frac{\Gamma, S : \phi, S : \phi_1, \ldots, S : \phi_m \vdash S : \psi_1, \ldots, S : \psi_n, \Delta}{\Gamma, s : <\alpha>\phi, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m \vdash s : <\alpha>\psi_1, \ldots, s : <\alpha>\psi_n, \Delta}\ S \textit{ fresh}$$

$$\mathsf{Mon2}\ \frac{\Gamma, S : \phi_1, \ldots, S : \phi_m \vdash S : \psi, S : \psi_1, \ldots, S : \psi_n, \Delta}{\Gamma, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m \vdash s : [\alpha]\psi, s : <\alpha>\psi_1, \ldots, s : <\alpha>\psi_n, \Delta}\ S \textit{ fresh}$$

Transition Rules The transition rules explore symbolic versions of the operational semantics transition relations. These relations have the shape

$$s \xrightarrow{pre, \alpha, post} s'$$

where *pre* is a (first-order) precondition for firing the transition from s to s', α (as above) is the transition label, and *post* is the resulting (first-order) postcondition in terms of e.g. variable bindings. Since the transformation of the semantics in

table 1 to a symbolic one is straightforward, we will only give a single example of the transformation where the original rule

$$\text{SEND} : \langle r[pid'!v], pid, q\rangle \xrightarrow{pid'!v} \langle r[v], pid, q\rangle \text{ if } pid \neq pid'$$

becomes the symbolic rule

$$\text{SEND}_\text{s} : \langle r[pid'!v], pid, q\rangle \xrightarrow{pid \neq pid', pid'!v, true} \langle r[v], pid, q\rangle .$$

The transition rules now become the following two:

$$\textsf{Diamond}\ \frac{\Gamma, pre, post \vdash s' : \phi \quad \Gamma \vdash pre \quad s \xrightarrow{pre, \alpha, post} s'}{\Gamma \vdash s : <\alpha>\phi}$$

$$\textsf{Box}\ \frac{\{\Gamma, \alpha = \alpha', pre, post \vdash s' : \phi \mid s \xrightarrow{pre, \alpha', post} s'\}}{\Gamma \vdash s : [\alpha]\phi}$$

In the rule Box we use $\alpha = \alpha'$ as an abbreviation. If α and α' have different sorts (e.g. α is empty and α' is not) then $\alpha = \alpha'$ abbreviates *false*. If α and α' are both empty, $\alpha = \alpha'$ abbreviates *true*. If α and α' are both send actions, e.g. $\alpha = p_1!p_2$ and $\alpha' = p_1'!p_2'$ then $\alpha = \alpha'$ abbreviates the conjunction $p_1 = p_1' \wedge p_2 = p_2'$. A similar condition holds if α and α' are both input actions.

Compositional Rules We give rules for inferring modal properties of composite system states $s_1 \parallel s_2$ in terms of modal properties of the parts s_1 and s_2.

$$\textsf{DiaPar1}\ \frac{\Gamma, S_1 : \phi, S_2 : \psi \ \vdash\ S_1 \parallel S_2 : \theta, \Delta}{\Gamma, s_1 : <p_1!p_2>\phi, s_2 : <p_1?p_2>\psi \ \vdash\ s_1 \parallel s_2 : <>\theta, \Delta}\ S_1, S_2 \textit{ fresh}$$

$$\textsf{DiaPar2}\ \frac{\Gamma, S : \phi \ \vdash\ S \parallel s_2 : \psi, \Delta}{\Gamma, s_1 : <>\phi \ \vdash\ s_1 \parallel s_2 : <>\psi, \Delta}\ S \textit{ fresh}$$

$$\textsf{DiaPar3}\ \frac{\Gamma, S : \phi \ \vdash\ S \parallel s_2 : \psi, \Delta}{\Gamma, s_1 : <p_1?p_2>\phi \ \vdash\ s_1 \parallel s_2 : <p_1?p_2>\psi, \Delta}\ S \textit{ fresh}$$

$$\textsf{DiaPar4}\ \frac{\Gamma, S : \phi \ \vdash\ S \parallel s_2 : \psi, \Delta \qquad \Gamma \ \vdash\ s_2 : \texttt{foreign}(p_1)}{\Gamma, s_1 : <p_1!p_2>\phi \ \vdash\ s_1 \parallel s_2 : <p_1!p_2>\psi, \Delta}\ S \textit{ fresh}$$

$$\Gamma, S_1 : \phi_1 \vdash S_1 : [\,]\,\psi_{[\,]}, \Delta$$
$$\Gamma, S_1 : \psi_{[\,]}, s_2 : \phi_2 \vdash S_1 \parallel s_2 : \phi$$
$$\Gamma, S_2 : \phi_2 \vdash S_2 : [\,]\,\theta_{[\,]}, \Delta$$
$$\Gamma, s_1 : \phi_1, S_2 : \theta_{[\,]} \vdash s_1 \parallel S_2 : \phi$$
$$\Gamma, S_1 : \phi_1 \vdash S_1 : [V!V']\,\psi_{[V!V']}, \Delta$$
$$\Gamma, S_2 : \phi_2 \vdash S_2 : [V?V']\,\theta_{[V?V']}, \Delta$$
$$\Gamma, S_1 : \psi_{[V!V']}, S_2 : \theta_{[V?V']} \vdash S_1 \parallel S_2 : \phi$$
$$\Gamma, S_1 : \phi_1 \vdash S_1 : [V?V']\,\psi_{[V?V']}, \Delta$$
$$\Gamma, S_2 : \phi_2 \vdash S_2 : [V!V']\,\theta_{[V!V']}, \Delta$$
$$\textsf{BoxPar1}\ \frac{\Gamma, S_1 : \psi_{[V?V']}, S_2 : \theta_{[V!V']} \vdash S_1 \parallel S_2 : \phi}{\Gamma, s_1 : \phi_1, s_2 : \phi_2 \vdash s_1 \parallel s_2 : [\,]\,\phi, \Delta}\ S_1, S_2, V, V' \textit{ fresh}$$

$$\Gamma, S_1 : \phi_1 \vdash S_1 : [p_1?!p_2]\,\psi_{[p_1?!p_2]}, \Delta$$
$$\Gamma, S_1 : \psi_{[p_1?!p_2]}, s_2 : \phi_2 \vdash S_1 \parallel s_2 : \phi$$
$$\Gamma, S_2 : \phi_2 \vdash S_2 : [p_1?!p_2]\,\theta_{[p_1?!p_2]}, \Delta$$
$$\textsf{BoxPar2}\ \frac{\Gamma, s_1 : \phi_1, S_2 : \theta_{[p_1?!p_2]} \vdash s_1 \parallel S_2 : \phi}{\Gamma, s_1 : \phi_1, s_2 : \phi_2 \vdash s_1 \parallel s_2 : [p_1?!p_2]\,\phi, \Delta}\ S_1, S_2 \textit{ fresh}$$

The last two rules are less complex than they appear. They just represent the obvious case analyses needed to infer the α-indexed necessity properties stated in their conclusions. For instance in the case where α is empty it is clearly required to analyse all cases where either

- s_1 performs an internal transition and s_2 does not,
- s_2 performs an internal transition and s_1 does not,
- s_1 performs a send action and s_2 performs a corresponding receive action, and
- s_2 performs a send action and s_1 performs a corresponding receive action.

In fact, the last two rules are simplified versions of more general rules having multiple assumptions about s_1 and s_2 in the conclusion. However, a formal statement of these rules, while quite trivial, becomes graphically rather unwieldy.

Observe that these rules deal only with composite system states of the form $s_1 \parallel s_2$. Related compositional rules are needed also for singleton processes $\langle e, \mathit{pid}, q\rangle$, to decompose properties of e in terms of properties of its constituent parts. For closed processes, however, and for tail-recursive programs e (as is the case in the main example considered later), the transition rules are quite adequate, and so this collection of compositional proof rules for sequential processes is not considered further in the present paper.

Theorem 1 (Local Soundness). *Each of the above rules is sound, i.e. the conclusion is a valid sequent whenever all premises are so and all side conditions hold.*

Proof. Here we show soundness of the more interesting rules only; the proofs of the remaining rules are either standard or similar to these.

Rule ProcCut. Assume the premises of the rule are valid sequents. Assume all assumptions in Γ are validated by some arbitrary valuation η. Then validity of the first premise implies that also some assertion in $s_1 : \psi, \Delta$ is validated by η. Since S is fresh (i.e. not free in Γ or Δ) and since $s_1\eta$ equals $S\eta\{s_1\eta/S\}$, some assertion in $S : \psi, \Delta$ is validated by $\eta\{s_1\eta/S\}$. Then either some assertion in Δ is validated by $\eta\{s_1\eta/S\}$, or all assertions in $\Gamma, S : \psi$ are validated by $\eta\{s_1\eta/S\}$. In the latter case, the validity of the second premise implies that also some assertion in $s_2 : \phi, \Delta$ is validated by $\eta\{s_1\eta/S\}$. Since S is fresh and since $s_2\eta\{s_1\eta/S\}$ equals $s_2\{s_1/S\}\eta$, some assertion in $s_2\{s_1/S\} : \phi, \Delta$ is validated by η. Hence the conclusion of the rule is also a valid sequent.

Rule Mon1. Assume the premise of the rule is a valid sequent. Assume all assumptions in $\Gamma, s : <\alpha>\phi, s : [\alpha]\phi_1, \ldots, s : [\alpha]\phi_m$ are validated by η. Then, according to the semantics of the "diamond" and "box" modalities, there is a closed system state s' such that $s\eta \xrightarrow{\alpha\eta} s'$ and s' satisfies all formulas in $\phi, \phi_1, \ldots, \phi_m$ under η. Since s' equals $S\eta\{s'/S\}$, and since S is fresh, all assumptions in $\Gamma, S : \phi, S : \phi_1, \ldots, S : \phi_m$ are validated by $\eta\{s'/S\}$. By validity of the premise, some assertion in $S : \psi_1, \ldots, S : \psi_n, \Delta$ is also validated by $\eta\{s'/S\}$. This means that either some assertion in Δ is validated by η, or s' satisfies some formula in $\psi_1, \ldots, \psi_n$ under η, and consequently some assertion in $s : <\alpha>\psi_1, \ldots, s : <\alpha>\psi_n, \Delta$ is validated by η. Hence the conclusion of the rule is also a valid sequent.

Rule DiaPar1. Assume the premise of the rule is valid. Assume all assumptions in $\Gamma, s_1 : <p_1!p_2>\phi, s_2 : <p_1?p_2>\psi$ are validated by η. Then there are closed system states s' and s'' such that $s_1\eta \xrightarrow{p_1\eta!p_2\eta} s' \in\| \phi \|\ \eta$ and $s_2\eta \xrightarrow{p_1\eta?p_2\eta} s'' \in\| \psi \|\ \eta$. Since s' equals $S_1\eta\{s'/S_1, s''/S_2\}$ and s'' equals $S_2\eta\{s'/S_1, s''/S_2\}$, and since S_1 and S_2 are fresh, all assumptions in $\Gamma, S_1 : \phi, S_2 : \psi$ are validated by $\eta\{s'/S_1, s''/S_2\}$. Then, by validity of the premise, some assertion in $S_1 \parallel S_2 : \theta, \Delta$ is also validated by $\eta\{s'/S_1, s''/S_2\}$. As a consequence, either some assertion in Δ is validated by η, or $s' \parallel s'' \in\| \theta \|\ \eta$. In the latter case we have $s_1\eta \parallel s_2\eta \longrightarrow s' \parallel s'' \in\| \theta \|\ \eta$, and therefore some assertion in $s_1 \parallel s_2 : <>\theta, \Delta$ is validated by η. Hence the conclusion of the rule is also valid.

Rule BoxPar1. The proof is along the lines of the preceding ones and shall only be sketched here. Assume the premises to the rule are valid sequents. Assume all assumptions in $\Gamma, s_1 : \phi_1, s_2 : \phi_2$ are validated by η. Then, from the first two premises it follows that either some assertion in Δ is validated by η, or it is the case that for every closed system state s such that $s_1\eta \longrightarrow s$, process $s \parallel s_2\eta$ satisfies ϕ under η. Similarly, the next two assumptions imply that $s_1\eta \parallel s$ satisfies ϕ under η whenever $s_2\eta \longrightarrow s$. From the next group of three assumptions we obtain that $s' \parallel s''$ satisfies ϕ under η whenever $s_1\eta \xrightarrow{v'!v''} s'$ and $s_2\eta \xrightarrow{v'?v''} s''$ for some values v' and v''. The last three premises imply that $s' \parallel s''$ satisfies ϕ under η whenever $s_1\eta \xrightarrow{v'?v''} s'$ and $s_2\eta \xrightarrow{v'!v''} s''$ for some values v' and v''. As a consequence of these relationships, either some assertion in Δ is validated by η, or every closed system state s such that $s_1\eta \parallel s_2\eta \longrightarrow s$

satisfies ϕ under η. Hence, some assertion in $s_1 \parallel s_2 : [\,]\,\phi, \Delta$ is validated by η, and therefore the conclusion of the rule is valid. □

5 Inductive and Coinductive Reasoning

To handle recursively defined formulas some mechanisms are needed for successfully terminating proof construction when this is deemed to be safe. We discuss the ideas on the basis of two examples.

Example 1 (Coinduction). Consider the following Core Erlang function:

$$stream(N, Out) \rightarrow Out!N, stream(N+1, Out).$$

which outputs the increasing stream N, $N+1$, $N+2$, ... along Out. The specification of the program could be that it can always output some value along Out:

$$stream_spec(Out) = always(\exists X. \ll Out!X \gg true)$$

The goal sequent takes the shape

$$Out \neq P \vdash \langle stream(N, Out), P, Q\rangle : stream_spec(Out). \tag{3}$$

That is, assuming $Out \neq P$ (since otherwise output will go to the stream process itself), and started with pid P and any input queue Q, the property *stream_spec*(Out) will hold. The first step is to unfold the formula definition. This results in a proof goal of the shape

$$Out \neq P \vdash \langle stream(N, Out), P, Q\rangle : always(\exists X. \ll Out!X \gg true). \tag{4}$$

Using the proof rules (4) is easily reduced to the following subgoals:

$$Out \neq P \vdash \langle stream(N, Out), P, Q\rangle : \ll Out!N \gg true \tag{5}$$

$$Out \neq P \vdash \langle stream((N+1), Out), P, Q\rangle : always(\exists X. \ll Out!X \gg true) \tag{6}$$

$$Out \neq P \vdash \langle stream(N, Out), P, Q \cdot V'\rangle : always(\exists X. \ll Out!X \gg true). \tag{7}$$

Proving (5) is straightforward using the local proof rules and fixed point unfolding. For (6) and (7) we see that these goals are both instances of a proof goal, namely (4), which has already been unfolded.

Continuing proof construction in example 1 beyond (6) and (7) is clearly futile: The further information contained in these sequents compared with (4) does not bring about any new potentiality for proof. So we would like the nodes (6) and (7) to be discharged, as this happens to be safe. This is not hard to see: The fixed point

$$\phi = always(\exists V. \ll Out!V \gg true)$$

can only appear at its unique position in the sequent (6) because it did so in the sequent (4). We can say that ϕ is regenerated along the path from (4) to (6).

Moreover, *always* constructs a greatest fixed point formula. It turns out that the sequent (6) can be discharged for these reasons. In general, however, fixed point unfoldings are not at all as easily analysed. Alternation is a well-known complication. The basic intuition, however, is that in this case sequents can be discharged for one of two reasons:

1. Coinductively, because a member of Δ is regenerated through a greatest fixed point formula, as in example 1.
2. Inductively, because a member of Γ is regenerated through a least fixed point formula.

Intuitively the assumption of a least fixed point property can be used to determine a kind of progress measure ensuring that loops of certain sorts must eventually be exited. This significantly increases the utility of the proof system. For instance it allows for datatype induction to be performed.

Example 2 (Induction). Consider the following function:

$$stream2(N+1, Out) \rightarrow Out!N, stream2(N, Out)$$

which outputs a decreasing stream of numbers along Out. If N is a natural number, *stream2* has the property that, provided it does not deadlock, it will eventually output zero along Out. This property can be formalised as follows:

$$evzero(Out) \Leftarrow [\,]evzero(Out) \wedge \forall V.[Out!V](V = 0 \vee evzero(Out))$$

The goal sequent is

$$Out \neq P, nat(N) \vdash \langle stream2(N+1, Out), P, Q\rangle : evzero(Out) \qquad (8)$$

where *nat* is defined in (1). The least fixed point appearing in the definition of *nat* will be crucial for discharge later in the proof. By unfolding the function according to its definition we obtain the sequent:

$$Out \neq P, nat(N) \vdash \langle (Out!N, stream2(N, Out)), P, Q\rangle : evzero(Out)$$

Now the formula has to be unfolded, resulting in a conjunction and hence in two sub-goals. The first of these is proved trivially since the system state cannot perform any internal transition when $Out \neq P$. The second sub-goal, after handling the universal quantifier, becomes:

$$\begin{aligned} &Out \neq P, nat(N) \\ &\vdash \langle (Out!N, stream2(N, Out)), P, Q\rangle : [Out!V](V = 0 \vee evzero(Out)) \end{aligned} \qquad (9)$$

By following the output transition enabled at this state we come a step closer to showing that zero is eventually output along Out:

$$Out \neq P, nat(N), N = V \vdash \langle stream2(N, Out), P, Q\rangle : V = 0, evzero(Out) \qquad (10)$$

In the next step we perform a case analysis on N by unfolding $nat(N)$. This results in a disjunction on the left amounting to whether N is zero or not, and yields the two sub-goals:

$$Out \neq P, N = 0, N = V \vdash \langle stream2(N, Out), P, Q\rangle : V = 0 \tag{11}$$

$$Out \neq P, nat(N'), N = N' + 1, N = V \vdash \langle stream2(N, Out), P, Q\rangle : evzero(Out) \tag{12}$$

The first of these is proved trivially. The second can be simplified to:

$$Out \neq P, nat(N') \vdash \langle stream2(N' + 1, Out), P, Q\rangle : evzero(Out) \tag{13}$$

This sequent is an instance of the initial goal sequent (8). Furthermore, it was obtained by regenerating the least fixpoint formula $nat(N)$ on the left. This provides the progress required to discharge (13).

Finitary data types in general can be specified using least fixpoint formulas. This allows for termination or eventuality properties of programs to be proven along the lines of the above example. In a similar way we can handle program properties that depend on inductive properties of message queues.

6 Proof Rules for Recursive Formulas

The approach we use to handle fixed points, and capture the critical notions of "regeneration", "progress", and "discharge" is, essentially, formalised well-founded induction. When some fixed points are unfolded, notably least fixed points to the left of the turnstile, and greatest fixed points to the right of the turnstile, it is possible to pin down suitable *approximation ordinals* providing, for least fixed points, a progress measure toward satisfaction and, for greatest fixed points, a progress measure toward refutation. We introduce explicit ordinal variables which are maintained, and suitably decremented, as proofs are elaborated. This provides a simple method for dealing with a variety of complications such as alternation of fixpoints and the various complications related to duplication and interference between fixed points that are dealt with using the much more indirect approach of [Dam98].

We first pin down some terminology concerning proofs. A *proof structure* is a finite, rooted, sequent-labelled tree which respects the proof rules in the sense that if π is a proof node labelled by the sequent δ, and if $\pi_1, \ldots, \pi_n$ are the children of π in left to right order labelled by $\delta_1, \ldots, \delta_n$ then

$$\frac{\delta_1 \quad \cdots \quad \delta_n}{\delta}$$

is a substitution instance of one of the proof rules. A node π is *elaborated* if its label is the conclusion of a rule instance as above. A *proof* is a proof structure for which all nodes are elaborated. In the context of a given proof structure we

write $\pi_1 \rightarrow \pi_2$ if π_2 is a child of π_1. A *path* is a finite sequence $\Pi = \pi_1, \ldots, \pi_n$ for which $\pi_i \rightarrow \pi_{i+1}$ for all $i : 1 \leq i < n$. Generally we use the term "sequent occurrence" as synonymous with node. However, when the intention is clear from the context we sometimes confuse sequents with sequent occurrences and write eg. $\delta_1 \rightarrow \delta_2$ in place of $\pi_1 \rightarrow \pi_2$ when the sequents label their corresponding nodes.

Ordinal Approximations In general, soundness of fixed point induction relies on the well-known iterative characterisation where least and greatest fixed points are "computed" as iterative limits of their ordinal approximations. This also happens in the present case. Let κ range over ordinal variables. Valuations and substitutions are extended to map ordinal variables to ordinals. Let U, V range over fixed point formula abstractions of the form $\sigma Z(V_1, \ldots, V_n).\phi$. We introduce new formulas of the shape U^κ and $\kappa < \kappa'$. Ordinal inequalities have their obvious semantics, and $\kappa \leq \kappa'$ abbreviates $\kappa < \kappa' \vee \kappa = \kappa'$ as usual. For approximated fixed point abstractions suppose first that $U = \sigma Z(V_1, \ldots, V_k).\phi$ and $\sigma = \nu$. Then

$$\| U^\alpha(P_1, \ldots, P_k) \| \eta = \begin{cases} \mathcal{S}, & \text{if } \alpha = 0 \\ \| \phi\{U^{\alpha'}/Z\} \| \eta\{P_1/V_1, \ldots, P_k/V_k\}, & \text{if } \alpha = \alpha' + 1 \\ \bigcap\{\| U^{\alpha'}(P_1, \ldots, P_k) \| \eta \mid \alpha' < \alpha\}, & \text{if } \alpha \text{ limit ord.} \end{cases}$$

Dually, if $\sigma = \mu$:

$$\| U^\alpha(P_1, \ldots, P_k) \| \eta = \begin{cases} \emptyset, & \text{if } \alpha = 0 \\ \| \phi\{U^{\alpha'}/Z\} \| \eta\{P_1/V_1, \ldots, P_k/V_k\}, & \text{if } \alpha = \alpha' + 1 \\ \bigcup\{\| U^{\alpha'}(P_1, \ldots, P_k) \| \eta \mid \alpha' < \alpha\}, & \text{if } \alpha \text{ limit ord.} \end{cases}$$

We get the following basic monotonicity properties of ordinal approximations:

Proposition 1. *Suppose that $\alpha \leq \alpha'$.*

1. *If U is a greatest fixed point abstraction then*

$$\| U^{\alpha'}(P_1, \ldots, P_n) \| \eta \subseteq \| U^\alpha(P_1, \ldots, P_n) \| \eta$$

2. *If U is a least fixed point abstraction then*

$$\| U^\alpha(P_1, \ldots, P_n) \| \eta \subseteq \| U^{\alpha'}(P_1, \ldots, P_n) \| \eta$$

Proof. By wellfounded induction. □

Moreover, and most importantly, we get the following straightforward application of the well-known Knaster-Tarski fixed point theorem.

Theorem 2 (Knaster-Tarski). *Suppose that $U = \sigma Z(V_1, \ldots, V_k).\phi$. Then*

$$\| U(P_1, \ldots, P_n) \| \eta = \begin{cases} \cap\{\| U^\alpha(P_1, \ldots, P_n) \| \eta \mid \alpha \text{ an ordinal}\}, & \text{if } \sigma = \nu \\ \cup\{\| U^\alpha(P_1, \ldots, P_n) \| \eta \mid \alpha \text{ an ordinal}\}, & \text{if } \sigma = \mu \end{cases}$$

As the intended model is countable the quantification in theorem 2 can be restricted to countable ordinals.

The main rules to reason locally about fixed point formulas are the unfolding rules. These come in four flavours, according to whether the fixed point abstraction concerned has already been approximated or not, and to the nature and position of the fixed point relative to the turnstile.

$$\mathsf{ApprxL}\ \frac{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash \Delta}{\Gamma, s : U(P_1, \ldots, P_n) \vdash \Delta}\ U \text{ lfp},\ \kappa \text{ fresh}$$

$$\mathsf{ApprxR}\ \frac{\Gamma \vdash s : U^{\kappa}(P_1, \ldots, P_n), \Delta}{\Gamma \vdash s : U(P_1, \ldots, P_n), \Delta}\ U \text{ gfp},\ \kappa \text{ fresh}$$

$$\mathsf{UnfL1}\ \frac{\Gamma, s : \phi\{U/Z, P_1/V_1, \ldots, P_n/V_n\} \vdash \Delta}{\Gamma, s : U(P_1, \ldots, P_n) \vdash \Delta}\ U = \sigma Z(V_1, \ldots, V_n).\phi$$

$$\mathsf{UnfR1}\ \frac{\Gamma \vdash s : \phi\{U/Z, P_1/V_1, \ldots, P_n/V_n\}, \Delta}{\Gamma \vdash s : U(P_1, \ldots, P_n), \Delta}\ U = \sigma Z(V_1, \ldots, V_n).\phi$$

$$\mathsf{UnfL2}\ \frac{\Gamma, s : \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \kappa_1 < \kappa \vdash \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash \Delta}\ U \text{ lfp}, \kappa_1 \text{ fresh}$$

$$\mathsf{UnfR2}\ \frac{\Gamma, \kappa_1 < \kappa \vdash s : \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \Delta}{\Gamma \vdash s : U^{\kappa}(P_1, \ldots, P_n), \Delta}\ U \text{ gfp}, \kappa_1 \text{ fresh}$$

$$\mathsf{UnfL3}\ \frac{\Gamma, s : \kappa_1 < \kappa \supset \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\} \vdash \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash \Delta}\ U \text{ gfp}$$

$$\mathsf{UnfR3}\ \frac{\Gamma \vdash s : \kappa_1 < \kappa \wedge \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \Delta}{\Gamma \vdash s : U^{\kappa}(P_1, \ldots, P_n), \Delta}\ U \text{ lfp}$$

Normally we would expect only least fixed point formula abstractions to appear in approximated form to the left of the turnstile (and dually for greatest fixed points). However, ordinal variables can "migrate" from one side of the turnstile to the other through one of the cut rules. Consider for instance the following application of the process cut rule:

$$\frac{\Gamma \vdash s_2 : U^{\kappa} \quad \Gamma, S : U^{\kappa} \vdash s_1 : U^{\kappa}}{\Gamma \vdash s_1\{s_2/S\} : U^{\kappa}}$$

In this example U may be a greatest fixed point formula which, through some earlier application of ApprxR has been assigned the ordinal variable κ. The second antecedent has U^{κ} occurring to the left of the turnstile.

In addition to the above 8 rules it is useful also to add versions of the identity rules reflecting the monotonicity properties of ordinal approximations, prop. 1:

$$\mathsf{IdMon1}\ \frac{\Gamma \vdash \kappa \leq \kappa', \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash s : U^{\kappa'}(P_1, \ldots, P_n), \Delta}\ U \text{ lfp}$$

$$\mathsf{IdMon2}\ \frac{\Gamma \vdash \kappa \geq \kappa', \Delta}{\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \vdash s : U^{\kappa'}(P_1, \ldots, P_n), \Delta}\ U \text{ gfp}$$

Additionally a set of elementary rules are needed to support reasoning about well-orderings, including transitivity and irreflexivity of $<$. These rules are left out of the presentation.

For the above rules we obtain the following basic soundness result:

Theorem 3. *The rules* ApprxL, ApprxR, UnfL1, UnfR1, UnfL2 *and* UnfR2 *are sound.*

Proof. Rules ApprxL and ApprxR. For ApprxL assume $\Gamma, s : U^{\kappa}(P_1, \ldots, P_n) \models_{K,\kappa} \Delta$, and that κ is fresh. Assume also that Γ and $s : U(P_1, \ldots, P_n)$ holds, up to some valuation. Then, for this valuation, so does $s : U^{\alpha}(P_1, \ldots, P_n)$ for some ordinal α. But then we find that some assertion in Δ is true as well, completing the case. For ApprxR the dual argument applies.
Rules UnfL1 and UnfL2. The soundness of these rules follows directly from the fact that $\sigma Z(V_1, \ldots, V_n).\phi$ is a parametrised fixed point of ϕ.
Rules UnfL2 and UnfR2. We consider UnfL2. Assume that

$$\Gamma, s : \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\}, \kappa_1 < \kappa \models \Delta.$$

Assume also that U is a least fixed point abstraction, and that κ_1 is fresh. Assume furthermore that a valuation is given, making Γ and $U^{\alpha}(P_1, \ldots, P_n)$ true. Either α is 0, or $\alpha = \alpha_1 + 1$, or α is a limit ordinal. The first case is contradictory. For the second case we get the κ_1 we are looking for directly, and some assertion in Δ is established as desired. For the third case we find some $\alpha_1' < \alpha$ such that $U^{\alpha_1'}(P_1, \ldots, P_n)$ is true. We can assume that α_1' is a successor ordinal. But then the previous subcase applies, and we are done. Again UnfR2 is proved by a symmetric argument.
Rules UnfL3 and UnfR3. We consider UnfL3. Assume that

$$\Gamma, s : \kappa_1 < \kappa \supset \phi\{U^{\kappa_1}/Z, P_1/V_1, \ldots, P_n/V_n\} \models \Delta.$$

Assume also that a valuation is given such that Γ and $s : U^{\alpha}(P_1, \ldots, P_n)$ is true. Then whenever $\alpha_1 < \alpha$, $s : \phi\{U^{\alpha_1}/Z, P_1/V_1, \ldots, P_n/V_n\}$ is true as well. If $\alpha = 0$ this is trivially so. If α is a successor ordinal it follows by prop. 1, and if α is a limit ordinal we know that whenever $\alpha_1' < \alpha$ then $s : U^{\alpha_1'+1}(P_1, \ldots, P_n)$, so $s : \phi\{U^{\alpha_1}/Z, P_1/V_1, \ldots, P_n/V_n\}$. In any case we can conclude that some assertion in Δ must be true, finishing the argument. Again UnfR3 is symmetric.
Rules IdMon1 and IdMon2 are trivial, given 1. □

Discharge: Some Intuition The fundamental problem in arriving at a sound, yet powerful, rule of discharge, is to control the way fixed points may interfere as proofs are elaborated. We illustrate the problem by two examples.

Example 3. Consider the proof goal

$$S : \nu Z_1.\mu Z_2.[]Z_1 \wedge \forall P, V.[P!V]Z_2 \ \vdash\ S : \mu Z_3.\nu Z_4.[]Z_4 \wedge \forall P, V.[P!V]Z_3 \quad (14)$$

The assumption states that any infinite sequence of internal or send transitions can only contain a finite number of consecutive send transitions, while the assertion states that any infinite sequence of internal or send transitions can only contain a finite number of send transitions. Thus (14) is false.

Let us introduce the following abbreviations:

$$U_1 = \nu Z_1.\mu Z_2.[\,]Z_1 \wedge \forall P, V.[P!V]Z_2$$
$$U_2 = \mu Z_2.[\,]U_1 \wedge \forall P, V.[P!V]Z_2$$
$$U_3 = \mu Z_3.\nu Z_4.[\,]Z_4 \wedge \forall P, V.[P!V]Z_3$$
$$U_4 = \nu Z_4.[\,]Z_4 \wedge \forall P, V.[P!V]U_3$$

We start by refining (14) to the subgoal

$$S : U_2^{\kappa_2} \;\vdash\; S : U_4^{\kappa_4} \tag{15}$$

using the rules UnfL1, UnfR1, ApprxL and ApprxR. Continuing a few steps further (by unfolding the fixed point formulas and treating the conjunctions on the left and on the right) we obtain the two subgoals

$$S : [\,]U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \;\vdash\; S : [\,]U_4^{\kappa_4'} \tag{16}$$
$$S : [\,]U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \;\vdash\; S : \forall P, V.[P!V]U_3 \tag{17}$$

Subgoal 16 is refined via rule Mon2 to

$$S' : U_1, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \;\vdash\; S' : U_4^{\kappa_4'} \tag{18}$$

and after unfolding U_1 using UnfL1 we arrive at

$$S' : U_2, S : \forall P, V.[P!V]U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \;\vdash\; S' : U_4^{\kappa_4'} \tag{19}$$

which sequent one might expect to be able to discharge against (15) by coinduction in κ_4. By the same token when we refine (17) to

$$S : [\,]U_1, S' : U_2^{\kappa_2'}, \kappa_2' < \kappa_2, \kappa_4' < \kappa_4 \;\vdash\; S' : U_4 \tag{20}$$

we would expect to be able to discharge against (15) inductively in κ_2. This does not work, however, since derivation of (19) from (15) fails to preserve the induction variable κ_2 needed for (20), and vice versa, κ_4 is not preserved along the path from (15) to (20). Therefore, the infinite proof structure resulting from an infinite repetition of the above steps contains paths in which neither of the two variables is actually being preserved and decremented infinitely many times, and hence the attempted ordinal induction fails. It would still have been sound to discharge if at least one of the two ordinal variables had been preserved in the corresponding other branch; then there would have been no such paths.

Example 4. Consider the (reversed) proof goal

$$S : \mu Z_1.\nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]Z_1 \;\vdash\; S : \nu Z_3.\mu Z_4.[\,]Z_3 \wedge \forall P, V.[P!V]Z_4 \quad (21)$$

stating that if all infinite sequences of internal or send transitions of a process can only contain a finite number of send transitions, then these infinite sequences of internal or send transitions can only contain finite sequences of consecutive send transitions. This goal is obviously valid.

The abbreviations we shall use are:

$$\begin{aligned} U_1 &= \mu Z_1.\nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]Z_1 \\ U_2 &= \nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]U_1^{\kappa_1'} \\ U_3 &= \nu Z_3.\mu Z_4.[\,]Z_3 \wedge \forall P, V.[P!V]Z_4 \\ U_4 &= \mu Z_4.[\,]U_3^{\kappa_3'} \wedge \forall P, V.[P!V]Z_4 \end{aligned}$$

First we apply rules ApprxL, ApprxR, UnfL2 and UnfR2 to reduce (21) to the subgoal

$$S : U_2, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \;\vdash\; S : U_4 \quad (22)$$

Continuing in much the same way as in the preceding example we arrive at the two subgoals

$$S' : U_2, S : \forall P, V.[P!V]U_1^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \;\vdash\; S' : U_3^{\kappa_3'} \quad (23)$$

$$S : [\,]U_2, S' : U_1^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3 \;\vdash\; S' : U_4 \quad (24)$$

These subgoals are refined, using UnfR2 and UnfL2 respectively, to

$$\begin{aligned} &S' : U_2, S : \forall P, V.[P!V]U_1^{\kappa_1'}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3, \kappa_3'' < \kappa_3' \\ &\vdash S' : \mu Z_4.[\,]U_3^{\kappa_3''} \wedge \forall P, V.[P!V]Z_4 \quad (25) \\ &S : [\,]U_2, S' : \nu Z_2.[\,]Z_2 \wedge \forall P, V.[P!V]U_1^{\kappa_1''}, \kappa_1' < \kappa_1, \kappa_3' < \kappa_3, \kappa_1'' < \kappa_1' \\ &\vdash S' : U_4 \quad (26) \end{aligned}$$

These sequents can be discharged against (22) inductively in κ_3, and coinductively in κ_1, respectively. In contrast with the previous example, here every ordinal variable which is used for induction (or coinduction) in one of the two leaves is preserved throughout the path to the other leaf.

The Rule of Discharge We now arrive at the formal definition of the rule of discharge.

Convention 7. From this point onwards proof elaboration takes place in the context of some fixed, but arbitrary linear ordering $<$ on fixed point formula abstractions U.

Assuming one fixed linear ordering can be too restrictive when recursive proof structures are independent. Below we briefly discuss ways of relaxing the construction to allow the linear ordering to be built incrementally.

Below we define the critical notions of regeneration, progress, and discharge. Discharge is applied when facing a proof goal π_n which is unelaborated, such that, below π_n we find some already elaborated node π_1 such that π_n is in a sense an instance of π_1. This requires variables present in π_1 to be interpreted as terms in π_n. This is what the substitution ρ of the following definition serves to achieve.

Definition 8 (Regeneration, Progress, Discharge). *Let $\Pi = \pi_1, \ldots, \pi_n$ be a path such that π_n is not elaborated. Suppose that π_i is labelled by $\Gamma_i \vdash \Delta_i$ for all $i : 1 \leq i \leq n$.*

1. *The path Π is* regenerative *for U and the substitution ρ, if whenever there is a κ_i such that U^{κ_i} is a subformula of Γ_i (Δ_i) then there also are $\kappa_1, \ldots, \kappa_{i-1}, \kappa_{i+1}, \ldots, \kappa_n$ such that for all $j : 1 < j \leq n$, U^{κ_j} is a subformula of Γ_j (Δ_j), and $\Gamma_j \vdash \kappa_j \leq \kappa_{j-1}$. Moreover we require that $\rho(\kappa_1) = \kappa_n$.*
2. *The path Π is* progressive *for U and ρ if we can find $\kappa_1, \ldots, \kappa_n$ such that:*
   (a) *For all $i : 1 < i \leq n$, U^{κ_i} is a subformula of Γ_i (Δ_i), and $\Gamma_i \vdash \kappa_i \leq \kappa_{i-1}$.*
   (b) *$\rho(\kappa_1) = \kappa_n$.*
   (c) *For some $i : 1 < i \leq n$, $\Gamma_i \vdash \kappa_i < \kappa_{i-1}$.*
3. *The node π_n can be* discharged *against the node π_1 if we can find some U and substitution ρ such that:*
   (a) *Π is regenerative for all $U' < U$ and ρ.*
   (b) *Π is progressive for U and ρ.*
   (c) *For all assumptions $s : \phi$ in Γ_1, $\Gamma_n \vdash s\rho : \phi\rho$, and all assertions $s : \phi$ in Δ_1 then $s\rho : \phi\rho \vdash \Delta_n$.*

   In this case we term π_n a discharge node *and π_1 its* companion node.

In this definition we are being slightly sloppy with our use of U's: Really we are identifying fixed point formula abstractions up to ordinal approximations except where they are explicitly stated.

It is quite easy to verify that for Example 3 no linearisation of the fixed point formulas can be devised such that the nodes (18) and (19) can be discharged. On the other hand, for Example 4, any linear ordering which (up to approximation ordinals) has $U_4 < U_2$ will do.

Observe that the linear ordering on fixed point formula abstractions can be chosen quite freely. One might expect some correlation between position in the linear ordering and depth of alternation, viz. example 4 above. In practice this is in fact a good guide to choosing a suitable linear ordering. However, as we show, we do not need to require such a correlation a priori. Moreover one can construct examples, using cut's, of proofs for which the above rule of thumb does not work.

Now, the *full proof system* is obtained by adding the proof rules for fixed points, including the rule of discharge, to the local rules of section 4.

Theorem 4 (Soundness, Recursive Formulas). *The full proof system is sound.*

Proof. The proof is by induction on the size of proof trees. Assume a proof of root sequent $\Gamma_0 \vdash \Delta_0$, and assume soundness for all proof trees of a strictly smaller size. Assume for a contradiction that $\Gamma_0 \not\models \Delta_0$. We then find a valuation η_0 which invalidates $\Gamma_0 \not\models \Delta_0$, i.e. which makes all assumptions in Γ_0 valid, and all assertions in Δ_0 invalid. We use this assumption to construct an infinite *rejection sequence* Σ of sequent-valuation pairs $(\Gamma_0 \vdash \Delta_0, \eta_0)(\Gamma_1 \vdash \Delta_1, \eta_1) \cdots$ such that for all i, η_i invalidates $\Gamma_i \vdash \Delta_i$, and, considering nodes of discharge to be followed by their respective companion nodes, the sequent sequence $(\Gamma_0 \vdash \Delta_0)(\Gamma_1 \vdash \Delta_1) \cdots$ forms a run through the proof tree. Subsequently we use this sequence to derive a contradiction.

The sequence Σ is constructed inductively. Assuming that the construction has reached the i'th element we show how to construct the $(i+1)$'th element depending on the rule by which $\Gamma_i \vdash \Delta_i$ was elaborated in the proof. For all rules except discharge the construction is trivial, by the "local" soundness results, Theorems 1 and 3. So assume that $\Gamma_i \vdash \Delta_i$ was discharged against $\Gamma_i' \vdash \Delta_i'$ using substitution ρ as specified by the definition of the discharge rule. We then define the $(i+1)$'st element of the sequence as $(\Gamma_i' \vdash \Delta_i', \eta_i \circ \rho)$ and show that $\eta_{i+1} = \eta_i \circ \rho$ invalidates $\Gamma_i' \vdash \Delta_i'$. Let $s : \phi$ be any assertion in Γ_i'. By the induction hypothesis and condition 8.3.(c) we see that since all assumptions in Γ_i are validated under η_i then so is $s : \phi$ under η_{i+1}. Secondly let $s : \phi$ be any assertion in Δ_i'. We need to show that $s\eta_{i+1} : \phi\eta_{i+1}$ is false. But if it were not, by the induction hypothesis and condition 8.3.(c) we would obtain some assertion in Δ_i which is valid under η_i, and this is an impossibility since η_i invalidates $\Gamma_i \vdash \Delta_i$. The construction is thus complete.

Infinitely often along Σ the discharge rule is applied. The proof being finite, the number of distinct fixed point abstractions that can appear in the proof is finite too. As a consequence we must be able to find a smallest U under $<$ which is appealed to infinitely often (in 8.3) in applications of discharge along Σ. Let i be such that $\Gamma_i \vdash \Delta_i$ is elaborated infinitely often through the rule of discharge by appealing to U, and that, for no $j \geq i$, is $\Gamma_j \vdash \Delta_j$ discharged with reference to a U' which is strictly smaller than U. For some κ_i we find an occurrence of U^{κ_i} in the corresponding sequent $\Gamma_i \vdash \Delta_i$, say that U^{κ_i} is a subformula of Γ_i. We then see that for each $j > i$ we can find an ordinal variable κ_j such that U^{κ_j} occurs as a subformula of Γ_j. We shall sketch an argument that the subformulas U^{κ_j} can be chosen so that the values assigned to κ_j by η_j form a sequence which is non-increasing and in fact infinitely often decreasing. But this is not possible, since ordinals are well-founded, and we hence shall arrive at a contradiction.

Consider an arbitrary interval $\Sigma(j_1, j_m)$ of Σ such that the first sequent $\Gamma_{j_1} \vdash \Delta_{j_1}$ is equal to the last $\Gamma_{j_m} \vdash \Delta_{j_m}$ and does not occur inbetween. Then there must be an element in the interval whose sequent is a discharge node, and whose companion node is either $\Gamma_{j_m} \vdash \Delta_{j_m}$ or is some sequent higher in the proof tree (i.e. closer to the root sequent). We shall call the earliest such element the *characterising element* of the interval. The interval itself might contain other

intervals of the same shape. Moreover, one can choose these intervals in such a way that the elements not occurring in any of the intervals, the characterizing element being among them, form a simple run (i.e. a run not visiting any sequent more than once) through the loop defined by the discharge path for the characterising sequent. Given an initial partitioning of Σ into such intervals, one can iteratively apply this decomposition scheme until no interval can be further decomposed. Given an index j, we shall call the *active* interval the least interval of the above type containing both the j'th and the $j+1$'th element of Σ.

We perform an initial partitioning of Σ in intervals $\Sigma(j_1, j_m)$ so that $\Gamma_{j_1} \vdash \Delta_{j_1}$ is the companion node of $\Gamma_i \vdash \Delta_i$, and continue the decomposition process as described above. Starting from index $j = i$, we shall choose κ_j according to the discharge condition for the path having as a discharge node the sequent from the characterising element of the active interval. If the current interval is characterised by $\Gamma_i \vdash \Delta_i$ (which can only happen in outermost intervals) we choose κ_j as for progression (cf. 8.2), in all other cases we choose κ_j for regeneration (cf. 8.1). The discharge condition and the induction hypothesis guarantee that the values assigned to κ_j by η_j form a sequence which is non-increasing (in regenerative intervals) and infinitely often decreasing (in progressive intervals), thus yielding a contradiction.

So, no such rejection sequence Σ can exist, and the assumption $\Gamma_0 \not\vdash \Delta_0$ must have been false.

□

7 Verifying the Resource Manager

In this section the proof system is demonstrated by outlining a proof that the resource manager function introduced in section 2 satisfies the *safe* specification defined in section 3. The proof will be kept informal. For instance we will write out neither ordinal variables nor the linear ordering on fixed point formula abstractions, since they can easily be added to the proof. Adding ordinal annotations to the proof and taking them into account presents no real difficulty since the fixed point definitions in the example are flat, i.e., they never refer to other fixed point definitions.

For simplicity it is assumed that the manager knows of only one resource, with public name P_u and private P_r. The corresponding list $[\{P_u, P_r\}]$ is referred to as R_L, and R_P denotes the process identifier of the resource manager process.

Since the definition of *safe* is parametrised on a billing agent and a user account the formula must be preceded by an initialisation phase (notice the use of the weak modality $[[\alpha]]$ introduced in section 3):

$$\forall PubRes, UAcc, UserPid, Agent.$$
$$[R_P?\{contract, \{PubRes, UAcc\}, UserPid\}]$$
$$[[UserPid!\{contract_ok, Agent\}]]safe(Agent, BankPid, UAcc, 0)$$

So we set out to prove the following sequent:

$$\Gamma \vdash \langle rm(R_L, BankPid, RAcc), R_P, \epsilon \rangle$$

$$: \forall PubRes, UAcc, UserPid, Agent.$$
$$[R_P?\{contract, \{PubRes, UAcc\}, UserPid\}] \ldots \quad (27)$$

The needed inequations on process identifiers (e.g., $R_P \neq P_r$) are collected in Γ. By application of simple proof steps – four applications of A||R and then repeated applications of the rules for unfolding, elimination of conjunctions, and the rule for the box modality – the following proof state is reached:

$$\Gamma' \vdash \langle rm(R_L, BankPid, RAcc), R_P, \epsilon \rangle$$
$$|| \langle billagent(P_r, BankPid, RAcc, UAcc), B_P, \epsilon \rangle$$
$$: safe(B_P, BankPid, UAcc, 0) \quad (28)$$

where Γ' is Γ extended with the fact that B_P is a fresh process identifier. This is a critical proof state, where we must come up with properties of the resource manager and the billing agent, that are sufficiently strong to prove that their parallel composition satisfies the *safe* property. In general such a proof step may be very difficult, but here the choice is relatively simple:

ϕ_a: The billing agent satisfies the *safe* property, i.e., $safe(B_P, BankPid, UAcc, 0)$.
ϕ_b: The billing agent communicates the user account to no process except the bank, unless some other process first sends it the account.
ϕ_c: The resource manager does not communicate the user account, unless some other process first sends it the account. This property can be formulated as $notrans(UAcc)$ given the definition of the *notrans* property at the end of section 3.
ϕ_d: The resource manager does not send a tuple containing the atom *use* in the first position (a usage request to a billing agent).
ϕ_e: The resource manager cannot receive messages sent to the bank process, nor can it receive messages sent to the billing agent.

Properties ϕ_b, ϕ_d and ϕ_e can easily be formulated in a manner similar to ϕ_c. Essentially these conditions guarantee that bank transfers are the result of user requests, rather than incorrectly programmed billing agents or resource managers that exchange information with each other.

The result of applying the ProcCut rule twice, after generalising the proof goals, is the following proof obligations:

$$\Gamma', \mathtt{not}(contains(B_Q, UAcc)), countuse(B_Q, M), M \leq N \vdash$$
$$\langle billagent(P_r, BankPid, RAcc, UAcc), B_P, B_Q \rangle$$
$$: safe(B_P, BankPid, UAcc, N) \wedge \phi_b \quad (29)$$
$$\Gamma', \mathtt{not}(contains(R_Q, UAcc)) \vdash \langle rm(R_L, BankPid, RAcc), R_P, R_Q \rangle$$
$$: notrans(UAcc) \wedge \phi_d \wedge \phi_e \quad (30)$$
$$\Gamma', S_1 : safe(B_P, BankPid, UAcc, N) \wedge \phi_b, S_2 : notrans(UAcc) \wedge \phi_d \wedge \phi_e$$
$$: S_1 || S_2 : safe(B_P, BankPid, UAcc, N) \quad (31)$$

To prove the leftmost conjunct in the goal (29) one has to show that the number of valid usage requests in the input queue (the parameter M in $countuse(B_Q, M)$) is always less than or equal to the number of transfer requests that are possible (the parameter N). This proof involves well-known techniques for proving correctness of sequential programs, and the proof of ϕ_b is even less involved (proofs omitted). Instead we concentrate on the leftmost conjunct of (30), i.e., that rm satisfies $notrans(UAcc)$ as long as no element in its input queue contains $UAcc$. The proofs of properties ϕ_d and ϕ_e follow the same pattern (details omitted). To prove (30) we first unfold the definition of $notrans$ and eliminate the conjunctions. In case of an input step $[V?V']$ either we are done immediately (if $contains(V', UAcc)$). Otherwise the resulting proof state is

$$\Gamma', \mathtt{not}(contains(R_Q, UAcc)), \mathtt{not}(contains(V', UAcc)) \vdash \langle rm(R_L, BankPid, RAcc), R_P, R_Q \cdot V' \rangle : notrans(UAcc) \quad (32)$$

which can be rewritten into (by referring to the definition of $contains$)

$$\Gamma', \mathtt{not}(contains(R_Q \cdot V', UAcc)) \vdash \langle rm(R_L, BankPid, RAcc), R_P, R_Q \cdot V' \rangle : notrans(UAcc) \quad (33)$$

which can be discharged against the leftmost conjunct of (30). The rm process can clearly not perform any output step so that part of the conjunction is trivially true. Thus only the internal step remains, and such a step must correspond to unfolding the application $rm(R_L, BankPid, RAcc)$. The resulting proof state is:

$$\Gamma', \mathtt{not}(contains(R_Q, UAcc)) \vdash \langle \mathtt{case}\ \{R_L, BankPid, RAcc\}\ \mathtt{of}\ \ldots, R_P, R_Q \rangle : notrans(UAcc) \quad (34)$$

By repeating the above steps, i.e., handling input, output and internal steps eventually one reaches the goal:

$$\Gamma'', \mathtt{not}(contains({R_Q}', UAcc)) \vdash \big\langle UserPid!\{contract_ok, B_p'\}, rm(R_L, BankPid, RAcc) \ldots, R_P, {R_Q}' \big\rangle \parallel \big\langle billagent(P_r, BankPid, RAcc, UAcc'), {B_P}', \epsilon \big\rangle : notrans(UAcc) \quad (35)$$

where Γ'' is Γ' together with inequations involving the fresh process identifier ${B_P}'$, and the fact that $UAcc' \neq UAcc$. This goal is handled by applying ProcCut to the parallel composition using $notrans(UAcc)$ as the cut formula both to the left and to the right. The resulting goals are:

$$\Gamma'', \mathtt{not}(contains({R_Q}', UAcc)) \vdash \big\langle UserPid!\{contract_ok, {B_P}'\}, rm(R_L, BankPid, RAcc) \ldots, R_P, {R_Q}' \big\rangle : notrans(UAcc) \quad (36)$$

$$\Gamma'' \vdash \big\langle billagent(P_r, BankPid, RAcc, UAcc'), {B_P}', \epsilon \big\rangle : notrans(UAcc) \quad (37)$$

$$\Gamma'', S_3 : notrans(UAcc), S_4 : notrans(UAcc) \vdash S_3 \parallel S_4 : notrans(UAcc) \quad (38)$$

Goal (37) is easy to prove, since no new processes are created (proof sketch omitted). For goal (36) we have to show $\mathtt{not}(contains(\{contract_ok, B'_p\}, UAcc))$, since this is the value the resource manager will send to pid $UserPid$. The property is clearly true since B'_P is a fresh pid. The resulting goal, after a simple step where the resulting sequence is reduced, becomes

$$\Gamma'', \mathtt{not}(contains(R_Q{}', UAcc)) \vdash \langle rm(R_L, BankPid, RAcc), R_P, R_Q{}' \rangle : notrans(UAcc) \quad (39)$$

This goal can be discharged against the leftmost conjunct of (30). Thus only goals (31) and (38) remain. These types of goals are handled in a uniform and regular way, using applications of BoxPar1 and BoxPar2, repeated use of boolean reasoning, the UnfR and UnfL rules, and discharging against previously seen goals (details omitted).

8 Concluding Remarks

We have introduced a specification logic and proof system for the verification of programs in a core fragment of Erlang, and illustrated its application on a small, but quite delicate, agent-based example. Our approach is quite general both regarding the kinds of languages and models that can be addressed, and the kinds of assertions that can be formulated. For instance we are not restricted, as in many other approaches to compositional verification, to linear-time logic, neither does the proof system rely on auxiliary features like history or prophecy variables. In addition our approach permits the treatment of programming language constructs such as dynamic process creation, non-tail recursion and inductive data type definitions in a uniform way, via a powerful rule of discharge.

An important feature of our approach is the use of fixed points to describe recursively the fine structure of computation trees, and to use these recursive descriptions to decompose properties according to system structure. No fixed vocabulary of temporal connectives such as those of LTL, CTL, or CTL* would permit a similarly general decomposition. The proof-theoretical setting given here represents a substantial advance on the initial work for CCS reported in [Dam98]. That work suffered from a number of shortcomings which we think have now been resolved in a satisfactory manner. This concerns:

1. The account of discharge in [Dam98] used an indirect approach, tracking and indexing fixed point unfoldings in a very syntactical and opaque manner. The present approach, using explicit ordinal annotations, is arguably far simpler, more intuitive, and semantically clearer.
2. The sequent format used in [Dam98] was more restrictive than the one used here, in effect preventing contraction, affecting proof power very severely, theoretically as well as in practice.
3. The discharge condition of [Dam98] required much more rigid relationships between the structure of discharged nodes and the internal nodes motivating their discharge. In effect it was required that all information be completely

cyclic in a pointwise manner. But many examples are extremely cumbersome, if not outright impossible, to force into such a framework.

The drawback, if any, of the approach used here is the explicit use of ordinals. However, in an implementation of this proof system, users need rarely, if ever, be directly exposed to ordinals. Ordinal annotations can be automatically synthesised, and only in very special circumstances do we envisage ordinal information being passed explicitly to users for proof debugging.

Several important lessons were learned in the process of doing proofs like the billing agent example. We have already mentioned the need for more flexible sequent formats and discharge conditions. Practical proofs tend to get very large. Without support for reducing duplication of proof nodes the proof example outlined for the billing agent has in the range of 10^5–10^6 proof tree nodes. Just by avoiding proof node duplication this figure can be brought down very substantially, for the billing agent example by roughly a factor of 15. But in fact very few steps in the proof convey information which is really interesting. These are:

1. Points where a process cut need to be applied, to initiate induction in system state structure.
2. Points at which some other symbolic or inductive argument needs to be done, to handle e.g. induction in the message queue structure.
3. Choice points which we may want to return to later, for backtracking.
4. Points which we expect to want to discharge against in the future.

One can easily envisage other proof elaboration steps being automated, and eliminated from view to a very large extent, perhaps using a selection of problem-dependent proof tactics. However, it is important to realise that, in contrast to mainstream proof editors such as HOL or PVS, in this some explicit support for managing proof node histories is essential for efficiency.

To investigate these issues, and to begin doing real application studies, we are currently building a prototype proof checking tool that can handle programs of a moderate size such as the billing agent example. Some support for automation of proof steps along the above lines already exists (e.g. for some model checking analyses), but we also need to identify other classes of sequents that can be solved algorithmically. Other ongoing work focuses on integrating the operational semantics of Erlang more tightly with the proof systems (along the lines of [Sim95]) and to improve the handling of process identifier scoping (but see [AD96] for an approach to this in the context of the π-calculus).

Acknowledgements Many thanks are due to Fredrik Orava of the department of Teleinformatics at the Royal Institute of Technology, and Thomas Arts, Tony Rogvall and Dan Sahlin of Ericsson Telecom Computer Science Laboratory.

References

[AD96] R. Amadio and M. Dam. A modal theory of types for the π-calculus. In *Proc. FTRTFT'96,* Lecture Notes in Computer Science, 1135:347–365, 1996.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Functional Programming*, 7:1–72, 1997.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.

[Dam98] M. Dam. Proving properties of dynamic process networks. To appear, *Information and Computation*, 1998. Preliminary version as "Compositional Proof Systems for Model Checking Infinite State Processes", Proc. CONCUR'95, LNCS 962, pp. 12–26.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40 and 41–77, 1992.

[Par76] D. Park. Finiteness is mu-Ineffable. *Theoretical Computer Science*, **3**:173–181, 1976.

[Sim95] A. Simpson. Compositionality via cut-elimination: Hennessy-Milner logic for an arbitrary GSOS. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 420–430, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

A Compositional Real-time Semantics of STATEMATE Designs

Werner Damm[1], Bernhard Josko[1], Hardi Hungar[1], and Amir Pnueli[2]

[1] OFFIS, Oldenburg, Germany
[2] Weizmann Institute, Rehovot, Israel

1 Introduction

This paper presents a reference semantics for a verification tool currently under development allowing to verify temporal properties of embedded control systems modelled using the STATEMATE system. The semantics reported differs from others reported in the literature [24] by faithfully modelling the semantics as supported in the STATEMATE simulation tool. It differs from the recent paper by Harel and Naamad [8] by providing a *compositional* semantics, a prerequisite for the support of *compositional verification methods*, and by the degree of mathematical rigour. We use a variant of *synchronous transition systems* introduced by Manna and Pnueli [18] as base model for our semantics.

The STATEMATE modelling language constructs covered in this paper are *Activity charts*, modelling the functional decomposition of a design into subunits called *activities* as well as the *information flow* between these, and *Statecharts*, modelling reactive behaviour using the well established approach of hierarchically organized state-machines. We strive for a verification approach which is compositional w.r.t. the decomposition of systems into subsystems. This will allow activities of "reasonable" complexity to be verified using *symbolic model checking* [5, 4, 19]. Larger activities will be verified on the basis of proof-systems relating properties of individual activities to properties of compound activities, using the well known *assumption commitment paradigm* [1, 21, 15]. A key topic for this paper is the construction of so called *compositional models*, which are "rich enough" to model the STATEMATE parallel composition by intersection of the infinite traces generated by the components of the parallel composition. Roughly, compositional models have to provide room for padding arbitrary (but still "legal") environment interactions into computations of a component. Alternatively, the construction of compositional models can be phrased as a requirement on the model to support a sufficiently rich class of *observables* for assumption-commitment style reasoning to be complete. In this sense, this paper derives the set of atomic propositions included as observables in the assumption-commitment style temporal logic supported by the verification tool.

The richness of the STATEMATE modelling languages forbids a complete treatment within such a formal semantics. While [8] elaborate in a detailed fashion the construction of compound transitions from transition segments, we take this as given in this paper. We also abstract from the concrete syntax of

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 186-238, 1998.
Springer-Verlag Berlin Heidelberg 1998

action annotations, but keep them rich enough to show how all the associated intricacies can be handled formally.

Following these disclaimers, we now elaborate those aspects we do consider to be central.

This paper focuses on the so-called *asynchronous semantics* or *super-step semantics* supported by the STATEMATE simulation tool. Intuitively, a super-step consists of a (possibly unbounded) chain of reactions (called *steps*) of the system under development (henceforth abbreviated SUD) to an *external stimulus*. Activities of the SUD perform steps in *synchrony*: at a step-boundary, a maximal conflict-free set of enabled transitions is selected based on events generated in the previous step and valuations of (shared and local) variables and conditions, leading to a new valuation and a new set of generated events visible first after completion of the step. Four issues deserve attention already in this introduction.

- Steps may *diverge* due to the presence of *while-loops* within action annotations; our language reflects this by allowing not only event generation and assignments, but also divergence as effect of executing an action.
- The super-step semantics distinguishes between events generated by the *environment* of the SUD, and events generated *locally*. While all events persist *for one step only*[1], external events *are only consulted at the first step* and are communicated to the environment of the SUD first after completion of a super-step. Events generated internally by the SUD will be sensed in the next step. Indeed the distinction between externally and locally generated events is paramount for the definition of a super-step: it terminates, if no further steps can be taken on the basis of presence or absence of *locally* generated events. We pick up this in the formal development by distinguishing *slow* events (from the environment of the SUD) from *fast* events (generated locally). Note that a compositional semantics of an activity will have to address events generated outside from the considered activity, but within the SUD (*a fast event*) different from one that is generated not only outside the considered activity, but also outside the SUD (*a slow event*).
- STATEMATE supports the concept of *data items*, which subsumes the notion of variables, in particular retaining their value over step-boundaries even if unchanged. Based on scoping rules (and actual usage detected at compile time), variables can be *local* (to an activity) or *shared*. In this paper we abstract from the concrete syntax of declarations and replace this by a classification of variables as *local* and *shared*. *Shared variables* are assumed to have usage annotations as *in*, *out*, or *inout*, and are for reasons explained above additionally labelled as *slow* or *fast*. Clearly, *slow* variables may only change at super-step-boundaries, while *fast* variables may change from step to step. A key aspect of this paper is the *compositional treatment* of shared variables, allowing interferences between a locally computed value and the value suggested by the environment of an activity.

[1] More precisely, externally generated events persist until the completion of the first step in the subsequent super-step.

– StateMate supports a *discrete simulation time*, subsequently referred to simply as "the clock". In the asynchronous execution mode, the clock is only incremented at super-step boundaries (which requires global agreement between all activities of the SUD upon termination). In our model, which in contrast to a simulation tool clearly cannot allow interactions by user-commands (such as `GO-ADVANCE` or `GO-NEXT`) to prescribe how time is advanced, we increment time at super-step *boundaries to the next relevant clock tick*, determined by counters handling *time-outs* and *scheduled actions*. Again a key emphasis of this paper lies in the *compositional* treatment of real-time aspects of StateMate designs. The timing-model chosen classifies StateMate as a *synchronous* language [2]: super-steps are executed infinitely fast (in the discrete time scale in fact in zero-time) obeying the non-zeno property, since the clock will always be incremented at least by one time-unit at super-step boundaries.

The paper is organized as follows. Chapter 2 introduces the semantical model of synchronous transition systems and their runs. The semantics of StateMate will be given in terms of this model. Chapter 3 details the abstract syntax for the StateMate subset treated in this paper and defines the set of observables supported for a StateMate activity. It also collects the standard functions to navigate in the state-hierarchy as well as the concept of orthogonality of transitions. Chapter 4 handles the construction and execution of a single step for a single statechart, which is extended to super-steps in Chapter 5. Finally, Chapter 6 deals with the semantics of activity charts.

2 Synchronous Transition Systems

The semantics of statecharts will be given in terms of synchronous transition systems. We introduce a slight variant called *compositional synchronous transition systems(CSTS)*, which forms the basic for compositional reasoning.

A transition system is given by a set of system states and a set of transitions between system states. Instead of giving an explicit representations of a transition system we will use symbolic representations given by variable assignments which may be defined by formulae.

Given a (typed) set V of variables and a (typed) data domain $\mathcal{D}$, a *valuation*[2] w.r.t. V is a type preserving mapping $\sigma : V \to \mathcal{D}$. We will use $\Sigma(V)$ to denote the set of all valuations over the variables V. If V is obvious from the context we will use the notation Σ instead of $\Sigma(V)$. Given a subset V' of V the restriction of a valuation $\sigma \in \Sigma(V)$ to V', denoted by $\sigma_{|V'}$, is a valuation of $\Sigma(V')$ given by

$$\sigma_{|V'} : V' \Leftrightarrow \mathcal{D} : v \mapsto \sigma(v)$$

[2] In the context of transition systems, a valuation is usually called a state. But, to distinguish between a state of a statechart and a state of a transition system we call the latter one a valuation.

A *transition system*

$$\Phi = (V, \Theta, \rho)$$

is given by a set V of variables, a set $\Theta \subseteq \Sigma(V)$ of *initial valuations*, and a *transition relation* $\rho \subseteq \Sigma(V) \times \Sigma(V)$.

A *run* π of a transition system Φ is a finite or infinite sequence of valuations

$$\pi = \sigma_0 \ \rightarrow \ \sigma_1 \ \rightarrow \ \sigma_2 \ \rightarrow \ \ldots$$

with

- $\sigma_0 \in \Theta$ (A run starts with an initial valuation.)
- $\forall\, i \in \mathbb{N} : (\sigma_i, \sigma_{i+1}) \in \rho$ (Successor system states are given by the transition relation.)

The i-th valuation in a run π is also denoted by $\pi(i)$. An infinite run is also called a *computation*.

For the transition relation ρ we also use the notation $\sigma \rightarrow \sigma'$ or $\sigma \rightarrow_\rho \sigma'$ to denote that $(\sigma, \sigma') \in \rho$. $\rightarrow^+$ and $\rightarrow^*$ denote the transitive, resp. transitive and reflexive, closure of $\rightarrow$.

$$\mathit{reachable}(\sigma) := \{\sigma' \ \mid \ \sigma \rightarrow^* \sigma'\}$$

is the set of valuations reachable from σ and

$$\mathit{reachable}(\Phi) := \bigcup_{\sigma \in \Theta} \mathit{reachable}(\sigma)$$

is the set of reachable valuations of a transition system Φ.

We denote by $\mathit{run}(\Phi)$ the set of all runs and by $\mathit{Comp}(\Phi)$ the set of all computations of the transition system Φ, i.e.

$$\begin{aligned} \mathit{run}(\Phi) &= \{\pi \mid \pi \text{ run of } \Phi\} \\ \mathit{Comp}(\Phi) &= \{\pi \mid \pi \text{ computation of } \Phi\} \ . \end{aligned}$$

A transition system Φ is called *consistent* if $\mathit{Comp}(\Phi) \neq \emptyset$, i.e. if there is at least one computation. Φ is called *viable* if every run π can be extended to a computation π'. Hence a system is viable if every reachable valuation has at least one successor valuation.

2.1 Compositional Synchronous Transition Systems

In this general view there is no distinction between internal, external or shared variables. Given a set of externally visible variables we can specify the observable behaviour of a transition system by restricting its computations to the external visible variables. The distinction between externally observable and local variables is needed to describe the composition of transition systems.

Given a transition system Φ and a set $E \subseteq V$ of *externally observable* variables, a *trace* over E is an infinite sequence ν of valuations $\nu(i) \in \Sigma(E)$ such

that ν can be extended to a computation π of Φ. I.e. $\nu : \mathbb{N} \to E \to \mathcal{D}$ is a trace of Φ iff there exists a computation π of Φ s.t. $\pi|_E = \nu$, i.e. $\pi(i)(v) = \nu(i)(v)$ for all $i \in \mathbb{N}$ and $v \in E$. We define the set of *traces* w.r.t. E by

$$[\![\Phi]\!]_E = \{\nu \mid \exists \text{ computation } \pi \; s.t. \; \pi|_E = \nu\} \; .$$

A transition system with a distinguished set of externally visible variables,

$$\Phi = (V, \Theta, \rho, E),$$

will be called a *compositional synchronous transition system (CSTS)*. The variables of $V \setminus E$ are considered to be local, hence when composing two systems we assume that these internal variables are disjoint. This can always be achieved by renaming. Hence, assuming that the local variables of two transition systems are disjoint we define the composition by the intersection of the given transition systems.

Given two systems $\Phi_1 = (V^1, \Theta^1, \rho^1, E^1)$ and $\Phi_2 = (V^2, \Theta^2, \rho^2, E^2)$ with $(V^1 \setminus E^1) \cap V^2 = \emptyset$ and $(V^2 \setminus E^2) \cap V^1 = \emptyset$ we define their parallel composition by $\Phi = (V, \Theta, \rho, E)$ where

- $V = V^1 \cup V^2$
- $E = E^1 \cup E^2$
- $\Theta = \{\sigma \in \Sigma(V) \mid \sigma|_{V_1} \in \Theta^1 \text{ and } \sigma|_{V_2} \in \Theta^2\}$
- $\rho \subseteq \Sigma(V) \times \Sigma(V)$ is given by
  $(\sigma, \sigma') \in \rho$ iff $(\sigma|_{V_1}, \sigma'|_{V_1}) \in \rho^1$ and $(\sigma|_{V_2}, \sigma'|_{V_2}) \in \rho^2$

The composition will be denoted by $\Phi_1 \| \Phi_2$. If $(V^1 \setminus E^1) \cap V^2$ or $(V^2 \setminus E^2) \cap V^1$ are not empty we will first rename the variables in $(V^1 \setminus E^1)$ and $(V^2 \setminus E^2)$ before applying the composition.

This composition does in general not preserve viability and consistency. It may be the case that both, Φ_1 and Φ_2 are viable (consistent) but the composition is not. Observe that e.g. Θ may be empty even if both Θ^1 and Θ^2 are not empty. Or, one transition system may require that a valuation with $v = 1$ has to be followed by a valuation with $v = 2$ and the other system may demand that a valuation with $v = 1$ is followed by a valuation with $v = 3$. Hence a system state with $v = 1$ reached in the composed system will have no successor. In the modelling of statecharts we will not have such contradictory requirements in components. Our semantics will not introduce deadlocks in a composed system when there is no corresponding deadlock in one component. To achieve this, the semantics of one component will contain all observable behaviour of its environment. Semantical models satisfying this property are called *compositional*.

3 STATEMATE Designs: Key Concepts

This section introduces the abstract syntax for the sublanguage of statecharts considered in this paper. We first describe the state-based concepts of statecharts, then we will discuss the data concepts including events. Finally the execution concepts are given.

3.1 State-based Concepts

The state-oriented hierarchical modelling style of statecharts rests on the definition of a hierarchically structured set of states, inducing a *child* relation between certain states. States can be either BASIC, OR, or AND states. Clearly only BASIC states have no children. To be in an OR state means to be in exactly one of its *children*. The *children* of an AND state constitute its *components*. To be in an AND state means to be in *all* its components. We denote the set of states of SC by $states(SC)$. It comes equipped with a *child* function

$$child : states(SC) \leftrightarrows 2^{states(SC)} \ ,$$

a *mode* function

$$mode : states(SC) \leftrightarrows \{\text{BASIC}, \text{OR}, \text{AND}\} \ ,$$

and a function designating for each OR-state one of its children as the *default* child, which is entered, whenever the OR-state itself is entered:

$$default : \{s \mid mode(s) = \text{OR}\} \leftrightarrows states(SC) \ .$$

To be well-defined, such a hierarchical set of states, $states(SC)$, has to be a finite tree with a distinguished root state *root* with $mode(root) = \text{OR}$[3]. The function *child* gives the set of successors of a node in the tree.

If a state s is a substate of s', i.e. $s \in child(s')$, the state s' is also called the *father state* of s, written as $father(s)$.

The *depth* of a state w.r.t. the state hierarchy is inductively defined by

$$depth(s) \ := \ \begin{cases} 0 & \text{if } s = root \\ depth(father(s)) + 1 & \text{otherwise} \end{cases}$$

and the depth of a statechart SC is given by the maximal nesting of states:

$$depth(SC) \ := \ \max\{depth(s) \mid s \in states(SC)\} \ .$$

The function *child* defines a partial ordering on the set of states:

$$\begin{array}{l} s \leq s \\ s \in child(s') \ \text{ then } \ s' \leq s \\ s \leq s', \text{ and } s' \leq s'' \ \text{ then } \ s \leq s'' \\ s < s' \ \text{ iff } \ s \neq s' \text{ and } s \leq s' \ . \end{array}$$

The dynamic behaviour of statecharts stems from firing *transitions* or executing *static reactions* associated with states. We discuss transitions first.

The elementary concept of a transition between states has been elaborated to a complex object in the statechart context. We discuss below some of the added complexity, before offering the abstract syntax for the chosen sublanguage.

[3] In STATEMATE the root need not be an OR-state. But in that case the simulator will introduce an additional top state of mode OR.

- *intra-level transitions*

  Statecharts allow transitions to exit and enter *multiple levels* of the state hierarchy, requiring a precise analysis as to which states remain active and which are exited or entered.

- *multiple sources and targets*

  This analysis is further complicated by allowing multiple sources and targets, obeying certain well-formedness properties formalized below.

- *connectors and transition segments*

  In the graphical representation, a transition can be split into several *segments* linked via *connectors*. Each segment can have separate firing conditions, inducing a non-trivial analysis when constructing *compound transitions* out of transition segments. In this paper, we assume that this analysis has been carried out, and only retain the concept of history and termination connectors discussed below.

- *history connectors*

  Among the allowed *targets* of a transition are *history connectors*, which come in two versions. *Shallow* history connectors (hence referred to simply as "history connectors") memorize the unique child of an OR state active when last exiting this OR state. When entering an OR state with a history connector, the child entered is given by the substate stored in the history connector, rather than the *default* child. *Deep history connectors* not only memorize the previously active child of the associated OR state, but the complete *state configuration* below this state (see the following section for a formal definition of the intuitive concept of state configuration). Hence when entering an OR state s, the descendants to be activated are completely retrieved from the current value of the deep history connector. We will discuss below implicit events associated with history connectors.

- *termination connectors*

  Among the allowed targets of transitions are *termination connectors*, which when entered cause the activity associated with SC to become *inactive* and emit the event *stopped*(A) after completing the current step.

Additional concepts as the labelling of transitions with guards and actions will be discussed in Section 3.3. We now formalize the above concepts.

Connectors. We assume disjoint sets $Hconn(SC)$, $Dhconn(SC)$, and $Tconn(SC)$ of history connectors, deep history connectors and termination connectors, which jointly define the set $conn(SC)$ of connectors of SC. Recall that connectors are allowed to occur as targets of transitions. For each connector, there is a unique OR-state with which it is associated via the function

$$state : conn(SC) \leftrightarrow\!\!\!\!\rightarrow \{s \in states(SC) \mid mode(s) = \text{OR}\} \ .$$

Moreover, each OR-state is allowed to have *at most one* history and deep history connector[4]. We will sometimes make use of the inverse (partial) functions

$$hist : \{s \in states(SC) \mid mode(s) = \text{OR}\} \multimap Hconn(SC)$$
$$dhist : \{s \in states(SC) \mid mode(s) = \text{OR}\} \multimap Dhconn(SC) \ .$$

Transitions. We assume a set $trans(SC)$ of *transition names*. This comes equipped with a set of functions defining $source(s)$ and $target(s)$ associated with a transition. Sources and targets of a transition have to obey well-formedness restrictions defined below.

$source : trans(SC) \Leftrightarrow 2^{states(SC)}$
defines for a transition t a *nonempty* set of source states.

$target : trans(SC) \Leftrightarrow 2^{states(SC) \cup conn(SC)}$
defines for t a *non-empty* set of target states or connectors.

Next we will define a collection of auxiliary functions which will be used to define well-formedness conditions on statecharts and which are necessary to define the effect of firing a transition. We also introduce the concept of *configurations* describing maximal subset of states allowed to be concurrently active. We will first define the smallest region in which changes, due to the execution of a transition, may occur.

The *least common ancestor* $lca(S)$ of a non-empty set of states defines the closest state which subsumes all states of S. As the root is an ancestor of every state, $lca(S)$ will exist for every subset S of states. It is defined by

$$lca(S) \leq S \ (lca(S) \text{ is an ancestor of every state of } S) \text{ and}$$
$$\forall \hat{s} \in states(SC) \text{ with } \hat{s} \leq S : \hat{s} \leq lca(S) \ (lca(S) \text{ is minimal}) \ .$$

The *least common or-ancestor* $lca^+(S)$ of a non-empty set of states defines the smallest OR-state which subsumes all states of S and is not contained in S itself. If the least common ancestor is an OR-state not contained in S this is also the least common OR-ancestor, otherwise, we pick the closest OR-state above the least common ancestor. As we require that the root is an OR-state, the least common OR-ancestor exists for every subset of states not containing the root. If the root is contained in S the root will also be the least common OR-ancestor by definition. Hence the least common OR-ancestor is defined by

[4] The STATEMATE system allows more then one (deep) history connector for one OR-state. But this can be seen as syntactical sugar.

$$lca^+(S) := \begin{cases} root & \text{if } root \in S \\ s & \text{if } root \notin S \text{ and } s < S \text{ and } mode(s) = \text{OR and} \\ & \forall \hat{s} \in states(SC): \ mode(\hat{s}) = \text{OR and } \hat{s} < S \\ & \qquad \Rightarrow \ \hat{s} \leq s \end{cases}$$

Two states s and s' are *orthogonal*, denoted by $s \perp s'$, if they are in parallel states, i.e. they are not related w.r.t. the $child^*$ function and their common ancestor is an AND-state.

$$s \perp s' \quad \text{iff} \quad s \not\leq s' \text{ and } s' \not\leq s \text{ and } mode(lca(\{s, s'\})) = \text{AND}$$

A set S of states is called *orthogonal*, denoted by $\perp(S)$, if the states of S are pairwise orthogonal.

A set of states $S \subseteq states(SC)$ is called *consistent*, denoted by $\downarrow(S)$ iff every two states s, s' of S are either related by the *child* relation – $s \leq s'$ or $s' \leq s$ – or orthogonal.

A *state configuration* is a maximal consistent set of states.

The *scope* of a transition t, denoted by $scope(t)$, is the smallest range which is affected by firing the transition t. It is the OR-state obtained as the lca^+ of the source and target states of the transition. As the target may also contain some connectors, we have to replace these connectors by the corresponding OR-state to compute the common ancestor. Let

$$states : 2^{states(SC) \cup conn} \leftrightarrows 2^{states(SC)} : S \cup C \mapsto S \cup \{state(c) \mid c \in C\}$$

then

$$scope(t) := lca^+\big(source(t) \cup states(target(t))\big) \ .$$

Given a consistent set $S \subseteq states(SC)$ the *default completion* $dcompl(S)$ is the smallest set C such that

1. $S \subseteq C$,
2. $s \in C$ and $s \neq root$ then also $father(s) \in C$,
3. $s \in C$ and $mode(s) = \text{OR}$ and $child^+(s) \cap S = \emptyset$ then $default(s) \in C$,
4. $s \in C$ and $mode(s) = \text{AND}$ then $child(s) \subseteq C$.

The completion of a consistent set of states w.r.t. history connectors will be defined in Section 4.

Two transitions are consistent if they are active in two orthogonal regions, i.e. if their scopes are orthogonal.

$$\downarrow(t_1, t_2) \quad \text{iff} \quad scope(t_1) \perp scope(t_2)$$

This notion can be extended to a set of transitions. A set T of transitions is consistent, denoted by $\downarrow(T)$ iff the transitions of T are pairwise consistent.

If a set of possible executable transitions is not consistent we will use a priority relation to select a consistent subset. The priority of a transition is given by the distance of its scope from the root:

$$prio : T \Leftrightarrow\!\!\!\!\rightarrow \mathbb{N}$$
$$t \mapsto depth(SC) \Leftrightarrow depth(scope(t)) \ .$$

A statechart SC is well-formed ($wff(SC)$) iff for all transitions t we have

- $\downarrow(source(t))$ and $\downarrow(states(target(t))$
- $\forall\, s \in source(t) : mode(s) = \text{OR} \ \Rightarrow \ \forall s' : s < s' \Rightarrow s' \notin source(t)$
- $\forall\, s \in target(t) : mode(s) = \text{OR} \ \Rightarrow \ \forall s' : s < s' \Rightarrow s' \notin target(t)$
- $\forall\, h \in target(t) : mode(state(h)) = \text{OR}$
  $\Rightarrow \ \forall s' : state(h) < s' \Rightarrow s' \notin target(t)$
- $root \notin source(t) \cup target(t)$

In the rest of this paper we will only consider well-formed statecharts.

3.2 The Data-Space of a Statechart

The compositional models we generate distinguish between local variables, and external variables, which are *observable* for a designer analyzing the black-box behaviour of an activity. Only the observables enter the interface.

The concept of an interface of an activity is both implicitly and explicitly available in the STATEMATE system. For each activity, a compile-time analysis is carried out to determine usage of data-items and events, leading to a classification into *local* and *shared*, which is further refined by attributes characterizing the direction of information flow. This interface concept is also *explicitly* defined for so-called *generic activity charts* through corresponding data-dictionary entries. As explained in the introduction, our classification will additionally introduce attributes *fast* and *slow*, modeling the distinction between shared variables and/or events manipulated by the environment of the SUD, and those which are non-local to the activity, but under the control of the SUD.

STATEMATE defines a number of implicitly generated events which e.g. allow to monitor accesses to shared variables or are related to the scheduling primitives used in a controlling statechart. We will below list those observables, which are *implicitly associated with an activity* based on the *explicitly declared interface objects*. We will subsequently use the term "explicit" and "implicit" observables to distinguish if needed between these interface objects.

In the context of a *compositional semantics*, the interface of an activity must finally provide additional *auxiliary* observables, handling divergence, synchronization and incrementation of the clock. Whereas implicit observables are well-known to STATEMATE designers, auxiliary observables arise purely in the context of compositionality. Thus, they require additional documentation to make them accessible to STATEMATE designers willing to perform a compositional proof.

For practical purposes it is important to note, that already the *explicit interface objects* determine completely the set of all observables: all activities share analogous sets of auxiliary observables, and implicit events are canonically associated with explicit observables. Our formal definition reflects this observation.

Events and Variables. The data space of a STATEMATE design may contain *events*, *conditions* and *variables*. Events and conditions are boolean in nature, with different updating mechanisms. To simplify the exposition, we identify conditions with Boolean variables, and hence also disregard all implicit events associated with conditions.

Both events and variables may be bidirectional in the syntax of STATEMATE. In our CSTS semantics, we introduce two CSTS variables e_{in} and e_{out} for each bidirectional shared event, and similarly two variables for each shared variable. The CSTS will use these copies according to their implicit direction.

The set of supported types *SM_Types* for variables is defined in e.g. [13]; it subsumes bits, bit-arrays, records, arrays, queues as well as user defined types. For the purpose of this paper, the exact type system is of no importance. Every type τ is associated with a data domain $\mathcal{D}^\tau$. The disjoint union of all data domains will be denoted by $\mathcal{D}$.

The Interface of an Activity. As explained above, the interface consists of an *explicit*, an *implicit* and an *auxiliary* part.

The Explicit Interface. With an activity A we assume as given its *explicit interface* $e_int(A)$ which contains:

- the set $events(A)$ of events of A together with two functions

$$\begin{aligned} dir &: events(A) \Leftrightarrow\!\!\!\rightarrow \{in, out, inout\} \\ speed &: events(A) \Leftrightarrow\!\!\!\rightarrow \{slow, fast\} \end{aligned}$$

  giving the direction of information flow as well as the distinction between events external to the SUD and local to the SUD;
- the typed set $var(A)$ of shared variables of A together with three functions

$$\begin{aligned} type &: var(A) \Leftrightarrow\!\!\!\rightarrow SM_Types \\ dir &: var(A) \Leftrightarrow\!\!\!\rightarrow \{in, out, inout\} \\ speed &: var(A) \Leftrightarrow\!\!\!\rightarrow \{slow, fast\} \end{aligned}$$

  giving type, direction and speed of shared variables.

If the direction is *in*, the event, resp., variable, may be reacted upon, resp., read. If the direction is *out*, it may only be set. The use of objects of the direction *inout* is unrestricted.

The Implicit Interface. We now turn to the *implicit interface* of an activity A, $i_int(A)$. The following reflects the subset supported in the verification environment under construction, which slightly deviates from those defined in STATEMATE in order to support compositional reasoning.

Shared Variables. For each variable v of direction *inout* in the external interface, we assume that *events*(A) contains the following events:

written_out(v)	event	out	*speed*(v)	activity A writes on v
written_in(v)	event	in	*speed*(v)	environment of A writes on v
changed_out(v)	event	out	*speed*(v)	activity A changes value of v
changed_in(v)	event	in	*speed*(v)	environment of A changes value of v
read_out(v)	event	out	*speed*(v)	activity A reads v
read_in(v)	event	in	*speed*(v)	environment of A reads v

If the direction is *in*, the events written_out(v) and changed_out(v) are omitted. If the direction is *out*, the event read_out(v) is missing.

In this paper we do not handle the STATEMATE constructs `write_data(v)` and `read_data(v)`, but instead allow the above events to occur within actions.

Scheduling Control. We assume that *events*(A) contains the following events controlling activation, suspension, and termination of activity A:

st!(A)	event	in	fast	activates activity A
started(A)	event	out	fast	reports, that activity A has been started
sp!(A)	event	in	fast	terminates activity A
stopped(A)	event	out	fast	reports, that activity A has been terminated
sd!(A)	event	in	fast	suspends activity A
rs!(A)	event	in	fast	resumes activity A

We assume, that *var*(A) contains the following conditions reporting on the status of activity A:

active(A)	bool	var	out	fast	activity A is active
hanging(A)	bool	var	out	fast	activity A is suspended

Note: If A itself is bound to a control statechart, *events*(A) will in particular contain events controlling its sibling activities, and sense conditions regarding their state contained in *var*(A). I.e. if A is a control activity with sibling activities $A_1, \ldots, A_n$, then *events*(A) contains also the events *st!*(A_i), *started*(A_i), *sp!*(A_i), *stopped*(A_i), *sd!*(A_i), *rs!*(A_i), and *var*(A) contains also the conditions *active*(A_i) and *hanging*(A_i) for $1 \leq i \leq n$. Though these events and conditions have a particular pragmatic, they can be treated uniformly as other events and variables of the interface of A in the formal definition of the semantics, and thus do not require an additional syntactic category.

Auxiliary Interface. We finally add within the *auxiliary interface* of A those observables required to deal in a compositional way with divergence within a step, synchronization at super-step boundaries, and incrementation of the clock. All of these are declared as *directed variables*. Auxiliary variables and events are not allowed to occur within guards or actions.

Step-divergence.

step_div(A)	event	out	fast	false iff A is willing to initiate a step
step_div_env(A)	event	in	fast	false iff environment of A is willing to initiate a step

Synchronisation at Super-step Boundaries.

stable(A)	event	out	fast	true iff A is willing to initiate a super-step
stable_env(A)	event	in	fast	true iff the environment of A is willing to initiate a super-step

Incrementing the Clock.

time	event	in	slow	signals the next clock tick.

The Full Interface. The full interface provides the external variables of our CSTS semantics of an activity A. It deviates from the union of the three parts of the interface defined above in that it splits the bidirectional variables and events occurring the external StateMate interface.

Given an external interface *e_int*(A), we denote by *full_int*(A) the interface that contains

- all directed events and variables of *e_int*(A)
- all directed copies of bidirectional events and variables in *e_int*(A)
- all implicit interface objects associated with *e_int*(A)
- all auxiliary interface objects of A.

We will show in Chapter 6, that a construction of the CSTS providing the full interface as externally visible variables is rich enough to capture parallel composition of traces of activities A and B through the synchronized product of the associated CSTSs. Additional components in the product will take care of the issues connected to the updating of shared variables and events which have been replaced by directed copies. In particular, these components will permit us to retrieve the actual StateMate values from the values of the copies.

The Variables of a Statechart. A statechart SC inherits the *external* and *full interface* of the activity A it defines. In the sequel, we assume a fixed full interface of directed events *events*(A) and variables *var*(A) as given.

A statechart SC extends the state-space by defining *local* events and *local* variables, which we assume to be *disjoint* from events and variables of the full interface of SC. In the formal definition we extend the codomain of *dir* by allowing as additional attribute the "direction" *local*, and fix this as "direction" for all local objects. By definition, the *speed* of all local objects is *fast*. Types are assigned to local variables via a function *type*. We denote local events and variables of SC by *events*(SC) and *var*(SC), respectively.

Shared Variables. For each local variable v we assume that $events(SC)$ contains the following events:

written(v)	event	local	fast	SC writes on v
read(v)	event	local	fast	SC reads v
changed(v)	event	local	fast	SC changes value of v

States. For each state s we have the following implicit events and conditions:

in(s)	bool	var	local	fast	SC is in state s
entered(s)		event	local	fast	SC has entered state s
exited(s)		event	local	fast	SC has exited state s

History. History connectors come equipped with implicitly defined events allowing to clear the stored state (resp. configuration): if $state(h) = s$ then

hc!(s)	event	local	fast	clears s history connector
dc!(s)	event	local	fast	clears s deep history connector

Again we assume such implicit events to be contained in $events(SC)$.

We denote by $events(SC, A)$ the union of all local events of SC and events in the full interface of the associated activity, and similarly abbreviate all local and interface variables by $var(SC, A)$.

3.3 Execution Concepts

In addition to the elementary concepts of a transition discussed in Section 3.1 transitions are also labelled with guards to control the execution and actions to be performed on executing a transition. We first discuss these concepts informally:

- *complex guards*
  A transition may only be *fired*, if its *guard* evaluates to *true*. StateMate's syntax views guards as consisting of a pair of predicates, evaluating presence or absence of events, and the current valuation of variables. Among the event-related part are so called *time-out* events discussed below. For the purpose of this paper, a guard is simply a Boolean expression over events and variables, where we view events as (special) variables of type Boolean and identify $e = true$ with e being *present*.
- *complex actions*
  If a transition is fired, its *action* annotation causes events to be generated and variables to obtain new values. In our concrete syntax, the basic action `e := true` (for an event e) stands for the generation of event e (again allowing a more unified semantical treatment by viewing events as special Boolean variables which are automatically reset and which the user can only choose to set to *true*). In general, assignments are provided as basic actions to update variables. Basic actions can be combined by the ;-operator (denoting *parallel* rather than sequential composition, but c.f. the handling of *context variables* below), a variant of if-then-else, and iteration. For the purpose of this paper,

we replace iteration by just one new basic action *div* modelling divergence, and allow as action annotations a ;-list of guarded basic actions.
We delay a discussion of *scheduled actions* to the paragraph addressing the real-time programming features of StateMate.
StateMate evaluates all expressions in guards and assignments with respect to a valuation of events and variables *prior* to firing any transition in a new step.
However, StateMate also includes a class of variables, called *context variables*, which are evaluated with respect to the value determined by the *latest* assignment to this variable. Here, *latest* refers to the order of actions in the ;-list, hence giving this operator some sequential flavour after all. To distinguish these variables from ordinary variables, occurrences of these variables are decorated by \$. In a StateMate design, context variables are usually used as index variables in loops. We use $\$var(SC)$ to refer to the context variables of the statechart SC.

– *real-time constructs*
StateMate provides two ways of relating events and actions to the clock modelling real-time (c.f. chapter 1).
  - Transitions can be triggered, if a time-out period for a monitored event e has expired. Syntactically, StateMate allows as guards *time-out events* of the form $tm(e, texp)$ for events e and integer expressions $texp$. In StateMate, $texp$ counts in time units which are user-definable for each activity; in order not to burden the exposition with conversions between time-scales, we interpret $texp$ as denoting multiples of the real-time clock associated the SUD. In our paper, time-outs can only be monitored for local events and incoming external events.
    Time-out events induce setting of a (simulator internal) timer, which is reset whenever the monitored event occurs. If the timer expires, a (local fast) event is generated signalling the time-out, which is then handled uniformly as other local fast events.
  - Actions can be scheduled to occur after a user specified delay. Syntactically, StateMate allows actions of the form $sc!(a, texp)$, where a is an action and $texp$ an expression of type integer. We again interpret $texp$ as counting time units of the underlying real-time clock.
    It is important to remember, that the "real-time clock" of a StateMate model is NOT related to the physical clock of a target architecture of the embedded control application. A good interpretation of the above real-time constructs is that of *imposing constraints* on the actual (physical) execution time. Suppose, that the user defines the (fictitious) real-time clock of its StateMate model to run with a $1\,ms$ resolution, and poses as guard a time-out on some event e after 5 time-units. If all code-segments to be executed between generation of event e and checking for the time-out event can be *executed* on the target architecture within $5\,ms$, the designer's intuition about posing the time-out *in the model* will match what happens when the generated code runs on the target. The validation of compliance of the target code w.r.t. such timing constraints

is not addressed in this paper. Instead, "real-time verification" as used in this paper refers to the formal proof, that properties expressed in terms of the *fictitious* real-time clock are satisfied in a STATEMATE model operating w.r.t. this *fictitious* real-time clock.

Guards and Expressions. Recall that we view events $e \in \mathit{events}(SC, A)$ as a boolean variable (with restricted user rights). The set of *typed expressions* over $\mathit{events}(SC, A)$ and $\mathit{var}(SC, A)$, $\mathit{Exp}(\mathit{events}(SC, A), \mathit{var}(SC, A))$ is defined as usual, using the predefined operators as defined in the STATEMATE reference manual, with the additional restriction that *expressions do not contain occurrences of out events or variables* (these may only be set but never tested by SC). We just write Exp whenever other parameters are clear from the context, and Exp^{type} to designate expressions of type *type*, abbreviating Exp^{bool} by Bexp, and Exp^{int} by Texp whenever expressions are used to express delays, such as the expression texp within time-outs and scheduled actions. We use exp (resp. b) as meta-variable for (boolean) expressions.

Guards are simply boolean expressions constructed over the sets $\mathit{events}(SC, A)$, $\mathit{var}(SC, A)$, and time-out events in the set $\mathit{Tevents}(SC, A)$ defined inductively by

$$
\begin{aligned}
& e \in \mathit{events}(SC, A) \ \wedge \ \mathit{dir}(e) \in \{\mathit{in}, \mathit{local}\} \Rightarrow \mathit{tm}(e, \mathit{texp}) \in \mathit{Tevents}(SC, A) \\
& \qquad te \in \mathit{Tevents}(SC, A) \Rightarrow \mathit{tm}(te, \mathit{texp}) \in \mathit{Tevents}(SC, A) \\
& te_1, te_2 \in \mathit{Tevents}(SC, A) \cup \{e \mid e \in \mathit{events}(SC, A) \ \wedge \ \mathit{dir}(e) \in \{\mathit{in}, \mathit{local}\}\} \\
& \quad \Rightarrow \quad \mathit{tm}(\mathit{not}\ te_1, \mathit{texp}), \mathit{tm}(te_1\ \mathit{and}\ te_2, \mathit{texp}), \mathit{tm}(te_1\ \mathit{or}\ te_2, \mathit{texp}) \\
& \qquad \in \mathit{Tevents}(SC, A) \ .
\end{aligned}
$$

Note, that time-outs are allowed to be nested. We require, that the delay expression texp always evaluates to a positive delay greater than zero. Furthermore, we require that the delay expressions are constants at compile time. We refer by $\mathit{Tevents}(g)$ to the set of time-out events occurring in a guard g.

The set of *actions* over $\mathit{events}(SC, A)$ and $\mathit{var}(SC, A)$, which will be denoted by $\mathit{Actions}(\mathit{events}(SC, A), \mathit{var}(SC, A))$ or simply $\mathit{Actions}$ if the context is understood, is defined by

`e := true`	where $e \in \mathit{Events}$, $\mathit{dir}(e) \neq \mathit{in}$
`v := exp`	where $\mathit{exp} \in \mathit{Exp}(\mathit{var}(SC, A))^{type(v)}$ and $v \in \mathit{var}(SC, A) \setminus \$\mathit{var}(SC)$, $\mathit{dir}(v) \neq \mathit{in}$
`$v := exp`	where $\mathit{exp} \in \mathit{Exp}(\mathit{var}(SC, A))^{type(v)}$ and $\$v \in \$\mathit{var}(SC)$
`div`	(divergence)
`if b then a else a' fi`	whenever $a, a' \in \mathit{Actions}$
`a ; a'`	whenever $a, a' \in \mathit{Actions}$
`sc!(a, texp)`	whenever $a \in \mathit{Actions}$.

Again, we require texp to evaluate to a positive integer at compile time.

Transition Labels and Static Reactions. We assume that the set of transition names $trans(SC)$ is associated with two additional functions defining the guards and actions of each transition:

$guard : trans(SC) \leftrightarrow\!\!\!\rightarrow Guards(SC, A)$
where $Guards(SC, A) =$
$Bexp(events(SC, A) \cup Tevents(SC, A) \cup var(SC, A))$
defines for t the guard enabling firing of the transition.

$action : trans(SC) \leftrightarrow\!\!\!\rightarrow Actions(SC, A)$
where $Actions(SC, A) = Actions(events(SC, A), var(SC, A))$
defines for a transition t the action to be executed when t is fired.

Statecharts allow as well to associate behaviour to *states* using the concept of *static reactions*: these consist in their most general form of so called *reactions* i.e. a *set of pairs guard/action*. We model this through the function

$$sr : states(SC) \leftrightarrow\!\!\!\rightarrow 2^{Guards(SC,A) \times Actions(SC,A)} .$$

Annotations to states controlling activation/termination of sibling activities such as $within(s)$ (causing termination of the listed activities whenever exiting the state s) or $throughout(s)$ (causing in addition activation of the listed activities when entering s) can be considered as syntactic sugar for actions to be associated with transitions exiting (resp. entering and exiting) state s using the class of scheduling events listed above, and are thus not considered explicitly in this paper.

4 The Dynamic Behaviour of Statecharts I: What Happens in a Step?

4.1 Introduction

We split the discussion of the dynamic behaviour of statecharts into two chapters, roughly corresponding to the two modes of simulation supported by STATEMATE. The current chapter focuses on the concept of a *step* of a statechart. On first sight, the concept is very simple: a step effects a transition from a given state-configuration and a given valuation of (explicit and implicit) variables and events when firing a maximal conflict free set of enabled transitions to a new state-configuration and a new valuation of variables and events. This simple concept becomes complex due to the extension from the simple-minded concept of transitions in finite automata to the full-fledged graphical real-time programming language called statecharts.

Our approach to tackle this complexity is an incremental presentation of the compositional synchronous transition system associated with a given statechart

SC. In particular, the current chapter disregards all issues related to super-steps, incrementing the clock and induced time-bookkeeping issues. Within the chapter, we consider a base model first, which disregards time-outs, scheduled actions, history connectors, and scheduling issues, and introduce these in the above order in subsequent sections.

4.2 The Base Model

This section provides a compositional synchronous transition system for a given statechart SC. In particular, the model allows handling of shared variables and bi-directional events, based on the full interface of the statechart as defined in Section 3.2, and the extensions with local explicit and implicit variables and events.

We will denote the *CSTS* $\Phi_{\text{base}} = (V_{\text{base}}, \Theta_{\text{base}}, \rho_{\text{base}}, E_{\text{base}})$ associated with SC by

$$CSTS[\![SC]\!]_{\text{base}}$$

and will define its constituents in the following subsections.

The System Variables. External variables of Φ_{base} are exactly those defined by the full interface of the activity A whose behaviour is defined by SC

$$E_{\text{base}} = \mathit{full_int}(A)$$

where for the purpose of this section we ignore all items in the full interface related to timing, super-step synchronization and stabilization, as well as scheduling. To allow a uniform presentation of the semantics, we annotate all external *out* variables v_{out} by a prime as in v'_{out}; *primed* variables may only be *set* but never *tested* by SC.

V_{base} is the extension of E_{base} by local system variables defined as union of the following:

- a dedicated variable c of type $\mathit{state_conf}(SC)$ carrying the current state-configuration;
- all local variables $\mathit{var}(SC)$ up to conditions representing states such as $\mathit{in}(s)$ as well as a copy $\mathit{var}(SC)' = \{v' \mid v \in \mathit{var}(SC)\}$ of these; these are needed to ensure, that variables in expressions as well as guards are evaluated w.r.t. the valuation of variables *prior* to updating a step: v' carries the value of v resulting from taking the step; note that an occurrence of a context variable $\$v$ will be evaluated w.r.t. the current value of $\$v'$;
- all local events $\mathit{events}(SC)$ as well as a copy $\mathit{events}(SC)'$ of these, again allowing in guards to refer to events generated in the *previous* step using the unprimed version of events.

Semantics of Expressions, Guards, and Actions. Recall from Chapter 2 that a *valuation* of Φ_{base} is a type preserving mapping assigning values in a (typed) semantic domain $\mathcal{D}$ to the system variables.

$$\sigma : V_{\text{base}} \Leftrightarrow\!\!\rightarrow \mathcal{D}$$

This valuation extends to a *semantics of typed expression* $[\![exp]\!]$ canonically; the only non-standard aspect concerns context-variables and conditions on state:

$$[\![\$v]\!]\sigma = \sigma(\$v')$$
$$[\![in(s)]\!]\sigma = s \in \sigma(c)$$

The semantics of an action a in a valuation σ, $[\![a]\!]\sigma$, is defined canonically up to handling divergent actions and assignment, using the above expression semantics.

Divergent actions set the step divergence flag *step_div* contained in the full interface to *true*, thus retaining totality of the valuation transformation associated with actions.

$$[\![div]\!]\sigma = \sigma[\text{step_div}(A)'/true]$$

Assignment – whether to context variables or ordinary variables – just updates the primed copy of the variable, such as in

$$[\![\$v := exp]\!]\sigma = \sigma[\$v'/[\![exp]\!]\sigma] \ .$$

Selecting Steps and Their Effect on State Configurations. A transition $t \in trans(SC)$ is *enabled in* σ, denoted by $\sigma \models en(t)$, iff its sources are contained in the current state configuration and its guard evaluates to *true*:

$$\sigma \models en(t) \quad \text{iff} \quad source(t) \subseteq \sigma(c) \text{ and } [\![guard(t)]\!]\sigma = true \ .$$

The following algorithm computes in a top-down fashion a set $Steps(\sigma)$ of all maximal consistent subsets of transitions enabled in a valuation σ. Taking the priority into account we can stop the algorithm in a branch when an enabled transition is found, as transitions in a deeper arena have lower priorities.

Starting from the root we traverse through the state hierarchy following the given state configuration $\sigma(c)$. We inductively define a set of steps relatively to a given state $s \in \sigma(c)$ i.e. we compute $steps(\sigma, s) \in 2^{2^T}$ satisfying

$$st \in steps(\sigma, s) \quad \Rightarrow \quad \downarrow st$$

by induction on the depth of s.

1. $mode(s) = \text{BASIC}$:
   $steps(\sigma, s) = \emptyset$
   (A basic state has no inner transitions.)
2. $mode(s) = \text{AND}$:
   $steps(\sigma, s) = \{st_1 \cup \ldots \cup st_n \mid st_i \in steps(\sigma, s_i)\}$
   where $\{s_1, \ldots, s_n\} = child(s)$.

3. $mode(s) = \text{OR}$:
   Let $T = \{t_1, \ldots, t_k\} = \{t \mid scope(t) = s \text{ and } \sigma \models en(t)\}$
   (a) $T = \emptyset$:
   let s' be the unique child of s in $\sigma(c)$.
   $steps(\sigma, s) = steps(\sigma, s')$
   (b) $T \neq \emptyset$:
   all transitions in T have the same priority and they are in conflict to each other.
   $steps(\sigma, s) = \{\{t_1\}, \ldots, \{t_k\}\}$

The steps of a given configuration are then computed by:

$$Steps(\sigma) = steps(\sigma, root) \ .$$

For a given step $st \in Steps(\sigma)$ we compute the sets

$$\begin{aligned} exited(st) &\text{ of states in } \sigma(c) \text{ exited when firing } st, \\ entered(st) &\text{ of states in } \sigma(c) \text{ entered when firing } st, \text{ and} \\ active(st) &\text{ of states active before and after firing } st \end{aligned}$$

by

$$\begin{aligned} exited(st) &:= \{s \mid \exists\, t \in st \text{ with } scope(t) < s\} \cap \sigma(c) \ , \\ active(st) &:= \sigma(c) \setminus exited(st) \ , \\ entered(st) &:= dcompl\left(active(st) \cup \bigcup_{t \in st} target(t) \right) \setminus active(st) \ . \end{aligned}$$

The Effect of Executing a Step. The diagram in Figure 1 summarizes our semantical treatment of steps. Annotations on transitions are elaborated below.

Passing the Synchronization Barrier. A step is only initiated, if both the environment and the component are willing to initiate a new step.

Step Selection. Based on the resulting valuation σ of system variables, we compute the set $Steps(\sigma)$ of all maximally consistent sets of transitions enabled in σ. If $Steps(\sigma)$ is empty, $step_div(A)'$ remains *false* and we re-enter the synchronization barrier; otherwise, the resulting valuation will depend on the chosen step $st \in Steps(\sigma)$. Based on the selected st, we can determine, which states remain active. For all such states, we add their static reactions when computing the new valuation of system variables.

Evaluation Order. Since actions of transitions contained in st as well as those in static reactions may assign conflicting values to shared variables, non-determinism also arises from the order of evaluation of actions. In this semantics, we choose nondeterministically one serialization of all actions of transitions in st and the enabled static reactions. The verification tool will instead detect race conditions and report an error if races occur.

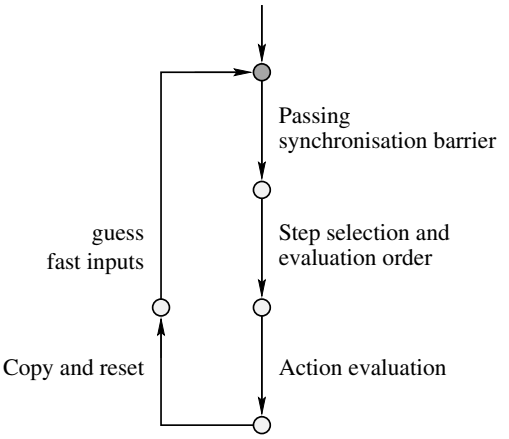


Fig. 1. Synchronization cycle

Action Evaluation. Given an evaluation order, we compute a new valuation of primed variables using the action semantics of the selected sequential composition of the involved actions.

Implicit Events and Variables. We then update the primed copies of implicit variables and events induced from the previous computation.

Update State Configuration. The system variable containing the state configuration is updated with the configuration closure of states entered when firing st.

Making Effect of Step Visible. The initiation of a new step is prepared by copying values of primed system variables, which served to collect the effect of executing the previous step, into their unprimed version using the valuation transformation

$$\mathit{fast_copy} : \Sigma_{\text{base}} \Leftrightarrow \Sigma_{\text{base}}$$

$$\mathit{fast_copy}(\sigma)(sv) = \begin{cases} \sigma(sv') & \text{if } sv \in \mathit{events}(SC, A) \cup \mathit{var}(SC, A) \\ & \quad \text{and } \mathit{speed}(sv) = \mathit{fast} \\ & \quad \text{and } \mathit{dir}(sv) \in \{\mathit{local}, \mathit{out}\} \\ \sigma(sv) & \text{otherwise.} \end{cases}$$

Note, that this does not affect the valuation of *fast* external inputs. While in model generation we will only allocate unprimed copies of out variables for those, which occur within a *changed* test, in our semantics we simply copy the primed versions into unprimed versions for all *out* variables.

Recall, that all *fast* events only survive one step. Hence, after copying an event generated in the last step to the unprimed variable, the primed version will be reset to *false* at the beginning of a step, using the valuation transformation

$$fast_reset : \Sigma_{\text{base}} \Leftrightarrow\!\!\!\!\rightarrow \Sigma_{\text{base}}$$

defined by

$$fast_reset(\sigma)(sv) = \begin{cases} false & \text{if } sv \equiv e' \text{ for } e \in events(SC, A) \\ & \text{and } speed(e) = fast \\ & \text{and } dir(e) \in \{local, out\} \\ \sigma(sv) & \text{otherwise.} \end{cases}$$

In particular, *fast out* events are reset in this initialization phase for the next step.

Guessing Fast Inputs. Whenever SC has completed "its" part of a step, it guesses new values for all *fast* external system variables, including the step-flag of the environment and updates its fast out variables and events.

We note, however, that the environment is not completely free in guessing values of inputs. Recall from Chapter 3, that bi-directional events and variables of the external interface of A are replaced by two directed copies, and that for each variable v events $written_out(v)'$ and $written_in(v)$ as well as events monitoring changes of v $changed_out(v)'$ resp. $changed_in(v)$ have been introduced to give a compositional semantics to accesses to shared variables. We handle resolution of conflicting accesses to shared events and variables externally to the SC (through a "monitor activity" described in more detail in Chapter 6). Resolving conflicts for events is simple: all subactivities of an activity chart indicate via their *out* copy of the event, whether the event was generated in this subactivity; it is generated in the step, if at least one subactivity has raised the *out* line of this event, hence shared events are handled by a "wired or", propagated back to the subactivities through the *in* copy. For shared variables, the *written_out* lines of all subactivities are resolved similarly, in particular setting the *written_in* line if at least one subactivity has written a value on the variable. The value written is offered on the v'_{out} copy of variable v to the monitor; if more than one writer exists, the monitor also propagates the value offered by the winning subactivity to all subactivities through the v_{in} copy of v.

For reasons of efficiency in model generation, the actual implementation treats shared variables slightly different: the *written_in*-line is only set, if a *sibling* activity writes on v, hence indicating, that the locally computed value is invalid. However, if *written_in* is not set, the locally computed value is valid, hence the propagation from v_{out} to the v_{in} copy has to be done within the model, by extending the *copy* valuation transformer defined below.

Changed events are handled analogously to written events by the monitor.

We formalize these remarks as invariants restricting values of guessed inputs within the formal definition of the transition relation associated with the synchronization barrier.

Consider a valuation σ resulting from the execution of a previous step (we discuss conditions on the initial valuation in a subsequent subsection). We formally define the transition relation ρ_{base} modelling the effect of a step as the

product of transition relations corresponding to the different phases elaborated above:

$$\rho_{\text{base}} = \rho_{\text{synch}} \circ \rho_{\text{action_evaluation_state}} \circ \rho_{\text{copy_reset}} \circ \rho_{\text{guess_fast_inputs}}$$

of subrelations defined below.

- ρ_{synch} is just a partial identity on Σ_{base} guaranteeing synchronization:

$$\sigma \; \rho_{\text{synch}} \; \sigma' \quad \text{iff} \quad \sigma = \sigma' \wedge \sigma(\mathit{step_div}(A)) = \sigma(\mathit{step_div_env}(A)) = \mathit{false}$$

- We define the relation $\rho_{\text{copy_reset}}$ on valuations in Σ_{base} by

$$\sigma \; \rho_{\text{copy_reset}} \; \sigma' \quad \text{iff} \quad \sigma' = \mathit{fast_reset}(\mathit{fast_copy}(\sigma)) \; .$$

  New valuations computed at the current step are copied into unprimed variables, and the primed versions of *fast out* and *local* events are reset.
- $\rho_{\text{guess_fast_inputs}}$ guesses new values only for fast inputs, keeps slow out events and slow variables stable, and resets slow in events, since these only survive one step. Formally

$\sigma \; \rho_{\text{guess_fast_inputs}} \; \sigma'$ iff

- $\forall\, sv \in \mathit{var}(SC) \cup \mathit{events}(SC) \cup \mathit{var}(SC)' \cup \mathit{events}(SC)'$
  $\cup \{c, c'\} \cup \{e \in \mathit{events}(A) \mid \mathit{dir}(e) = \mathit{out}\}$
  $\cup \{v \in \mathit{var}(A) \mid \mathit{speed}(v) = \mathit{slow} \vee \mathit{dir}(v) = \mathit{out}\}$:
  $\sigma(sv) = \sigma'(sv)$
- $\forall\, e \in \mathit{events}(A)\ \mathit{dir}(e) = \mathit{in} \wedge \mathit{speed}(e) = \mathit{slow} \;\Rightarrow\; \sigma'(e) = \mathit{false}$
- $\forall\, e \in \mathit{events}(A)\ \mathit{speed}(e) = \mathit{fast} \wedge \mathit{dir}(e) = \mathit{inout} \wedge \sigma(e_{out}) = \mathit{true}$
  $\Rightarrow\; \sigma'(e_{in}) = \mathit{true}$
- $\forall\, v \in \mathit{var}(A)\ \mathit{speed}(v) = \mathit{fast} \wedge \mathit{dir}(v) = \mathit{inout}$
  $\wedge\, \sigma(\mathit{written_in}(v)) = \mathit{false} \;\Rightarrow\; \sigma'(v_{in}) = \sigma(v'_{out})$.

- We define the relation $\rho_{\mathit{action_evaluation}}$ on valuations in Σ_{base} by

$$\sigma \; \rho_{\mathit{action_evaluation_state}} \; \sigma' \quad \text{iff}$$
$$\exists\, st \in \mathit{Steps}(\sigma)\ \exists\, n \in \mathbb{N}\ \exists\, a \equiv a_1; \ldots ; a_n \in \mathit{Actions}(SC)$$
$$\{a_1, \ldots, a_n\} = \mathit{action_set}(st, \sigma)$$
$$\wedge\, \sigma' = \mathit{update_implicits}(st, [\![a]\!]\sigma) \; .$$

The resulting state is changed by evaluating the actions associated with all transitions of a chosen step and static reactions enabled in this step in some sequential order, yielding a new valuation $[\![a]\!]\sigma$. In the above definition, we abbreviated the set of actions from transitions in st as well as of enabled static reactions by

$$action_set(st, \sigma) \;=\; \bigcup_{t \in st} action(t) \\ \cup\; \{a' \mid \exists\, s \in active(st)\; \exists\, g \in Guards(SC, A) \\ (g, a') \in sr(s) \;\wedge\; [\![g]\!]\sigma = true\}\;.$$

This valuation is further transformed by generating all implicit events and setting implicit variables resulting from the evaluation of transitions in st and enabled static reactions, and by updating the state-configuration according to the step st selected and generating all implicit events associated with changing the state-configuration. Note that we can safely use the valuation $[\![a]\!]\sigma = \sigma'$ rather than the predecessor valuation σ, since the semantics of guards of transitions and static reactions and of expressions is by construction *not* affected from the evaluation of a.

We present now the formal definition of this auxiliary function, which takes as parameters a set of (maximal conflict free enabled) transitions st and a valuation σ_{step} and updates this valuation by analyzing what implicits have to generated.

$$update_implicits(st, \sigma_{\text{step}})(sv) \;=$$

$$\begin{cases} true & \text{if } sv \equiv written_out(v)' \text{ for some external variable } v \\ & \text{and } v \in write_set(action_set(st, \sigma_{\text{step}})) \\ true & \text{if } sv \equiv changed_out(v)' \text{ for some external variable } v \\ & \text{and } \sigma_{\text{step}}(v'_{out}) \neq \sigma_{\text{step}}(v_{out}) \\ true & \text{if } sv \equiv read_out(v)' \text{ for some external variable } v \\ & \text{and } v \in read_set(action_set(st, \sigma_{\text{step}})) \\ & \text{or } \big(\exists\, t \in trans(SC)\ source(t) \subseteq \sigma_{\text{step}}(c) \wedge \\ & v \in read_set(guard(t))\big) \\ & \text{or } v \in read_set(g) \text{ for some guard } g \in Guards(SC, A) \\ & \text{s.t. there exists } s \in active(st) \text{ and} \\ & a' \in Actions(SC, A) \text{ with } (g, a') \in sr(s) \\ true & \text{if } sv \equiv written(v)' \text{ for some local variable } v \\ & \text{and } v \in write_set(action_set(st, \sigma_{\text{step}})) \\ true & \text{if } sv \equiv changed(v)' \text{ for some local variable } v \\ & \text{and } \sigma_{\text{step}}(v') \neq \sigma_{\text{step}}(v) \\ true & \text{if } sv \equiv read(v)' \text{ for some local variable } v \\ & \text{and } v \in read_set(action_set(st, \sigma_{\text{step}})) \\ & \text{or } \big(\exists\, t \in trans(SC)\ :\ source(t) \subseteq \sigma(c) \\ & \wedge\, v \in read_set(guard(t))\big) \\ & \text{or } v \in read_set(g) \text{ for some guard } g \in Guards(SC, A) \\ & \text{s.t. there existss} \in active(st) \text{ and} \\ & a' \in Actions(SC, A) \text{ with } (g, a') \in sr(s) \end{cases}$$

$$update_implicits(st, \sigma_{\text{step}})(sv) \;= (continued)$$

$$\begin{cases} active(st) \cup entered(st) & \text{if } sv \equiv c \text{ (state-configuration)} \\ true & \text{if } sv \equiv exited(s)' \text{ and } s \in exited(st) \\ true & \text{if } sv \equiv entered(s)' \text{ and } s \in entered(st) \\ \sigma_{\text{step}}(sv) & \text{otherwise} \end{cases}$$

Note, that also primed versions of slow events and variables are updated. However, they first become visible when copied into their unprimed version at superstep boundaries.

Initial Valuation. An initial valuation of the base-transition system is a valuation σ satisfying the conjuncts of Θ_{base} defined below:

- the initial valuation of c, the system variable maintaining the current state configuration, is simply the closure of the root-state of SC:

$$\sigma(c) = dcompl(root(SC))$$

- initially no event is present:

$$\begin{aligned} \sigma(e'_{out}) = false = \sigma(e_{in}) &\text{ for all fast inout events } e \\ \sigma(e') = false &\text{ for all out events } e \\ \sigma(e) = false &\text{ for all fast in events } e \\ \sigma(e') = false = \sigma(e) &\text{ for all local events } e \end{aligned}$$

  Note, that this entails, that initially both SC as well as its environment are willing to initiate a step. Slow in events are not restricted.
- all variables carry predefined default values: we assume for each type *type* in StateMate's type system a default value $d_{init}^{type} \in \mathcal{D}^{type}$ and require

$$\begin{aligned} \sigma(v'_{out}) = d_{init}^{tpye(v)} = \sigma(v_{in}) = \sigma(v_{out}) &\text{ for all inout variables } v \\ \sigma(v') = d_{init}^{tpye(v)} = \sigma(v) &\text{ for all out variables } v \\ \sigma(v) = d_{init}^{tpye(v)} &\text{ for all in variables } v \\ \sigma(v') = d_{init}^{tpye(v)} = \sigma(v) &\text{ for all local variables } v \;. \end{aligned}$$

  Note that this valuation is consistent with all changed events being absent.

4.3 Adding Time-outs and Scheduled Actions

Introduction. Statecharts support two constructs to model the real-time behaviour of the SUD.

Time-outs of the form $tm(te, texp)$ allow to monitor time-periods elapsed since the last occurrence of the monitored event *te*. Intuitively, whenever the monitored event occurs, a timer associated with *te* is reset. If it is allowed to increment to the value denoted by *texp*, an interrupt is raised signalling that the provided time-period has expired. The simulator handles this by generating an event, which is included in the list of events generated in the step where the exception was raised. Since individual steps in the asynchronous semantics are executed in zero time, this timeout event is to be considered as being raised somewhere in the current super-step, at the value of the clock when initiating the current super-step, and hence is handled as a *slow* event. Since the interrupt raised is considered to be an event, it is natural to allow also such interrupts events to be modelled by time-outs, hence the provision for nested time-outs.

Our semantics – and in fact also the model generated for verification – models time-outs slightly different, though semantically equivalent. With each event *te* monitored by a time-out, we keep *one* counter, which is reset whenever *te* occurs. The counter is incremented whenever the clock advances, i.e. at super-step boundaries only, in the way described in detail in the chapter to come. All time-out expressions occurring in guards are evaluated with respect to this counter, hence comparing the current value of the counter with the value indicated by *texp*. Our mechanism of advancing the clock will guarantee, that we will always have super-steps initiated at *exactly* the expiration time of a time-out. The semantics given below will hence evaluate such a time-out in a guard to *true*, if the value of the relevant counter equals the value allowed by the time-expression *texp*.

As pointed out in the introduction, we assume in this paper that the time-unit of time-expressions coincides with clock ticks.

Scheduled actions of the form $sc!(a, texp)$ allow to trigger evaluation of action *a* after the delay specified by *texp*. For each action *a* occurring within a scheduled action, we will maintain *one* list of scheduled delay times for executing *a*. The evaluation of the above scheduled action will then update the delay-list associated with *a* with the delay entry obtained by evaluating *texp*. This time, delays are decremented as the clock advances, i.e. at super-step boundaries. Again we will guarantee in Chapter 5, that the clock is never incremented beyond a relevant delay, hence there will always be a super-step initiated at exactly that point in time, when a delay expires. The scheduled action is then simply added to the list of actions evaluated in the first step of this super-step.

The subsequent sections will only formalize those aspects of these constructs which are independent of advancement of the clock.

We construct the *CSTS* $\Phi_{\text{time}} = (V_{\text{time}}, \Theta_{\text{time}}, \rho_{\text{time}}, E_{\text{time}})$ associated with *SC* by

$$CSTS[\![SC]\!]_{\text{time}}$$

by defining the changes to the base model in the subsequent sections.

The System Variables. We add to the set of system variables for each event *te* monitored in time-outs a local fast *integer* variable *expire*(*te*). Moreover, for

each action a occurring within a scheduled action a we associate a local fast variable $schedule(a)$ of type *set of integers*. The set of external variables is not extended w.r.t. the base model.

More formally, let

$$\begin{aligned} Sactions(SC) \;=\; \{a \in Actions(SC,A) \;\mid\; & a \in action(trans(SC)) \vee \\ & \exists\, s \in states(SC)\; \exists\, g \in Guards(SC,A)\; (g,a) \in sr(s)\} \end{aligned}$$

then we define

$$\begin{aligned} V_{\text{time}} = V_{\text{base}} \;\cup\; & \{expire(te) \mid te \in Tevents(SC,A)\} \\ & \cup \{schedule(a) \mid a \in Sactions(SC)\} \;. \end{aligned}$$

Semantics of Expressions, Guards, and Actions. We extend the semantics of boolean expressions allowed as guards by defining

$$[\![tm(te,texp)]\!]\sigma = true \quad \text{iff} \quad \sigma(expire(te)) = [\![texp]\!]\sigma \;.$$

We extend the semantics of actions to scheduled actions by defining

$$[\![sc!(a,texp)]\!]\sigma \;\;=\;\; \sigma[schedule(a)/\sigma(schedule(a)) \cup \{[\![texp]\!]\sigma\}] \;.$$

Note, that it is safe to perform the update directly on the local variable (rather than a primed copy), because the value of the timer will first be tested at initiation time of the next super-step.

The Effect of Executing a Step. When checking for enabledness of transitions and static reactions, we now take into account time-outs using the adapted semantic function for guards. Similarly, the adaption of the action semantics defined above already covers inserting future scheduling points for scheduled actions appearing as annotation in fired transitions and enabled static reactions. The one issue still to be resolved in this chapter relates to resetting the timer for time-outs, whenever the monitored event occurs.

Recall from Chapter 3, that the syntax of StateMate provides for nested time-outs, and allows boolean combinations of such nested time-outs and events to be monitored. While nested time-outs by construction will only be reset when the clock advances (and hence at super-step boundaries, see Section 5.4), we have to address in this section resetting of timers due to the occurrence of events which either were generated locally during the current step or provided as input when entering the step.

We now formally define by induction on the monitored events te a function $reset_cond(te)$ which takes a valuation of system variables and determines, whether the reset condition for te has occurred. For local events, the reset condition is obvious: we simply check the (primed version of) for presence of the event. Now consider a timeout event of the form $te' \equiv tm(te,texp')$, where $te \equiv tm(e,texp)$. We have to reset the time associated with te' whenever te occurs. Since in our semantics the occurrence of a time-out event te is represented

by the condition, that the time associated with te has reached the expiration time $texp$, this defines the reset condition for te'. These base cases are then canonically extended to define reset-conditions for all timeout events.

We define

$$reset_cond(te) : \Sigma_{\text{time}} \Leftrightarrow\!\!\!\!\rightarrow \{true, false\}$$

inductively by

$$\begin{aligned}
reset_cond(e)(\sigma) &= (\sigma(e') = true) \text{ for all local events } e \\
reset_cond(e)(\sigma) &= (\sigma(e) = true) \text{ for all events } e \text{ with } dir(e) = in \\
reset_cond(tm(te, texp))(\sigma) &= (\sigma(expire(te)) = [\![texp]\!]\sigma) \\
reset_cond(not\ te)(\sigma) &= \neg reset_cond(te)(\sigma) \\
reset_cond(te_1\ and\ te_2)(\sigma) &= (reset_cond(te_1)(\sigma) \wedge reset_cond(te_2)(\sigma)) \\
reset_cond(te_1\ or\ te_2)(\sigma) &= (reset_cond(te_1)(\sigma) \vee reset_cond(te_2)(\sigma))
\end{aligned}$$

We now integrate resetting of timers as induced by change of local variables and guessed input events as parts of the implicit effects of executing a step. Note that within steps no timers will be reset due to expired time-outs, since the clock is only advanced at super-step boundaries. We will reuse the function *reset_cond* in Chapter 5 to treat resets induced from expired timeouts.

We extend the function *update_implicits* by

$$update_implicits(st, \sigma_{\text{step}})(sv) \quad =$$

$$\begin{cases}
\text{all cases listed in section 4.2 (except the otherwise clause)} \\
0 \qquad \text{if } sv \equiv expire(te) \text{ for some time out event } te \text{ and} \\
\qquad\quad reset_cond(te)(\sigma_{\text{step}}) = true \\
\sigma_{\text{step}}(sv) \text{ otherwise .}
\end{cases}$$

Initial Valuation. Initially, we want to assure, that all time-outs occurring in guards evaluate to *false*. Moreover, since initially no event is present, just letting time pass should not induce a time-out to expire (unless, of course, the monitored event has been generated). We thus extend our semantic domain for timers with a non-standard integer t_{max} satisfying $n < t_{max}$, and initialize all counters $expire(te)$ with t_{max}.

Initially, no action is scheduled, hence we set $schedule(a) = \emptyset$ for all scheduled actions a.

4.4 Adding History

Introduction. History connectors allow to reenter an OR-state in the child last visited. To this end, the target of a transition would point to the history connector associated with an OR-state, rather than to the OR-state itself. By definition, if the OR-state has not yet been visited, the default-state associated with the OR-state is entered. The history of a state can be cleared by emitting

the clear-history event. Following standard practice, the effect of clearing the history will first become visible in the subsequent step.

If not only the child, but the complete configuration below the considered OR-state is to be retrieved, so-called *deep* history connectors may be used as targets of transitions, which conceptually store not only the last visited child, but as well all its last visited descendants. Again, a dedicated event allows to clear such a deep history.

In this paper, we will assume, that events dedicated to clear history can only be emitted and never tested, allowing us to maintain only the primed version of these events. The role of the unprimed version is taken over by local fast variables h_s introduced for each OR-state; (the actual implementation will carry out a dependency analysis to reduce the number of such variables). For these, no primed version is needed, since they are only manipulated through the clear-history events.

We construct the *CSTS* $\Phi_{\text{hist}} = (V_{\text{hist}}, \Theta_{\text{hist}}, \rho_{\text{hist}}, E_{\text{hist}})$ associated with SC by

$$CSTS[\![SC]\!]_{\text{hist}}$$

by defining the changes to the time model in the subsequent sections.

The System Variables. We add to the set of system variables primed versions of clear history and deep-clear history events and local fast variables of type $states(SC)$ remembering the last visited child for all OR-states of SC. More formally, we define

$$V_{\text{hist}} = V_{\text{time}} \cup \{hc!(s)', dc!(s)', h_s \mid s \in states(SC) \wedge mode(s) = \text{OR}\} \ .$$

The set of external variables is not extended w.r.t. the time-model.

Semantics of Expressions, Guards, and Actions. Clear history events occurring in actions are handled as all local fast events, inducing only an update of the primed version of the event. Since none of the introduced objects is allowed in guards or expressions, no change is required in the definition of their semantics.

The Effect of Executing a Step. To determine the effect of firing a set of transitions on the state configuration, we have to take into account history connectors occurring in targets of transitions. To this end, we define the concept of a *history completion* of a connector h w.r.t. a current valuation σ, in particular determining the state-configuration before firing the step as well as the current history of all OR-states.

The history completion $hcompl(h,\sigma)$ of a connector h w.r.t. to a given valuation σ is given by

$h \in Hconn(SC)$ then $hcompl(h,\sigma) = \{state(h), \sigma(h_{state(h)})\}$

$h \in Dhconn(SC)$ then $hcompl(h,\sigma) = S$ where S is the smallest set satisfying

- $state(h) \leq S$
- $state(h) \in S$
- $s \in S$ and $mode(s) = \text{AND}$ then $child(s) \subseteq S$
- $s \in S$ and $mode(s) = \text{OR}$ then $\sigma(h_s) \in S$.

For a set H of history connectors the history completion is given by the union of the individual history completions:

$$hcompl(H,\sigma) := \bigcup_{h \in H} hcompl(h,\sigma)$$

The modification of the set $entered(st)$ of a step st taking history connectors into account is given by

$$entered(st) := dcompl\left(active(st) \cup \bigcup_{t \in st} (states(target(t)) \cup hcompl(conn(target(t)),\sigma)) \right) \setminus active(st)$$

where $conn(target(t))$ denotes the set of history connectors in the target set of t, i.e.

$$conn(target(t)) = target(t) \cap (Hconn(SC) \cup Dhconn(SC)) \ .$$

The definitions of $exited(st)$ and $active(st)$ are not affected by history connectors and are given as in section 4.2.

We update the history variables for all OR-states exited when firing the step by extending the function *update_implicits* with the following cases:

$$update_implicits(st,\sigma_{\text{step}})(sv) = \begin{cases} \text{all cases listed in section 4.3 (except the otherwise clause)} \\ s' \qquad \text{if } \exists\, s \in \sigma_{\text{step}}(c)\ sv \equiv h_s \ \wedge\ s \in exited(st) \ \wedge\ mode(s) = \text{OR} \\ \qquad\quad \wedge\ child(s) \cap \sigma_{\text{step}}(c) = \{s'\} \\ \sigma_{\text{step}}(sv) \text{ otherwise} \ . \end{cases}$$

The last required modification relates to the *fast_copy* function, which has to reset history variables if a clear history event has been emitted. We extend the definition of this function to history variables by defining

$$fast_copy(\sigma)(sv) \quad = \quad \begin{cases} \text{all cases listed in section 4.2 (except the otherwise clause)} \\ default(s) \text{ if } sv \equiv h_s \text{ and} \\ \qquad\qquad \sigma(hc!(s)') = true \ \vee \ \exists\, s' \ \sigma(dc!(s')') = true \wedge \ s' \leq s \\ \sigma(sv) \qquad \text{otherwise .} \end{cases}$$

Initial Valuation. Initially, no clear-history event has been generated and all history variables carry the default entry point of the associated OR-state.

4.5 Adding Scheduling Control

Introduction. Until now we have considered the dynamic behaviour of statecharts at the step-level when *activated*. The STATEMATE system provides flexible mechanism for controlling activities: the controlling statechart can base decisions on activation or termination, suspension or resumption not only on the current activation status of an activity, but on in principle arbitrarily complex evaluations involving not only the current state-configuration but also valuation of events and variables, by allowing scheduling constructs to appear as first class citizens, wherever events are allowed to occur. This section addresses what little overhead is required in the formal model to cope with scheduling: informally, we encapsulate the behaviour of statecharts depicted by the automaton of Fig. 1 in an OR-state *not_hanging* which is itself contained in an OR-state *active*, and add two states *inactive* and *hanging*. The state *not_hanging* is entered whenever $st!(A)$ resp. $rs!(A)$ have been emitted, and left whenever $sp!(A)$ resp. $sd!(A)$ has been emitted – see Fig. 2 below. Conditions *hanging* and *active* simply correspond to the conditions monitoring whether the corresponding states are active. Stopping an activity means resetting the state-configuration to the default-completion of its root and clearing all events. The system variables are *not* reset to the initial valuation. Thus starting a stopped activity may lead to a different behaviour than the initial activation. In contrast, suspension of an activity freezes also the current state-configuration.

A special treatment is required for synchronization events. Consider an activity chart with two components C_1, C_2, only one of which, say C_1 is active. Clearly C_2 is part of the environment of C_1, hence C_2 has to give its permission to C_1 to pass its step-synchronization barrier. Thus, even if C_2 is not active or if it is hanging, it should still participate in handshakes regarding synchronization. It does so on the basis of the current valuation of its synchronization flags: for stopped activities, this entails, that they only provide positive answers to synchronization effects; for suspended activities, this depends on the valuation of synchronization flags at suspension time. Note that by definition all scheduling actions only affect an activities behaviour at step-boundaries.

A more subtle issue regarding suspended variables is a conflict of principles regarding treatment of fast events:

– suspension calls for freezing the current valuation
– however, since even suspended activities participate in steps, the fragile nature of events calls for resetting these after the first step passed being suspended.

The following formal definition follows the second variant.

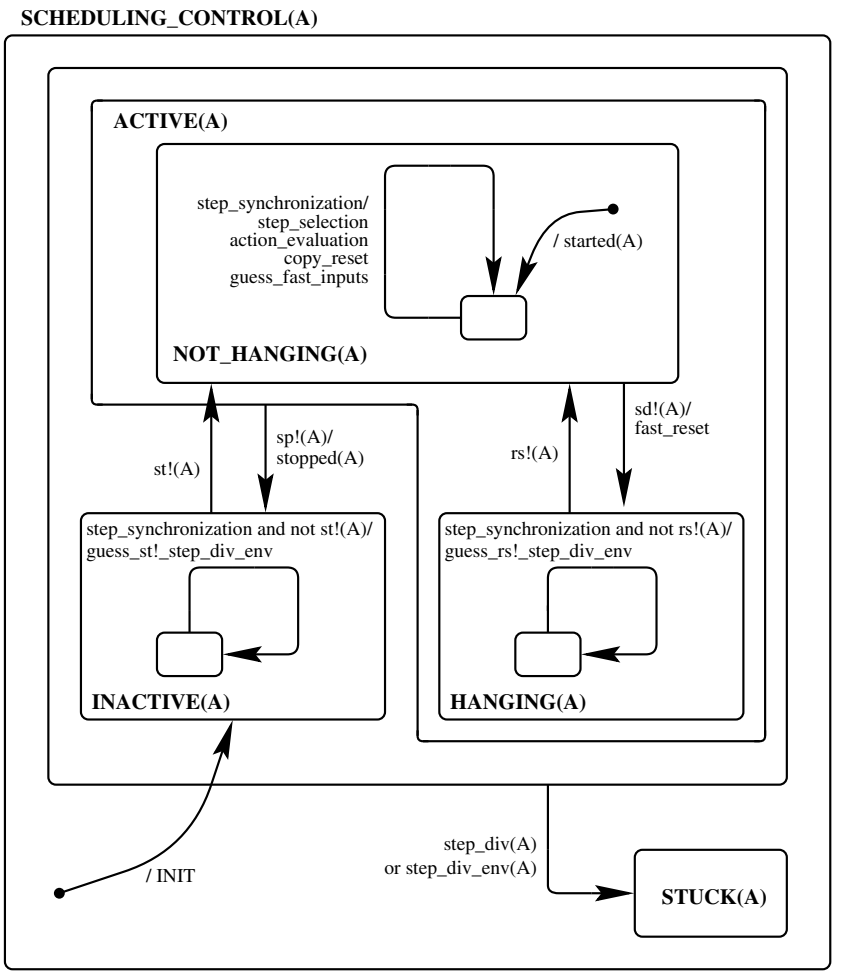


Fig. 2. Scheduling Control

We finally mention, that termination connectors can be equivalently expressed by viewing it as a normal state and asking transitions entering this state to include a self-stopping event in their action part.

System Variables. The external system variables required have already been summarized in Section 3.2. We enrich E_{hist} by fast events $\{st!(A), sp!(A), sd!(A), rs!(A)\}$, fast out events $\{started(A)', stopped(A)'\}$ and boolean fast out variables $\{hanging(A)', active(A)'\}$ to obtain E_{schedule}. The set V_{schedule} of local system variables remains unchanged w.r.t. V_{hist}.

The Effect of Executing a Step. We first define $\rho_{\text{active_synch}}$, requiring additionally to ρ_{synch}, that the activity is active.

$\sigma\ \rho_{\text{active_synch}}\ \sigma'$ iff

- $\sigma = \sigma'$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma(active(A)') = true$
- $\sigma(hanging(A)') = false$
- $\sigma(sp!(A)) = false$
- $\sigma(sd!(A)) = false$

We will redefine the step-transition relation using the additional activation check in synchronization, hence disabling all steps otherwise, and providing transitions characterizing the alternative behaviours depicted in Fig. 2 when being inactive, hanging or not hanging.

To pass from the inactive to the active state, we use ρ_{start}:

$\sigma\ \rho_{\text{start}}\ \sigma'$ iff

- $\sigma(st!(A)) = true$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma(active(A)') = false$
- $\sigma' = \sigma[active(A)'/true][hanging(A)'/false][started(A)'/true]$

Whenever the stop event occurs, A becomes inactive, the stopped event is generated, and the invariant characterizing the initial valuation becomes true.

$\sigma\ \rho_{\text{stop}}\ \sigma'$ iff

- $\sigma(sp!(A)) = true$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma'(stopped(A)') = true$
- $\sigma'(active(A)') = false = \sigma'(hanging(A)')$
- $\sigma'(c) = dcompl(root(SC))$

When inactive, the controller monitors at each step for start-requests, as captured in the transition relation ρ_{inactive} defined by

$\sigma\ \rho_{\text{inactive}}\ \sigma'$ iff

- $\sigma(st!(A)) = false$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma(active(A)') = false$
- $\sigma' = \sigma[st!(A)/b_{st}][step_div_env(A)/b_{div}]$
  for some $b_{st}, b_{div} \in \{true, false\}$.

When hanging, the controller monitors at each step for resume-requests, as captured in the transition relation ρ_{hanging} defined by

$\sigma\ \rho_{\text{hanging}}\ \sigma'$ iff

- $\sigma(rs!(A)) = false$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma(active(A)') = true$
- $\sigma(hanging(A)') = true$
- $\sigma' = \sigma[rs!(A)/b_{rs}][step_div_env(A)/b_{div}]$
  for some $b_{rs}, b_{div} \in \{true, false\}$.

Once a resume request occurs, the status changes from hanging to not_hanging.

$\sigma\ \rho_{\text{resume}}\ \sigma'$ iff

- $\sigma(rs!(A)) = true$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma(active(A)') = true$
- $\sigma(hanging(A)') = true$
- $\sigma' = \sigma[hanging(A)'/false]$.

When active, the activity becomes suspended upon a suspend-request:

$\sigma\ \rho_{\text{suspend}}\ \sigma'$ iff

- $\sigma(sd!(A)) = true$
- $\sigma(step_div(A)) = \sigma(step_div_env(A)) = false$
- $\sigma(active(A)') = true$
- $\sigma(hanging(A)') = false$
- $\sigma' = fast_reset(\sigma)[active(A)'/true][hanging(A)'/true]$.

Reaching a terminating connector will emit a stopped event. Hence we modify the function *update_implicits* by

$$update_implicits(st, \sigma_{\text{step}})(sv) \quad = \quad \begin{cases} \text{all cases listed in section 4.4 (except the otherwise clause)} \\ true \qquad \text{if } sv \equiv stopped(A)' \text{ and } \exists\, t \in st \text{ with} \\ \qquad\quad target(t) \cap Tconn \neq \emptyset \\ \sigma_{\text{step}}(sv) \text{ otherwise .} \end{cases}$$

We finally define the transition relation ρ_{schedule} as the union of the transition relations characterizing the possible behaviours in state inactive, not hanging, and hanging. Observe that an activity will immediately participate in the current step when it receives a start or resume event.

$$\begin{aligned} \rho_{\text{schedule}} = \rho_{\text{inactive}} \\ \cup\ \rho_{\text{start}} \circ \rho_{\text{action_evaluation}} \circ \rho_{\text{copy_reset}} \circ \rho_{\text{guess_fast_inputs}} \\ \cup\ \rho_{\text{active_synch}} \circ \rho_{\text{action_evaluation}} \circ \rho_{\text{copy_reset}} \circ \rho_{\text{guess_fast_inputs}} \\ \cup\ \rho_{\text{stop}} \\ \cup\ \rho_{\text{suspend}} \\ \cup\ \rho_{\text{hanging}} \\ \cup\ \rho_{\text{resume}} \circ \rho_{\text{action_evaluation}} \circ \rho_{\text{copy_reset}} \circ \rho_{\text{guess_fast_inputs}} \end{aligned}$$

The Initial Evaluation. We extend Θ_{time} by requiring additionally $active(A)' = false = hanging(A)'$. As for all fast events, initially none of the scheduling events is present.

5 The Dynamic Behaviour of Statecharts II: What happens in a super-step

5.1 Introduction

This chapter provides a compositional semantics for the asynchronous execution mode of the StateMate simulator.

Recall from Chapter 1, that the asynchronous or super step mode rests on the distinction between events generated from the environment of the SUD and those which are local to the SUD. Intuitively, we can view a StateMate design as a dynamic system which can remain in stable local states unless irritated by external stimuli from the environment of an SUD. The impulse provided by an external stimuli will cause the SUD to loose its state of equilibrium and induce a chain of reactions to recover from this impulse, finding a – possibly different – stable state. Once such a stable state has been achieved, only another external impulse can cause the system to become out of balance.

In the context of embedded systems, this super-step concept provides a natural means to group those sequences of steps necessary to compute new valuations for actuators based on sensor information provided as "external stimuli". STATEMATE as well as other synchronous languages try to reduce modelling effort by assuming the so-called *synchrony hypothesis*: reactions to externally impulses occur in zero time. It is the task of lower design steps to ensure via RT-scheduling mechanisms, that timeliness requirements are met even when actual execution time of the generated code on the target is taken into account. For the purpose of this paper, we take the simple view, that time increases with one unit whenever a new super-step is initiated.

This intuitive concept induces a non-trivial effort when striving for a *compositional* semantics, since it subsumes the *distributed termination problem*[22]. In a compositional setting, each activity has only a *local* view on termination of a super-step: while the activity itself may not be able to perform further steps based on local and fast objects, other activities may still be busy computing their share of the current super-step. In particular, as long as one activity is still unstable, it may always decide to emit fast events or change valuations of fast variables, causing a locally stable activity to become instable. Our compositional semantics incorporates protocols for detecting stability using the events *stable* and *stable_env* introduced in Chapter 3. We extend the simulation cycle from Fig. 1 by first explicitly handling the case, where step-selection induces an *empty action set.*[5] The transition system defined in Chapter 4 does not in such a situation define any successor, hence inducing a terminating computation sequence. In this chapter, we take this condition as an *indicator*, that stability may have been reached. For the purpose of this introduction, let us first assume, that an empty action set implies, that from the perspective of this activity, the super-step cannot be prolonged by further steps. The activity would signal this by generating the event *stable* [6], and then proceed to guess *fast* inputs, including the environment's flag *stable_env*. As long as the environment is not stable, new fast inputs represent reactions of other activities, produced in their strive to reach stability, hence we enter the "ordinary" step-simulation cycle, typically producing further local steps based on new fast inputs. The *copy_reset* phase will automatically reset the stable flag if indeed a step is now possible, hence maintaining the invariant, that this flag is only set, if no local continuation of a super-step is possible. If, on the other hand, the environments stable-flag was set, we have reached *global agreement* on completion of the previous super-step. This entails, that at this stage locally computed slow events and updates to slow variables have to be made visible to the environment, and new values for slow in variables as well as slow in events have to be guessed. The technical machinery to achieve this is a straightforward adaption from the "fast" case and will thus only be addressed in the following technical subsections. Note that together with

[5] Recall, that the action set comprises actions of transitions in the selected step as well as those of enabled static reactions.

[6] Conceptually replacing "step execution" and "copy_reset" in Fig. 1.

the already performed communication on the fast level we have now provided room for a complete update of the full interface of an activity.

Before we turn to the impact of super-steps on StateMate's real-time features, we need to patch the above picture in order to model a peculiar behaviour regarding stability built into the simulator.

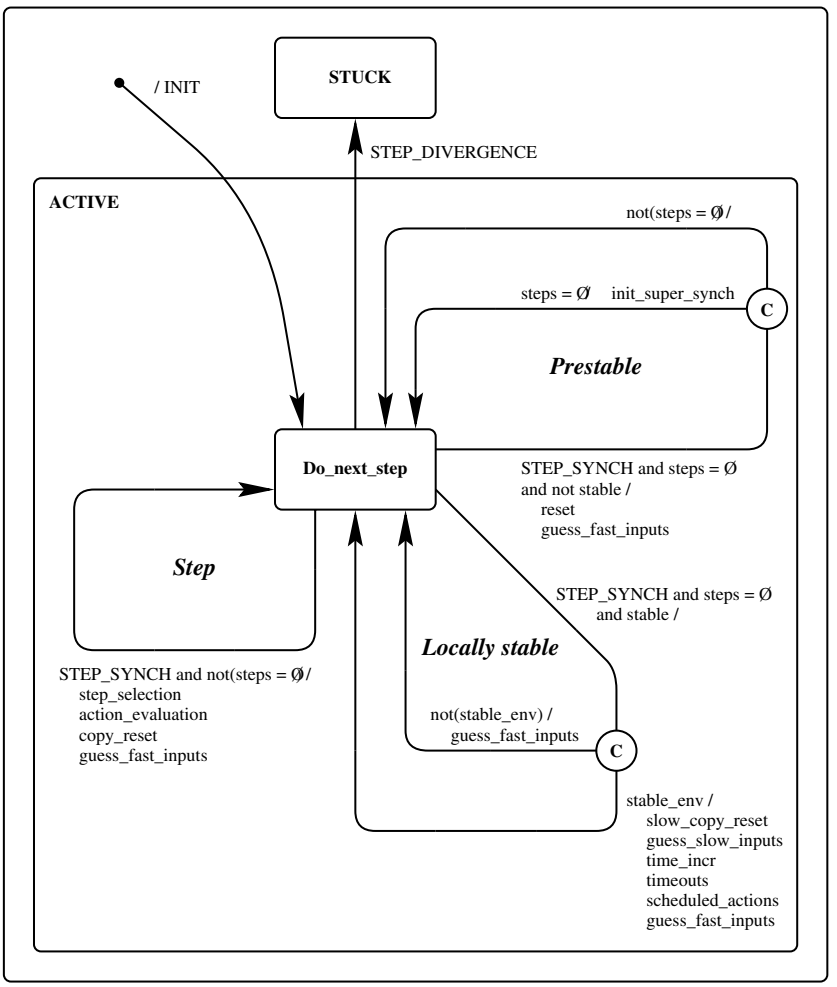


Fig. 3. Superstep synchronization[7]

Suppose again, that the step-evaluation phase results in an empty action-set. Now recall, that guards allow to test for absence of events. StateMate adopts

[7] In contrast with the usual StateMate interpretation the second test in the prestable loop has to be evaluated with the new values.

the view, that absence of events cannot only be caused by not having the event generated in the previous step, but also by the resetting mechanism (modelled by the reset-function). Hence, even if no transition is enabled after completion of a step, just resetting the events generated in the previous step may cause guards to become true, hence inducing successor steps. Since this paper strives for a semantics, which is congruent to the simulation semantics, our model reflects this additional cause of local instability, by introducing a prestable loop, where all events are reset. Only if the action-set remains empty the stable flag will be generated. Fig. 3 summarizes the result of this discussion.

We now discuss informally aspects relating to real-time. In the context of STATEMATE designs, we will assume, that the virtual clock *time* ticks exactly at superstep-boundaries. This causes several side-effects related to STATEMATE's real-time programming constructs: timers associated with time-outs and scheduled actions have to be updated, reflecting the passage of one time unit. As a result, time-outs may expire, and – due to the potential nesting of timeouts – timers for timeouts need to be reset. Moreover, scheduled actions may now become due for execution, and hence need to be evaluated prior to the initiation of the first step of the new superstep.

This chapter is organized as follows. We first introduce in Section 5.2 system variables required to detect stabilization of super-steps. Section 5.3 handles communication through slow interface objects. Three sections are devoted to time-dependent issues: the effect of incrementing the clock on timers, resetting timers of monitored and expired time-outs, and elaboration of due scheduled actions. Section 5.7 presents the formal definition of a compositional model supporting the asynchronous execution model, collecting the formal definition of the three types of transitions depicted in Fig. 3, taking into account the interaction between asynchronous execution of activities and scheduling, and leading to a formal definition of the compositional synchronous transition system

$$CSTS[\![SC]\!]_{\text{super}} \;=\; \Phi_{\text{super}} = (V_{\text{super}}, \Theta_{\text{super}}, \rho_{\text{super}}, E_{\text{super}})$$

associated with SC.

5.2 The System Variables

The set of external variables now subsumes unprimed versions of the stable flag of the activity A under consideration as well as for its environment, as fast out resp. in events.

$$E_{\text{super}} = E_{\text{schedule}} \cup \{stable(A), stable_env(A)\} \ .$$

The local variables are those of V_{schedule}.

5.3 Communication through Slow Interface Objects

Whenever a super-step synchronization succeeds, locally computed values of slow out interface objects become visible by copying primed into unprimed versions of

the corresponding external variables, and new values of slow in interface objects are guessed. The adaption of these transitions from the fast case is immediate.

Copy. If the environment's super-flag is set, the initiation of a new super-step is prepared by copying values of primed system variables, which served to collect the effect of executing the previous super-step, into their unprimed version using the valuation transformation

$$slow_copy : \Sigma_{\text{super}} \Leftrightarrow\!\!\!\rightarrow \Sigma_{\text{super}}$$

$$slow_copy(\sigma)(sv) = \begin{cases} \sigma(sv') \text{ if } sv \in events(SC, A) \cup var(SC, A) \\ \qquad \text{and } speed(sv) = slow \text{ and } dir(sv) = out \\ \sigma(sv) \text{ otherwise.} \end{cases}$$

Note, that this does not affect the valuation of slow external inputs. While in model generation we will only allocate unprimed copies of out variables for those, which occur within a changed test, in our semantics we simply copy the primed versions into unprimed versions for all out variables.

Reset. We will reset all slow out events to *false* at the beginning of a super-step, using the valuation transformation

$$slow_reset : \Sigma_{\text{super}} \Leftrightarrow\!\!\!\rightarrow \Sigma_{\text{super}}$$

defined by

$$slow_reset(\sigma)(sv) = \begin{cases} false \text{ if } sv \equiv e' \text{ for } e \in events(SC, A) \\ \qquad \text{and } speed(e) = slow \text{ and } dir(e) = out \\ \sigma(sv) \text{ otherwise.} \end{cases}$$

We define the relation $\rho_{\text{slow_copy_reset}}$ by

$$\sigma \; \rho_{\text{slow_copy_reset}} \; \sigma' \quad \text{iff} \quad \sigma' = slow_reset(slow_copy(\sigma)) \; .$$

Guessing Slow Inputs. $\rho_{\text{guess_slow_inputs}}$ guesses new values for slow inputs, and keeps all other interface objects stable. Again, invariants regarding bidirectional slow interface objects have to be respected. Formally,

$\sigma \; \rho_{\text{guess_slow_inputs}} \; \sigma'$ iff

- $\forall\, sv \in var(SC) \cup events(SC) \cup var(SC)' \cup events(SC)'$
  $\cup \{c, c'\} \cup \{e \in events(A) \mid speed(e) = fast \vee dir(e) = out\}$
  $\cup \{v \in var(A) \mid speed(v) = fast \vee dir(v) = out\}$
  $\sigma(sv) = \sigma'(sv)$
- $\forall\, e \in events(A)\; speed(e) = slow \wedge dir(e) = inout \wedge \sigma(e_{out}) = true$
  $\Rightarrow \sigma'(e_{in}) = true$
- $\forall\, v \in var(A)\; dir(v) = inout \wedge speed(v) = slow \wedge \sigma(written_in(v))$
  $= false \Rightarrow \sigma'(v_{in}) = \sigma(v'_{out}) \; .$

5.4 What Happens on a Clock-tick I: Updating Timers

Whenever a super-step synchronization has been successfully performed, the virtual clock *time* is advanced. STATEMATE allows the user to instrument the simulator as to the next point in time to which the virtual clock is to be advanced. In our formal semantics, we simply step through the time-axis of the virtual clock, adjusting the timers kept for time-outs and scheduled actions accordingly. The actual model constructed for model-checking takes a more elaborate approach to eliminate trivial steps, in which neither slow-inputs change, nor timeouts or scheduled actions expire.

We have introduced an explicit in event *time* representing clock ticks in order to provide a uniform reference clock in environments, where STATEMATE models are analyzed together with languages such as Signal, in which no single clock is identified as reference clock. From STATEMATE's point of view, we *assume* that the time event is present whenever a super-step synchronization occurs. This reflects the synchrony hypothesis, guaranteeing that all responses to external stimuli are computed infinitely fast, prior to new arriving external stimuli. This assumption is "built into" the formal semantics and has to be checked when mapping the generated model on a target architecture.

Recall from Section 4.3, that each timeout-event te is supported by a system variable $expire(te)$, maintaining the number of clock ticks which occurred since the timer was set. While such timers thus are incremented upon each clock tick, timers associated with scheduled actions maintain the number of clock ticks still required until the action is to be evaluated, hence requiring these to be decremented at each clock tick.

Formally, we define the transition relation $\rho_{\text{time_inc}}$ on Σ_{super} by

$\sigma \ \rho_{\text{time_inc}} \ \sigma'$ iff

- $\sigma(time) = true \wedge$
- $\sigma' = \sigma[expire(te)/\sigma(expire(te)) + 1 \mid te \in Tevents(SC, A)]$[8]
  $[schedule(a)/\{d \Leftrightarrow 1 \mid d \in \sigma(schedule(a))\} \mid a \in Sactions(SC)]$.

5.5 What Happens on a Clock-tick II: Time-out Events

By incrementing timers for time-out, we have guaranteed that time-outs occurring as guards evaluate to true, if their time expression evaluates to the now updated value of the associated timer (c.f. the definition of the semantics of time-outs in Section 4.3). In addition we now have to resolve complexities induced from *nested timeouts*.

Timers associated with time-outs have to be reset, whenever the monitored time-out event occurs. Since the monitored event can itself be a time-out event,

[8] The operation $+1$ is extended to the non-standard integer t_{max} by defining $t_{max}+1 = t_{max}$.

incrementing the clock can cause these to expire. In defining the function *reset_cond(te)*, we have already catered for resetting of timers due to expired timeouts. Recall from Section 4.3, that the reset condition for a nested timeout event of the form $te' \equiv tm(te, texp')$, where $te \equiv tm(e, texp)$, occurs, whenever the time associated with te matches its delay expression:

$$reset_cond(tm(te, texp))(\sigma) = (\sigma(expire(te)) = [\![texp]\!]\sigma) \ .$$

By incrementing the clock (and hence incrementing this timer), this condition may have become true and the counter associated to te has be be reset:

$$\sigma \ \rho_{\text{timeouts}} \ \sigma' \quad \text{iff}$$
$$\sigma' = \sigma[expire(te)/\text{if } reset_cond(te)(\sigma) \text{ then } 0 \text{ else } \sigma(expire(te)) \mid te \in Tevents(SC, A)] \ .$$

5.6 What Happens on a Clock-tick III: Scheduled Actions

As a second side-effect of incrementing time, a scheduled action a may be due, in our model represented by having a zero waiting time in the set $schedule(a)$ of delays until a is due to be evaluated. The StateMate simulator evaluates all such actions *prior* to initiating the first step of a new super-step. The transition relation $\rho_{\text{sc_actions}}$ executes any due scheduled actions in a random order. Within the model generated for model-checking, we will instead check for race-conditions and evaluate due scheduled actions in some fixed order. Additionally, we have to eliminate zero-delay entries from all schedules.

$$\sigma \ \rho_{\text{sc_actions}} \ \sigma' \quad \text{iff}$$
$$\sigma = \sigma' \ \wedge \ \forall\, a \in Sactions(SC) : 0 \notin \sigma(schedule(a))$$
$$\text{or}$$
$$\exists\, n \in \mathbb{N}\ \exists\, a \equiv a_1; \ldots ; a_n \in Actions(SC)$$
$$\sigma' = [\![a]\!]\sigma[schedule(a_i)/\sigma(schedule(a_i)) \setminus \{0\} \mid 1 \le i \le n]$$
$$\wedge \{a_1, \ldots, a_n\} = \{a' \in Actions(SC) \mid \exists\, sc!(a', texp) \in Sactions(SC)\ 0 \in \sigma(schedule(a'))\}$$

5.7 Putting Everything Together: A Compositional Model for the Asynchronous Simulation Mode

We now collect together the different pieces to form the control-automaton of Fig. 3. The individual transition relations will be collapsed into the three types of transitions depicted, each guarded by a partial identity modelling the "guard" of the transition in Fig. 3. Note that no transition can be taken, if step-divergence occurs.

The Step Transition. The step transition collects all test and valuation changes when executing a step, taking into account all features presented in Chapter 4, as included in the most elaborate model supporting scheduling presented in Section 4.5.

Define ρ_{instable} as the partial identity on Σ_{super} induced by the emptiness test on the step set:

$$\sigma \ \rho_{\text{instable}} \ \sigma' \quad \text{iff} \quad \sigma = \sigma' \wedge \mathit{Steps}(\sigma) \neq \emptyset$$

Note that static reactions are only executed, if a step was taken, hence it suffices to test for emptiness of the step-set to deduce, that no action will be executed.

We define the step-transition relation on Σ_{super} as the product of this test and the transition relation ρ_{schedule} defined in Section 4.5, handling all features of StateMate on the step-level:

$$\rho_{\text{step}} = \rho_{\text{instable}} \circ \rho_{\text{schedule}}$$

The Prestable Transition. This transition accounts for steps possibly resulting from resetting fast or local events, though no steps are possible based on the valuation of variables and events resulting from the previous step. In the second phase of this step we test whether the reset operation enables new transitions. If this is not the case, local stability is reached and hence the stable flag is generated.

Define $\rho_{\text{prestable_reset}}$ as the relation on Σ_{super} which tests for stability and performs a fast reset.

$$\sigma \ \rho_{\text{prestable_reset}} \ \sigma' \quad \text{iff}$$
$$\mathit{Steps}(\sigma) = \emptyset \wedge \sigma' = \mathit{fast_reset}(\sigma) \ .$$

We extend this by a test on step-synchronization and append a guess for new input values to define the transition relation $\rho_{\text{prestable_init}}$ associated with the first phase of the prestable transition.

$$\rho_{\text{prestable_init}} = \rho_{\text{synch}} \circ \rho_{\text{prestable_reset}} \circ \rho_{\text{guess_fast_inputs}}$$

For the second phase the relation $\rho_{\text{init_super_synch}}$ sets the stable flag if the action set is still empty and $\rho_{\text{not_stable}}$ denotes the partial identity in the case stability is not reached.

$$\sigma \ \rho_{\text{init_super_synch}} \ \sigma' \text{ iff } \mathit{Steps}(\sigma) = \emptyset \wedge \sigma' = \sigma[\mathit{stable}(A)/\mathit{true}]$$
$$\sigma \ \rho_{\text{not_stable}} \ \sigma' \text{ iff } \mathit{Steps}(\sigma) \neq \emptyset \wedge \sigma' = \sigma$$

Combining both phases the prestable step is given by

$$\rho_{\text{prestable}} = \rho_{\text{prestable_init}} \circ (\rho_{\text{init_super_synch}} \ \cup \ \rho_{\text{not_stable}}) \ .$$

The Locally-stable Transition. The relation $\rho_{\text{stable_synch}}$ is only taken, if locally stability has been achieved, as indicated by a set stable flag and an empty step-set, then setting the stable-flag.

$$\sigma \ \rho_{\text{stable_synch}} \ \sigma' \quad \text{iff} \quad \mathit{Steps}(\sigma) = \emptyset \wedge \sigma(\mathit{stable}(A)) = \mathit{true} \wedge \sigma' = \sigma$$

The relation $\rho_{\text{super_synch}}$ tests for super-step synchronization:

$$\sigma \ \rho_{\text{super_synch}} \ \sigma' \quad \text{iff} \quad \sigma' = \sigma \wedge \sigma(\mathit{stable}(A)) = \mathit{true} = \sigma(\mathit{stable_env}(A))$$

We denote the partial identity guaranteeing an unsuccessful super-step synchronization by $\rho_{\text{no_super_synch}}$.

Collecting together the relations handling communication through slow interface objects, guessing slow interface objects, update of time, handling expired time-outs, and scheduling due actions, we define the transition $\rho_{\text{locally_stable}}$ by

$$\begin{aligned}
\rho_{\text{locally_stable}} \ = \ & \\
& \rho_{\text{stable_synch}} \circ \\
& \big(\rho_{\text{super_synch}} \circ \rho_{\text{slow_copy_reset}} \circ \rho_{\text{guess_slow_inputs}} \\
& \quad \circ \rho_{\text{time_inc}} \circ \rho_{\text{timeouts}} \circ \rho_{\text{sc_actions}} \\
& \ \cup \ \rho_{\text{no_super_synch}}\big) \\
& \circ \rho_{\text{guess_fast_inputs}} \ .
\end{aligned}$$

The order of evaluation when initiating a new super-step is again defined in a way compliant to the StateMate simulator. If no super-step synchronization occurs, a step transition will be taken subsequently with the fast inputs guessed in this transition, since the synchronization test will remain positive.

We summarize the transition possible under the super-step semantics when active in the transition relation

$$\rho_{\text{active}} \quad = \quad \rho_{\text{locally_stable}} \cup \rho_{\text{step}} \cup \rho_{\text{prestable}} \ .$$

The Impact of Scheduling. Since all scheduling control is realized using fast interface objects, the formal model defined in Section 4.5 already provides a natural basis for an integration of scheduling and super-step execution: we simply replace the step-relation by the transition relation characterizing the active state, as done in the formal definition of the superstep-relation of the *CSTS* associated with *SC*. It is, however, worth while to consider the impact this definition has on the interaction of scheduling control and super-steps.

$$\begin{aligned}
\rho_{\text{super}} = \ & \rho_{\text{inactive}} \\
& \cup \rho_{\text{start}} \circ \rho_{\text{step}} \\
& \cup \rho_{\text{active}} \\
& \cup \rho_{\text{stop}} \\
& \cup \rho_{\text{suspend}} \\
& \cup \rho_{\text{hanging}} \\
& \cup \rho_{\text{resume}} \circ \rho_{\text{step}} \ .
\end{aligned}$$

By definition of the initial valuation given below, we are assured that in inactive states both step and super-step synchronization are possible, hence inactive activities will not block sibling activities to proceed in steps nor super-steps. For suspended activities, as soon as step divergence occurs, the activity will become stuck, while itself not producing step divergence when suspended. Since the status of the *stable*(A) flag is frozen when suspended, activities suspended in the middle of a super-step will inhibit other activities from completing their super-step. This requires either care when suspending activities, or a revision of the semantics.

The Initial Valuation. We extend the initial valuation by requiring *stable*(A) as well as *stable_env*(A) to be initially true.

6 Semantics of Activity Charts

An activity chart describes the activities of a system as well as the flow of data elements and events between these activities. As a statechart an activity chart can also be defined in a hierarchical way. In contrast to a statechart an activity chart does not describe the behaviour of a system, but gives only the static aspects of a considered system. It defines a systems in terms of activities and shows the information flow between these activities. The dynamic is described inside one activity by a statechart. An activity can also be some other externally defined component not modelled by a statechart [9]. Every structured activity contains at most one *controlling activity* which has to be given by a statechart, which will be denoted as the *controlling statechart*. In the context of our compositional semantics we assume that this controlling statechart is always given. This control activity determines the dynamics of all other subactivities. It has the control to activate or deactivate its sibling activities. If an activity is further structured by subactivities, one of the subactivities will again act as the controlling part.

This chapter will define the composition of activities in terms of the composition of its control activities. Assume that an activity A is given by a controlling statechart SC together with subactivities $A_1, \ldots, A_n$ (cf. Fig. 4). A subactivity A_i may again be defined by a statechart or by a composition of subactivities as illustrated in Fig. 5. In the previous sections we have developped a semantics of an activity chart given by one statechart, i.e. we have only described the semantics of a leaf in this hierarchical structure (e.g. A_0 or SC_{11} of Fig. 5). In this section we will discuss the composition of activities to derive a semantics of a hierarchically given activity chart.

The semantics of a composite system can be defined by a parallel composition of its subcomponents. In a first approach the parallel composition can be given by a parallel composition of statecharts as given in Fig. 6, i.e. the controlling

[9] The StateMate system offers also the description of an activity by so called *mini-specs*.

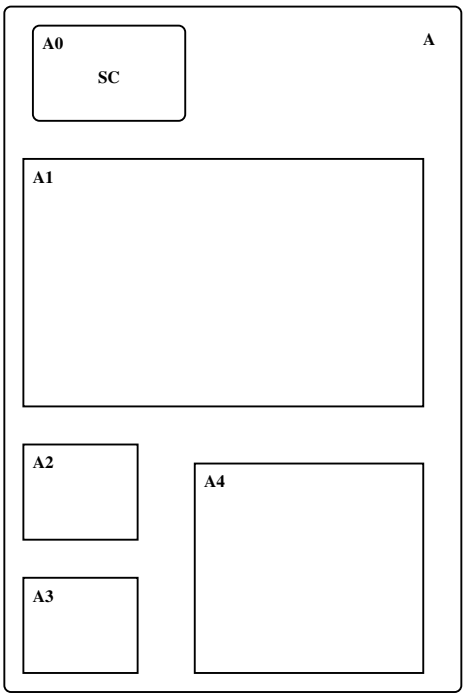


Fig. 4. Activity chart with subactivities

statechart SC operates in parallel with statecharts $SC(A_i)$ derived from the subactivities A_i. In addition to this we have to distribute scheduling information and manage the resolution of shared variables and events by the mentioned monitor concept. If the activity A is started this event has to be transferred to the controlling activity A_0 as $st!(A_0)$. All other scheduling events ($sp!(A)$, $sd!(A)$, $rs!(A)$) should also be distributed to the other subactivities (see Figure 7). This relationship between the scheduling events can easily be described by an invariant on the allowed valuations.

$$
\begin{array}{l}
started(A) \leftrightarrow started(A_0) \ \wedge \ stopped(A) \leftrightarrow stopped(A_0) \wedge \\
active(A) \leftrightarrow active(A_0) \ \wedge \ hanging(A) \leftrightarrow hanging(A_0) \wedge \\
st!(A) \rightarrow st!(A_0) \wedge \\
sp!(A) \rightarrow \bigwedge_{i=0}^{n} sp!(A_i) \ \wedge sd!(A) \rightarrow \bigwedge_{i=0}^{n} sd!(A_i) \ \wedge rs!(A) \rightarrow \bigwedge_{i=0}^{n} rs!(A_i)
\end{array}
$$

To handle inout events and shared variables we have to consider the interfaces of the activities. Each activity A resp. A_i is associated with an interface which defines the flow of data and events between these components. The semantics of A is given by combining the transition systems Φ_i of the components A_i and hiding the internal objects. But we have not only to take the parallel composition of the transition systems Φ_i as defined in section 2.1, but we have to deal with

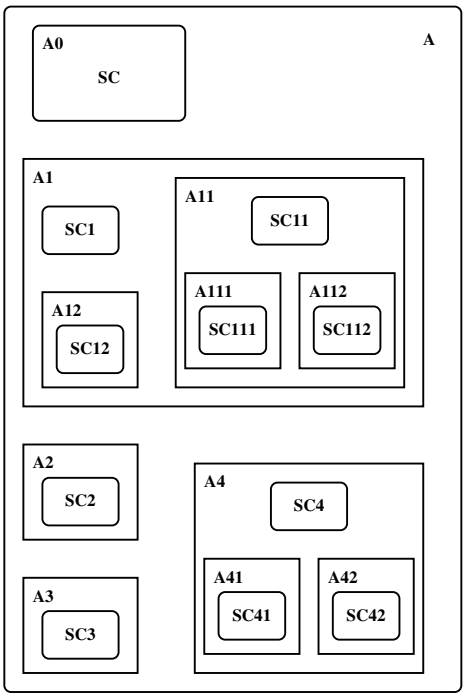


Fig. 5. Hierarchy of activities

the resolution of multi-written variables and multi-generated events. As already indicated this will be done by adding monitors for each shared variable and event of mode inout. A monitor of a shared variable v will resolve a common out value v_{out} of the activity A out of the out values of the individual components A_{i_j} which have v in their interface. Additionally the associated read, written, and changed events are handled by the monitor. To combine the individual out values we have to give them individual names in the combined valuation. Hence we will use a prefix operation which will rename the now local out values v_{out} of a component A_i to $A_i.v_{out}$.

The composite system A is also written as $A_0 \| \ldots \| A_n$ and the A_i are called the components of A. We use $Comp(A)$ to refer to these components. Concerning with the interface of the components we may be interested in those components which refer to a common object, variable v or event e. Hence, we use the notation $Comp(A, v)$ to denote those components C of $Comp(A)$ which have v in its interface, i.e. $Comp(A, v) = \{C \in Comp(A) \mid v \in e_int(C)\}$.

6.1 Component Renaming

For composing system we require that local variables and events are disjunct. For common variables v and bidirectional events e we first rename the directed

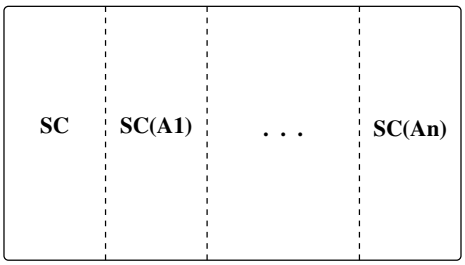


Fig. 6. Basic concept for composing subsystems

copies v_{in} and v_{out}, resp. e_{in} and e_{out}, to make them unique to a component. This is done by prefixing these variables and events by the component name.

Hence, given a transition system $\Phi = (V, \Theta, \rho, E)$ for an activity C as defined in the previous sections, we denote by $rename(\Phi, C)$ the transition system $\Phi = (V', \Theta', \rho', E')$ defined by

$$
\begin{aligned}
V' &:= V \\
E' &:= \{C.stable, C.stable_env, C.step_div, C.step_div_env\} \cup \\
&\quad \{C.e_{in}, C.e_{out} \mid e_{in}, e_{out} \in E\} \cup \\
&\quad \{C.v_{in}, C.v_{out} \mid v_{in}, v_{out} \in E\} \cup \\
&\quad \{e \mid e \in E \text{ and } dir(\mathrm{e}) \neq inout\ \} \cup \\
&\quad \{v \mid v \in E \text{ and } dir(\mathrm{v}) \neq inout\} \\
\Theta' &:= \{\sigma \mid \sigma_{rename(E',E)} \in \Theta\} \\
\rho' &:= \{(\sigma, \sigma') \mid (\sigma_{rename(E',E)}, \sigma'_{rename(E',E)}) \in \rho)\}\ .
\end{aligned}
$$

where $\sigma_{rename(E',E)}$ is a valuation of $V \cup E$ where an element of E has the value of the corresponding element of E' given by σ, i.e. assume that $\sigma : V \cup E' \Leftrightarrow \mathcal{D}$, then $\sigma_{rename(E',E)} : V \cup E \Leftrightarrow \mathcal{D}$ and

$$
\sigma_{rename(E',E)}(w) = \begin{cases}
\sigma(w) & \text{if } w \in V \\
\sigma(C.w) & \text{if } w \in \{stable, stable_env, step_div, step_div_env\} \\
 & \text{or } w \equiv e_{in} \text{ or } w \equiv e_{out} \text{ for some event } e \\
 & \text{or } w \equiv v_{in} \text{ or } w \equiv v_{out} \text{ for some variable } v \\
\sigma(w) & \text{otherwise}\ .
\end{cases}
$$

6.2 Monitor for Variables

To distribute a value of a shared variable v in a composite system $A = A_1 \| \ldots \| A_n$ we introduce a *monitor* $Monitor(v)$ as shown in Figure 8 for two components. It observes all actions of the components which write on v, collects the values and broadcasts a (nondeterministically) selected value to the environment. Furthermore it reads the given values from the environment and transports appropriate values to the components. It does not only manage the value of v but also the related read, written, and changed events. The behaviour is captured by the transition system

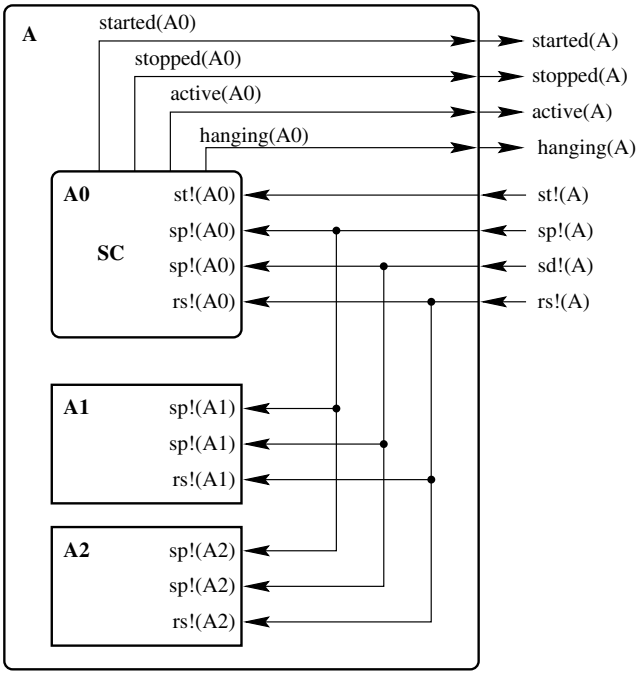


Fig. 7. Distribution of scheduling information

$$\Phi_{\mathrm{Mon}(v)} = (V_{\mathrm{Mon}(v)}, \Theta_{\mathrm{Mon}(v)}, \rho_{\mathrm{Mon}(v)}, E_{\mathrm{Mon}(v)})$$

with

- local variables $V_{\mathrm{Mon}(v)} = \emptyset$,
- environment variables

$$\begin{aligned} E_{\mathrm{Mon}(v)} = \{&v_{out}, v_{in}, \mathit{written}(v)_{out}, \mathit{written}(v)_{in}, \mathit{changed}(v)_{out}, \\ &\mathit{changed}(v)_{in}, \mathit{read}(v)_{out}, \mathit{read}(v)_{in}\} \\ \cup \{&C.v_{out}, C.v_{in}, C.\mathit{written}(v)_{out}, C.\mathit{written}(v)_{in}, \\ &C.\mathit{changed}(v)_{out}, C.\mathit{changed}(v)_{in}, C.\mathit{read}(v)_{out}, \\ &C.\mathit{read}(v)_{in} \mid C \in \mathit{Comp}(A, v)\}, \end{aligned}$$

- the set of initial valuations

$$\begin{aligned} \Theta_{\mathrm{Mon}(v)} = \{\sigma \mid &\sigma(v_{out}) = d_{init}^{type(v)},\ \sigma(C.v_{in}) = d_{init}^{type(v)}, \\ &\sigma(\mathit{written}(v)_{out}) = \sigma(\mathit{changed}(v)_{out}) = \sigma(\mathit{read}(v)_{out}) = \mathit{false}, \\ &\sigma(C.\mathit{written}(v)_{in}) = \sigma(C.\mathit{changed}(v)_{in}) = \sigma(C.\mathit{read}(v)_{in}) \\ &= \mathit{false}\}, \end{aligned}$$

and

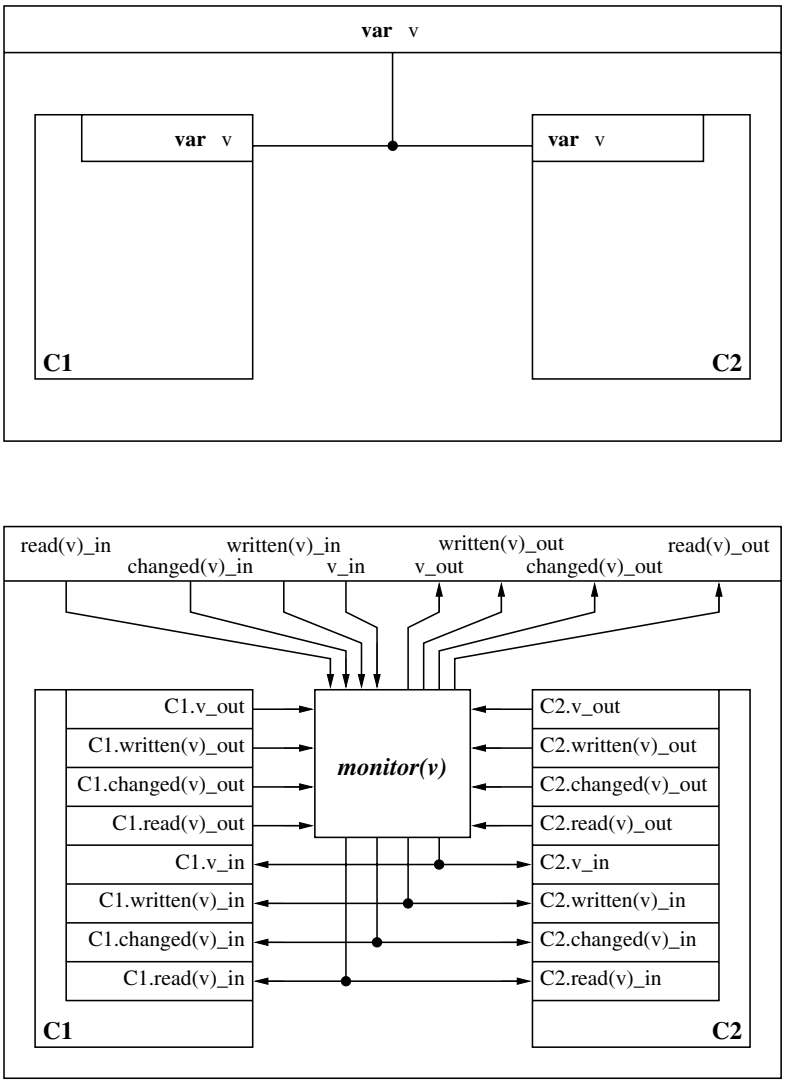


Fig. 8. Monitor concept

– transition relation

$$\rho_{\mathrm{Mon}(v)} = \{(\sigma, \sigma') \mid \sigma' \models \mathit{consistent}(v)\},$$

where the predicate $\mathit{consistent}(v)$ describes that the distributed values of v and the changed and written events are consistent with the observed values distributed by the components.

The predicate $\mathit{consistent}(v)$ is defined by

$$\begin{aligned}
&\mathit{consistent}(v) \;:= \\
&\quad \Big(\bigvee_{C \in Comp(A,v)} \big(v_{out} = C.v_{out} \wedge C.\mathit{written}(v)_{out}\big) \\
&\qquad \vee \big(\bigvee_{C \in Comp(A,v)} v_{out} = C.v_{out} \wedge \bigwedge_{C \in Comp(A,v)} \neg C.\mathit{written}(v)_{out}\big)\Big) \\
&\quad \wedge \; \bigwedge_{C \in Comp(A,v)} C.v_{in} = v_{in} \\
&\quad \wedge \Big(\mathit{written}(v)_{out} \Leftrightarrow \bigvee_{C \in Comp(A,v)} C.\mathit{written}(v)_{out}\Big) \\
&\quad \wedge \; \bigwedge_{C \in Comp(A,v)} C.\mathit{written}(v)_{in} = \mathit{written}(v)_{in} \\
&\quad \wedge \Big(\mathit{changed}(v)_{out} \Leftrightarrow \bigvee_{C \in Comp(A,v)} \big(C.\mathit{changed}(v)_{out} \wedge v_{out} = C.v_{out}\big)\Big) \\
&\quad \wedge \; \bigwedge_{C \in Comp(A,v)} C.\mathit{changed}(v)_{in} = \mathit{changed}(v)_{in} \\
&\quad \wedge \Big(\mathit{read}(v)_{out} \Leftrightarrow \bigvee_{C \in Comp(A,v)} C.\mathit{read}(v)_{out}\Big) \\
&\quad \wedge \; \bigwedge_{C \in Comp(A,v)} C.\mathit{read}(v)_{in} = \mathit{read}(v)_{in} \;\; .
\end{aligned}$$

If v is not in the interface of A, i.e. if v is local to A we have to modify the monitor described above by removing the variables v_{in}, v_{out}, $\mathit{written}(v)_{in}$, $\mathit{written}(v)_{out}$, $\mathit{changed}(v)_{in}$, $\mathit{changed}(v)_{out}$, $\mathit{read}(v)_{in}$, $\mathit{read}(v)_{out}$. The computed value for v_{out} is directly copied to the individual inputs $C.v_{in}$. The same is done for the written, changed, and read events. I.e. the predicate has the following form:

$$\begin{aligned}
&\mathit{consistent}(v) \;:= \\
&\quad \Big(\bigvee_{C \in Comp(A,v)} \big(\bigwedge_{C' \in Comp(A,v)} C'.v_{in} = C.v_{out} \wedge C.\mathit{written}(v)_{out}\big) \\
&\qquad \vee \big(\bigwedge_{C \in Comp(A,v)} C.v_{in} = C.v_{out} \wedge \bigwedge_{C \in Comp(A,v)} \neg C.\mathit{written}(v)_{out}\big)\Big) \\
&\quad \wedge \Big(\bigwedge_{C' \in Comp(A,v)} \big(C'.\mathit{written}(v)_{in} \Leftrightarrow \bigvee_{C \in Comp(A,v)} C.\mathit{written}(v)_{out}\big)\Big) \\
&\quad \wedge \Big(\bigwedge_{C' \in Comp(A,v)} \big(C'.\mathit{changed}(v)_{in} \Leftrightarrow \\
&\qquad\qquad \bigvee_{C \in Comp(A,v)} \big(C.\mathit{changed}(v)_{out} \wedge C.v_{in} = C.v_{out}\big) \;\big)\Big) \\
&\quad \wedge \Big(\bigwedge_{C' \in Comp(A,v)} \big(C'.\mathit{read}(v)_{in} \Leftrightarrow \bigvee_{C \in Comp(A,v)} C.\mathit{read}(v)_{out}\big)\Big) \;\; .
\end{aligned}$$

Global Assumptions. As the environment is responsible for the resolution of the values of shared variables, the environment should also set the written and changed events in accordance with the given input value.

$$written(v)_{in} = false \;\Rightarrow\; changed(v)_{in} = false$$
$$written(v)_{in} = false \;\Rightarrow\; written(v)_{out} = false \wedge v_{in} = v_{out}$$
$$read(v)_{out} = true \;\Rightarrow\; read(v)_{in} = true\ .$$

6.3 Monitor for Events

The difference for the handling of events is that the resulting value is given by a disjunction of all values of the relevant components. I.e. we do not deterministically select a value – true or false – but instead raise an event if some component raise that event. There is only one exception given by the event *stable* as a system is stable iff all its subsystems are stable. Hence, in that case we have to perform a conjunction.

The monitor $\Phi_{\mathrm{Mon}(stable)}$ is given by

$$\begin{aligned}
&E := \{stable, stable_env\} \cup \{C.stable, C.stable_env \mid C \in Comp(A)\}\\
&V := \emptyset\\
&\Theta := \{\sigma \mid \sigma(stable) = true, \sigma(C.stable_env) = true \text{ for all } C\}\\
&\rho := \{(\sigma,\sigma') \mid \sigma'(stable) = false \text{ iff } \exists C \text{ with } \sigma'(C.stable) = false,\\
&\qquad\quad \sigma'(C.stable_env) = false \text{ iff}\\
&\qquad\quad \sigma'(stable_env) = false \text{ or } \exists C' \neq C \text{with } \sigma'(C'.stable) = false\}\ .
\end{aligned}$$

The monitor $\Phi_{\mathrm{Mon}(step_div)}$ is given by

$$\begin{aligned}
&E := \{step_div, step_div_env\} \cup \{C.step_div, C.step_div_env \mid C \in Comp(A)\}\\
&V := \emptyset\\
&\Theta := \{\sigma \mid \sigma(step_div) = false, \sigma(C.step_div_env) = false \text{ for all } C\}\\
&\rho := \{(\sigma,\sigma') \mid \sigma'(step_div) = true \text{ iff } \exists C \text{ with } \sigma'(C.step_div) = true,\\
&\qquad\quad \sigma'(C.step_div_env) = true \text{ iff}\\
&\qquad\quad \sigma'(step_div_env) = true \text{ or } \exists C' \neq C \text{with } \sigma'(C'.step_div) = true\}\ .
\end{aligned}$$

The monitor $\Phi_{\mathrm{Mon}(e)}$ for a bidirectional event e is given by

$$\begin{aligned}
&E := \{e_{out}, e_{in}\} \cup \{C.e_{out}, C.e_{in} \mid C \in Comp(A, e)\}\\
&V := \emptyset\\
&\Theta := \{\sigma \mid \sigma(e_{out}) = false, \sigma(C.e_{in}) = false \text{ for all } C \in Comp(A, e)\}\\
&\rho := \{(\sigma,\sigma') \mid \sigma'(e_{out}) = true \text{ iff } \exists C \in Comp(A, e) \text{ with } \sigma'(C.e_{out}) = true\\
&\qquad\quad \text{and } \sigma'(C.e_{in}) = \sigma'(e_{in}) \text{ for all } C \in Comp(A, e)\}\ .
\end{aligned}$$

If e is not in the interface of A the variables e_{in} and e_{out} are removed from the set E and ρ is modified in the following way:

$$\rho := \{(\sigma,\sigma') \mid \sigma'(C'.e_{in}) = true \text{ iff } \exists C \in Comp(A, e) \text{ with } \sigma'(C.e_{out}) = true\}$$

6.4 Composition of Systems

After defining the various monitors we can give a semantics to the composite system $A = A_0 \| \ldots \| A_n$ by building the parallel composition of the individual

transition systems $rename(\Phi_i, A_i)$ together with the transition systems of the relevant monitors. Then all objects not visible outside of A are hidden by the hide operator. Let $var = \bigcup_{i=0}^{n}\{v \in e_int(A_i) \mid dir(v) = inout\}$ and $bevents = \bigcup_{i=0}^{n}\{e \in e_int(A_i) \mid dir(e) = inout\}$.

$$\begin{aligned} CSTS[\![A]\!] = hide\big(&rename(\Phi(A_0), A_0) \parallel \ldots \parallel rename(\Phi(A_n), A_n) \\ &\parallel \|_{v \in var} \Phi_{\mathrm{Mon}(v)} \parallel \|_{e \in bevents} \Phi_{\mathrm{Mon}(e)} \parallel \Phi_{\mathrm{Mon}(stable)} \parallel \Phi_{\mathrm{Mon}(step_div)} \ , \\ &\bigcup_{i=0}^{n} full_int(A_i) \setminus full_int(A)\big) \end{aligned}$$ [10]

The hiding operator *hide* removes some of the visible objects from the interface. This is simply defined by moving the variables from E to V, i.e.

$$hide((V, \Theta, \rho, E),\ W) = (V, \Theta, \rho, E \setminus W) \ .$$

7 Conclusion

In this paper we have defined a real-time semantics for statecharts based on the STATEMATE system. We have introduced a step semantics as well as a super-step semantics. These formal semantics allow the integration of the STATEMATE system within an environment for formal verification techniques. In current projects this has been achieved by providing a translation from STATEMATE into a finite state machine (FSM) description used as an interface to a model checker [3]. The compositional approach used in this paper does not only allow to check properties of statecharts by model checking, but also the use of compositional verification methods which is required for large designs due to the state explosion problem.

References

1. Martín Abadi and Leslie Lamport. Composing specifications. In Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness*, Lecture Notes in Computer Science430, pages 1–41. Springer-Verlag, 1990.
2. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.
3. U. Brockmeyer and G. Wittich. Tamagotchis need not die – verification of Statemate designs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, 1998. (To appear).
4. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference*, June 1990.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of theFifth Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[10] To be more precise, we hide the events and variables of $\bigcup_{i=0}^{n} E_i' \setminus full_int(A)$, where E_i' is the interface of A_i after renaming.

6. Werner Damm and Johannes Helbig. Linking visual formalisms: A compositional proof system for statecharts based on symbolic timing diagrams. In *Proc. IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 337–356, 1994.
7. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development od complex reactive systems. *IEEE Transactions on Software Engineering*, 16:403 – 414, 1990.
8. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. Technical Report CS95-31, The Weizmann Institute of Science, Rehovot, 1995.
9. D. Harel, A. Pnueli, J.P. Schmidt, and R.Sherman. On the formal semantics of statecharts. In *Proceeding IEEE Symposium on Logic in Computer Science*, pages 54 – 64, 1987.
10. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.
11. David Harel. On visual formalism. *CACM*, 31:514–530, 1988.
12. David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series F13, pages 477–498. Springer-Verlag, 1985.
13. David Harel and Michal Politi. Modeling reactive systems with statecharts: The statemate approach. Technical report, 1996.
14. J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A compositional axiomatization of statecharts. *TCS*, 101:289–335, 1992.
15. Bernhard Josko. *Modular Specification and Verification of Reactive Systems.* Habilitationsschrift, Universität Oldenburg, 1993.
16. Andrea Maggiolo-Schettini, Andriano Peron, and Simone Tini. Equivalences of statecharts. In *CONCUR 96, Seventh Int. Conf. on Concurrency Theory, Pisa*, LNCS 1119, pages 687–702. Springer-Verlag, 1996.
17. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
18. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems. Specification.* Springer-Verlag, 1992.
19. Kenneth L. McMillian. *Symbolic Model Checking. An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, 1992.
20. A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 526. Springer-Verlag, 1991.
21. Amir Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series, Vol. F13, pages 123–144. Springer-Verlag, 1985.
22. Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE'91 Parallel Architectures and Languages, Volume I: Parallel Architectures and Algorithms*, Lecture Notes in Computer Science 505, pages 137–149. Springer-Verlag, 1991.
23. Andrew C. Uselton and Scott A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *CONCUR 94*, LNCS 836, pages 2–17. Springer-Verlag, 1994.
24. M. van der Beek. A comparison of statechart variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science 863, pages 128–148. Springer-Verlag, 1994.

Deductive Verification of Modular Systems *

Bernd Finkbeiner, Zohar Manna and
Henny B. Sipma
finkbein|manna|sipma@cs.stanford.edu

Computer Science Department, Stanford University
Stanford, CA. 94305

Abstract. Effective verification methods, both deductive and algorithmic, exist for the verification of global system properties. In this paper, we introduce a formal framework for the modular description and verification of parameterized fair transition systems. The framework allows us to apply existing global verification methods, such as verification rules and diagrams, in a modular setting. Transition systems and transition modules can be described by recursive module expressions, allowing the description of hierarchical systems of unbounded depth. Apart from the usual *parallel composition*, *hiding* and *renaming* operations, our module description language provides constructs to *augment* and *restrict* the module interface, capablilities that are essential for recursive descriptions. We present proof rules for property inheritance between modules. Finally, *module abstraction* and induction allow the verification of recursively defined systems. Our approach is illustrated with a recursively defined arbiter for which we verify mutual exclusion and eventual access.

1 Introduction

In this paper we introduce a formal framework for the modular description and mechanical, modular verification of parameterized fair transition systems. The framework provides a system description language that allows concise, modular, possibly recursive and parameterized system descriptions. It is sufficiently expressive to represent various modes of communication between modules, and enables the reuse of code, that is, transition modules can be described once and referred to multiple times in a system description. The framework supports a variety of analysis techniques, such as verification rules, verification diagrams, model checking, abstraction and refinement, the results of which can be seamlessly combined in the course of a single proof.

Our framework extends the principles for modular verification presented in [MP95b] and those formulated for I/O automata [LT89, LT87]. The basic building block of our system description language is a transition module, consisting

* This research was supported in part by the National Science Foundation under grant CCR-95-27927, the Defense Advanced Research Projects Agency under NASA grant NAG2-892, ARO under grant DAAH04-95-1-0317, ARO under MURI grant DAAH04-96-1-0341, and by Army contract DABT63-96-C-0096 (DARPA).

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 239-275, 1998.
Springer-Verlag Berlin Heidelberg 1998

of an interface that describes the interaction with the environment, and a body that describes its actions. Communication between a module and its environment can be asynchronous, through shared variables, and synchronous, through synchronization of transitions. More complex modules can be constructed from simpler ones by (recursive) module expressions, allowing the description of hierarchical systems of unbounded depth. Module expressions can refer to (instances of parameterized) modules defined earlier by name, thus enabling the reuse of code and the reuse of properties proven about these modules. Apart from the usual *hiding* and *renaming* operations, our module description language provides a construct to *augment* the interface with new variables that provide a summary value of multiple variables within the module. Symmetrically, the *restrict* operation allows the module environment to combine or rearrange the variables it presents to the module. As we will see later, these operations are essential for the *recursive* description of modules, to avoid an unbounded number of variables in the interface.

The basis of our proposed verification methodology is the notion of modular validity, as proposed in [Pnu85, Cha93, MP95b]. An LTL property holds over a module if it holds over any system that includes that module (taking into account variable renamings). That is, no assumptions are made about the module's environment. Therefore, modular properties are inherited by any system that includes the module. Many useful, albeit simple properties can be proven modularly valid. However, as often observed, not many interesting properties are modularly valid, because most properties rely on some cooperation by the environment.

The common solution to this problem is to use some form of *assumption-guarantee* reasoning, originally proposed by Misra and Chandy [MC81] and Jones [Jon83]. Here, a modular property is an assertion that a module satisfies a guarantee G, provided that the environment satisfies the assumption A. An assumption–guarantee property can be formulated as an implication of LTL formulas with past operators [BK84, GL94, JT95]. Thus in LTL there is no need for compositional proof rules dealing with the discharge of assumptions as for example in [AL93]. In our framework these rules are subsumed by *property inheritance* rules: systems that are composed of modules by parallel composition directly inherit properties of their components. In this way assumptions can be discharged either by properties of other components, or by the actual implementation of the composite module. If the assumption cannot be discharged, it is simply carried over to the composite module. This flexibility in our approach as to when and how assumptions are discharged is similar to the one described by Shankar [Sha93]. In particular it does not require the verifier to anticipate assumptions that could be made on a module by other modules [Sha98].

Our verification methodology supports both composition and decomposition, as defined by Abadi and Lamport [AL93]. In *compositional* reasoning, we analyze a component without knowing the context it may be used in. We therefore state and prove properties that express explicitly under what assumptions on the environment a certain guarantee is given. This approach is taken by our mod-

ular proof rule and the *property inheritance* rules. In *decompositional* reasoning the composite system is analyzed by looking at one module at a time. In our experience both methods can be used during a system verification effort. Compositional reasoning is used to establish invariants and simple liveness properties about components. Then the system is analyzed from the top down, using the previously proven modular properties, and using abstraction to hide details that are irrelevant to the property at hand. We provide a modular inheritance rule that allows modules in expressions to be replaced with simpler modules, such that properties proven over the system containing the simpler module are also valid over the system containing the actual module. Alternatively this can also be used in the other direction, in design. Any (underspecified) module may be refined into a more detailed one, while preserving the properties proven so far.

A convenient abstraction, which can be constructed automatically, is the *interface abstraction*, which represents only the information in the interface and ignores all implementation details. Using the interface abstraction in place of a module is especially useful when we consider recursively described systems of unbounded depth: in this case the implementation details are in fact unknown. Such systems fit in naturally in our framework: we combine the decompositional interface abstraction with a compositional *induction rule*.

1.1 Example

We illustrate our description language and verification methodology with the verification of a recursively defined binary *arbiter* that guarantees mutual exclusion to a critical resource. A number of clients can each request access, and the arbiter gives a grant to one client at a time. Our design, shown in Figure 1, is based on a similar example in [Sta94]. The arbiter is described as a tree of nodes,

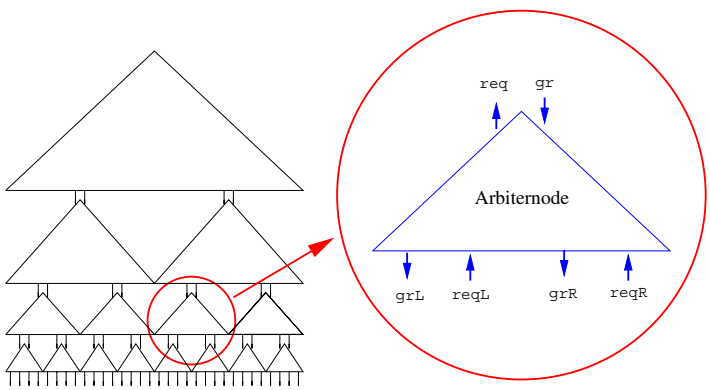


Fig. 1. A Hierarchical Arbiter.

where a tree consists of two subtrees and a node that guarantees mutual exclusion between the two subtrees. Thus while the simple algorithm represented in the nodes deals with two clients at a time, a tree of height h ensures mutual exclusion for 2^h clients. Local correctness proofs for implementations of the nodes are discussed in [Dil88], and safety properties of arbiter trees are verified in [GGS88].

1.2 STeP

Part of the framework presented here has been implemented in STeP (Stanford Temporal Prover), a tool for the deductive and algorithmic verification of reactive, real-time and hybrid systems [BBC+96, BBC+95, BMSU97]. STeP implements *verification rules* and *verification diagrams* for deductive verification. A collection of decision procedures for built-in theories, including integers, reals, datatypes and equality is combined with propositional and first-order reasoning to simplify verification conditions, proving many of them automatically. The proofs in the arbiter example, presented in Section 5, have been performed with STeP.

1.3 Outline

The rest of the paper is organized as follows. In Section 2 we introduce our computational model, fair transition systems, and specification language, LTL. In Section 3 we define transition modules and parameterized transition modules, and present the syntax and semantics of our module description language. Here we give a full description of the arbiter example. In Section 4 we propose a modular verification rule and devise verification rules for property inheritance across the operations of our module description language. We discuss modular abstraction and induction as techniques that can be used to prove properties over recursively defined modules. In Section 5 we verify mutual exclusion and eventual access for the arbiter using the rules presented in Section 4.

2 Preliminaries

2.1 Computational Model: Transition Systems

As the underlying computational model for verification we use *fair transition systems* (FTS) [MP95b].

Definition 1 Fair transition System. A fair transition system $\Phi = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ consists of

- V: A finite set of typed *system variables*. A *state* is a type-consistent interpretation of the system variables. The set of all states is called the *state space*, and is designated by Σ. We say that a state s is a p-state if s satisfies p, written $s \models p$.

- Θ: The *initial condition*, a satisfiable assertion characterizing the initial states.
- $\mathcal{T}$: A finite set of *transitions*. Each transition $\tau \in \mathcal{T}$ is a function

$$\tau : \Sigma \mapsto 2^{\Sigma}$$

  mapping each state $s \in \Sigma$ into a (possibly empty) set of τ-successor states, $\tau(s) \subseteq \Sigma$. Each transition τ is defined by a *transition relation* $\rho_\tau(V, V')$, a first-order formula in which the unprimed variables refer to the values in the current state s, and the primed variables refer to the values in the next state s'. Transitions may be parameterized, thus representing a possibly infinite set of similar transitions.
- $\mathcal{J} \subseteq \mathcal{T}$: A set of *just* transitions.
- $\mathcal{C} \subseteq \mathcal{T}$: A set of *compassionate* transitions.

Definition 2 Runs and Computations. A *run* of an FTS $\Phi = \langle V, \Theta, \mathcal{T}, \mathcal{J}, \mathcal{C} \rangle$ is an infinite sequence of states $\sigma : s_0, s_1, s_2, \ldots$, such that

- *Initiation*: s_0 is initial, that is, $s_0 \models \Theta$.
- *Consecution*: For each $j = 0, 1, \ldots$, s_{j+1} is a τ-successor of s_j, that is, $s_{j+1} \in \tau(s_j)$ for some $\tau \in \mathcal{T}$. We say that τ is *taken* at s_i if S_{i+1} is a τ-successor of s_i.

A *computation* of an FTS Φ is a run σ of Φ such that

- *Justice*: For each transition $\tau \in \mathcal{J}$, it is not the case that τ is continuously enabled beyond some point in σ without being taken beyond that point.
- *Compassion*: For each transition $\tau \in \mathcal{C}$, it is not the case that τ is infinitely often enabled beyond a certain point in σ without being taken beyond that point.

Definition 3 Parameterized Transition System. Let $\mathcal{F}$ be the class of all fair transition systems, and $P = \langle p_1, \ldots, p_n \rangle$ a tuple of parameters with type $t_1, \ldots, t_n$. Then a *parameterized transition system* $\mathbf{F} : t_{p1} \times \ldots \times t_{pn} \mapsto \mathcal{F}$ is a function from the input parameters to fair transition systems. Given a parameterized fair transition system $\mathbf{F}$, and a list of values $a_1, \ldots, a_n$, type consistent with $p_1, p_2, \ldots, p_n$, then the *instance* $\mathbf{F}(a_1, a_2, \ldots, a_n)$ denotes the fair transition system where all the references to $p_1, p_2, \ldots, p_n$ have been replaced by $a_1, a_2, \ldots, a_n$.

An infinite sequence of states $\sigma : s_0, s_1, \ldots$ is a computation of a parameterized transition system $\mathbf{F}$ if σ is a computation of $\mathbf{F}(a)$ for some a. Thus the set of computations of a parameterized system is the union of the sets of computations of all its instances.

2.2 Specification Language

We use linear-time temporal logic (LTL) with past operators to specify properties of reactive systems. LTL formulas are interpreted over infinite sequences of states. The truth-value of a formula for a given model is evaluated at the initial position of the model. We say that a temporal formula holds at a particular state of a model if it is true of the sequence which starts at that state. Below we only define those temporal operators used in the rest of the example. For the full set, the reader is referred to [MP91].

Given a model $\sigma = s_0, s_1, \ldots$, the temporal operators $\Box, \Diamond, \ominus$ are defined as follows:

$\Box p$ holds at state s_j iff p is true for all states s_i, $i \geq j$;
$\Diamond p$ holds at state s_j iff p is true for some state s_i, $i \geq j$;
$\ominus p$ holds at state s_j iff s_j is not the first state and p holds at state s_{j-1};

$\Box$ and $\Diamond$ are called *future* operators, while $\ominus$ is a *past* operator. We will refer to formulas that contain only past operators as *past formulas*. In this paper we do not allow temporal operators to appear within the scope of a quantifier. A formula containing no temporal operators is called a *state-formula* or an *assertion*.

Given an FTS $\mathcal{S}$, we say that a temporal formula φ is $\mathcal{S}$-*valid* if every computation of $\mathcal{S}$ satisfies φ, and write $\mathcal{S} \models \varphi$.

3 Transition Modules and Systems

A transition system describes a closed system, that is, a system that does not interact with its environment. To enable reasoning about individual components of a transition system, we define *transition modules*. A transition system is then constructed from interacting transition modules. Transition modules can communicate with their environment via shared variables or via synchronization of transitions.

A transition module consists of two parts: an interface and a body. The interface describes the interaction between the module and its environment; it consists of a set of interface variables and a set of transition labels. We distinguish four types of interface variables. *Constants* in the interface are often used as parameters for the module; they have a fixed value throughout a computation. *Input variables* belong to the environment, they cannot be changed by the module. *Output variables* are owned by the module and cannot be changed by the environment. Finally, *shared variables* can be modified by both the module and the environment.

The transition labels in the interface refer to transitions in the body. Such exported transitions can synchronize with other transitions with the same label in the environment. The result of synchronization is a new transition whose transition relation is the conjunction of the transition relations of the original transitions. One transition may have multiple labels, so it may synchronize with multiple transitions.

The module body is similar in structure to a fair transition system; it has its own set of private variables that cannot be observed nor modified by the environment. The transitions in the body have to respect the interface: they may modify private variables, output and shared variables, but not input variables or constants. Similarly, the initial condition cannot constrain the input variables or constants.

To be able to prove properties about modules we associate with each module a transition system such that the set of computations of the associated transition system is a superset of the set of computations of any system that includes the module. Having these semantics for modules allows us to "lift" properties of modules to properties of the whole system, that is, if a property has been proven valid over a module, then a corresponding property can be inferred for any system that includes that module.

Transition modules can be described directly, by giving its interface and body, or they can be constructed from other modules using *module expressions*.

3.1 Transition Module: Definition

The basic building block of a transition module system is the transition module.

Vocabulary We assume that all variables in a transition module description are taken from a universal set of variables $\mathcal{V}$, called the *variable vocabulary*, and that all transition labels are taken from a universal set of labels called $\mathcal{T}_{id}$.

Definition 4 Transition Module. A *transition module* $\mathbf{M} = \langle I, B \rangle$ consists of an interface declaration $I = \langle V, T \rangle$ and a body $B = \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C} \rangle$. The interface components are

- $V \subseteq \mathcal{V}$: the set of interface variables, partitioned in four subsets as follows:
  - V_c: constants, possibly underspecified, which cannot be modified;
  - V_i: input variables, which can only be modified by the environment;
  - V_o: output variables, which can only be modified by the module;
  - V_s: shared variables, which can be modified by both the module and the environment.
- $T \subseteq \mathcal{T}_{id}$: a set of transition labels. The transitions corresponding to these labels may synchronize with transitions in the environment.

A transition module is called *closed* if both the set of shared variables and the set of exported transitions are empty.

The components of the body are:

- V_p: a set of private variables, which can neither be modified nor observed by the module's environment.
- Θ: the initial condition, an assertion over $V_p \cup V_o$.

- $\mathcal{T}$: a set of transitions, specified in the same way as described in Section 2; we require that
$$\rho_\tau \rightarrow \bigwedge_{v \in V_c} v' = v \qquad \text{for all } \tau \in \mathcal{T}$$
- $\lambda \subseteq \mathcal{T} \times \mathcal{T}_{id}$: a transition labeling relation, relating transitions to their labels. Note that multiple transitions can have the same label, and that a single transition may have multiple labels. We require that the labeling relation λ relates every label in the exported transitions T to at least one transition in $\mathcal{T}$, that is
$$\forall id \in T \,.\, \exists \tau \in \mathcal{T} \,.\, (\tau, id) \in \lambda$$
For internal transitions, i.e., transitions that do not have a label in T, we require that they do not modify the input variables, that is,
$$\rho_\tau \rightarrow \bigwedge_{v \in V_i} v' = v \qquad \text{for all } \tau \in \{\tau \,|\forall id \in T \,.\, (\tau, id) \notin \lambda\}$$
- $\mathcal{J} \subseteq \mathcal{T}$: the set of just transitions.
- $\mathcal{C} \subseteq \mathcal{T}$: the set of compassionate transitions.

Modules can be *parameterized*, to represent, similar to the parameterized transition systems introduced in Section 2, functions from the parameters to transition modules.

Definition 5 Parameterized Transition Module. Let $\mathcal{M}$ be the class of all modules, and $P = \langle p_1, \ldots, p_n \rangle$ a set of parameters with type $t_1, \ldots, t_n$. Then a *parameterized transition module* (PTM) $\mathbf{M} : t_1 \times \ldots \times t_n \mapsto \mathcal{M}$ is a function from parameters to transition modules.

3.2 Example: Arbiternode

As described in Section 1, an arbiter is a device that guarantees mutual exclusion to a critical resource. Figure 1 shows the hierarchical design for an arbiter dealing with 2^n clients, which repeatedly uses the module `ArbiterNode` (shown enlarged). An `ArbiterNode` establishes mutual exclusion between two clients: its "left" and "right" client. In this section we only discuss the `ArbiterNode` module; in Section 3.7 we will return to the complete arbiter design.

The two clients of the `ArbiterNode` can request the grant by setting their request bits, `reqL` and `reqR` for the left and right client, respectively. If the `ArbiterNode` owns the grant, that is, if the `gr` bit is set, it can pass the grant on to a client by setting the client's grant bit, `grL` or `grR`. The client can subsequently release the grant by resetting its request bit, which causes the arbiter to reset the grant bit (`grL`, `grR`) and either give the grant to its other client, or release its own grant by resetting `req`.

Figure 2 shows the description of the `ArbiterNode` module in STeP input format. In STeP, variables declared as **external in**, **out**, **external out** refer

```
Module ArbiterNode:

external in gr, reqL, reqR : bool
out req, grL, grR : bool where !req /\ !grL /\ !grR

Transition RequestGrant Just:
    enable !gr /\ (reqL \/ reqR)
    assign req:=true

Transition GrantLeft Just:
    enable gr /\ req /\ reqL /\ !grR
    assign grL:=true

Transition GrantRight Just:
    enable gr /\ req /\ reqR /\ !grL /\ !reqL
    assign grR:=true

Transition ReleaseLeft Just:
    enable gr /\ req /\ !reqL /\ grL
    assign grL:=false, grR:=reqR, req:=reqR

Transition ReleaseRight Just:
    enable gr /\ req /\ !reqR /\ grR
    assign grR:=false, req:=false

EndModule
```

Fig. 2. `ArbiterNode` module.

to input, output and shared variables, respectively. The keywords **enable** and **assign** allow a description of the transition relation in a programming-like notation: the transition relation is the conjunction of the enabledness condition, a relation `a' = b` for each assignment `a := b`, and `c' = c` for any variable `c` that is not explicitly assigned a new value.

The left client enjoys a slightly higher priority than the right client: if the node has the grant, and both the left and the right client request it, the grant will be given to the left client, by transition `GrantLeft`. On the other hand, the node releases the grant after it is released by the right client, even if the left client requests it. This is to make sure that the node does not keep the grant forever: the grant is given at most once to each client before it is released again.

3.3 Associated Transition System

As mentioned before, it is our objective to reason about modules and use the results as lemmas in the proof of properties that use these modules. To do

so, we relate modules to transition systems: we define the associated transition system of a module, such that the set of computations of the associated transition system is a superset of the set of computations of any system that includes the module. We say that the set of computations of a module is equal to the set of computations of its associated transition system.

To ensure that the set of computations of a module includes all computations of a system including that module, we cannot make any assumptions about the environment of the module, except that it will not modify the module's private and output variables, and that it will not synchronize with the module's internal transitions. We model this environment by a single transition, called the *environment transition*, τ_{env} with transition relation

$$\rho_{\tau_{env}} : \bigwedge_{v \in V_c \cup V_o \cup V_p} v' = v$$

where V_c are the constants of the module, and V_o and V_p are its output and private variables, respectively.

Given a transition module $\mathbf{M} = \langle I, B \rangle$, with $I = \langle V, T \rangle$ and $B = \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C} \rangle$ we define its *associated transition system*

$$S_{\mathbf{M}} = \langle V_p \cup V, \Theta, \mathcal{T}^*, \mathcal{J} - \mathcal{T}_{exp}, \mathcal{C} - \mathcal{T}_{exp} \rangle$$

where $\mathcal{T}^*$ denotes the set of associated transitions, i.e.,

$$\mathcal{T}^* = \left\{ \tau^* \;\middle|\; \exists \tau \in \mathcal{T}_{int} \,.\, \rho_{\tau^*} = \rho_\tau \wedge \bigwedge_{v \in V_i} v' = v \right\} \cup \mathcal{T}_{exp} \cup \{\tau_{env}\}$$

and $\mathcal{T}_{exp}$ is the set of exported transitions, i.e., transitions in $\mathcal{T}$, that have an exported label

$$\mathcal{T}_{exp} = \{\tau \mid \exists id \in T \,.\, (\tau, id) \in \lambda\}$$

$\mathcal{T}_{int}$ is the set of internal transitions, i.e., transitions in $\mathcal{T}$, that have no exported label

$$\mathcal{T}_{int} = \{\tau \mid \forall id \in T \,.\, (\tau, id) \notin \lambda\}$$

The transition relation of the internal transitions is modified to account for the fact that these transitions, in contrast to the exported transitions, are guaranteed to preserve the values of the input variables, since they cannot synchronize with the environment. The fairness conditions are removed from the exported transitions, because we cannot make any assumptions about the enabling condition of the transitions with which they may synchronize (the enabling condition may be false), and therefore we can no longer assume that a just transition must eventually be taken as long as the local enabling condition continues to hold.

Example: The `ArbiterNode` presented in the previous section has three output variables: `grL, grR` and `req`, and no private variables. All transitions are internal. The associated transition system therefore consists of the module transitions shown in Figure 2 and the environment transition with the transition relation

$$\rho_{\tau_{env}} : \mathtt{grL} = \mathtt{grL}' \wedge \mathtt{grR} = \mathtt{grR}' \wedge \mathtt{req} = \mathtt{req}'$$

Parameterized transition modules. Parameterized transition modules have associated parameterized transition systems: Let $\mathbf{M} : P \mapsto \mathcal{M}$ be a parameterized module, then the associated parameterized transition system $S_{\mathbf{M}} : P \mapsto \mathcal{M}$ maps each parameter value p to the transition system associated with $\mathbf{M}(p)$.

3.4 Module Systems: Definition

We defined the notion of transition modules. In modular verification, however, we are not interested in single modules, but rather in a collection of modules that, together, describe the system behavior. For this purpose we define *module systems*.

Vocabulary We assume a universal set of module identifiers M_{id}. Let $\mathcal{M}$ denote the set of all modules.

Definition 6 Module System. A *module system* $\Psi = \langle \mathbf{Menv}, \mathbf{M_{main}} \rangle$ consists of a module environment $\mathbf{Menv} : M_{id} \mapsto \mathcal{M}$ and a designated main module $\mathbf{M_{main}}$.

3.5 Module Systems: Syntax

Module systems are described by a list of module declarations that define the module environment, followed by a module expression that defines the main module. The module declarations assign modules, also defined by module expressions, to module identifiers.

Module expressions

If $\mathcal{E}$, $\mathcal{E}_1 \ldots \mathcal{E}_n$, are well-formed module expressions, then so are the following:

- $\langle I, B \rangle$, a *direct* module description, defining the interface I and the body B of a transition module.
- $id(e)$, where id is a module identifier, and e is a (possibly empty) list of expressions over constant symbols and variables, indicating a *reference* to another module.
- $(g_1 : \mathcal{E}_1); \ldots; (g_n : \mathcal{E}_n)$, where $g_1 \ldots g_n$ are first-order formulas, denoting a *case distinction*: This allows to describe differently structured modules for different parameter values.
- $(\mathcal{E}_1 \parallel \mathcal{E}_2)$. The *parallel composition* operator merges two modules into one module, keeping private variables apart, merging the interfaces, and synchronizing transitions that have the same label.
- $\mathbf{Hide}(X, \mathcal{E})$, where X is a set of variables, or a set of transition labels.
  The **Hide** operator removes variables or transition labels from a module's interface. Removing variables from the interface makes them unavailable

for reading or writing by the module's environment. Removing transition labels makes the corresponding transitions unavailable for synchronization under that label (a single transition may have multiple labels, so it may still synchronize under other labels).

- **Rename**$(\beta, \mathcal{E})$, where β is a variable substitution $\beta : \mathcal{V} \mapsto \mathcal{V}$, or a transition label substitution $\beta : \mathcal{T}_{id} \mapsto \mathcal{T}_{id}$. The **Rename** operator renames variables or transition labels in the interface.
- **Augment**$(\beta, \mathcal{E})$, where β is a mapping from variables to expressions over variables and constants.
  The purpose of the augmentation operation is to create new variables in the interface that maintain the same value as the corresponding value of the expression. To ensure that the module can maintain these values, the expression may contain only private and output variables.
- **Restrict**$(\beta, \mathcal{E})$, where β is a mapping from variables to expressions over variables and constants.
  The purpose of the restrict operation is to replace input variables in the interface by expressions over other input variables.

Module Systems

If $\mathcal{E}_1 \ldots \mathcal{E}_n, \mathcal{E}_{main}$ are well-formed module expressions, $id_1 \ldots id_n$ are module identifiers and $P_1 \ldots P_n$ are (possibly empty) lists of formal parameters, then

$$id_1(P_1) = \mathcal{E}_1 \ ; \ \ldots \ ; \ id_n(P_n) = \mathcal{E}_n \ ; \ \mathcal{E}_{main}$$

is a well-formed module system.

3.6 Module Systems: Semantics

A description of a module system defines both a modular environment **Menv** and a main module. We will first define the semantics of module expressions, assuming the existence of a modular environment **Menv**. The semantics of a module system will be defined at the end of this section.

Module Expressions

A module expression $\mathcal{E}$ denotes a transition module. To be able to resolve references to other modules and to evaluate guards, the meaning of module expressions is relative to a module environment **Menv** and variable environment **Venv**. In the following we assume that these are given.

Direct descriptions The semantics of module expressions is defined inductively. As the base case we have the expression that describes a module directly; in this case

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

Reference If the expression is a reference to another module, that is $\mathcal{E} = id(A)$, then the expression is well-defined if the module environment **Menv** assigns a parameterized module **M** to id, and A is type consistent with **M**'s parameters. Then:

$$[\![id(A)]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}([\![A]\!]_{\mathbf{Venv}})$$

Definitions and Conventions Before we define the semantics of the other operations we will introduce some definitions and conventions. To ensure that the result of composing two modules is again a transition module, we have to impose some conditions on their interfaces, in particular that they do not make inconsistent assumptions about their environments. We also require that transitions that will be synchronized in the composition have the same fairness conditions.

Definition 7 Compatible Modules. Two modules are compatible if:

1. Their interfaces $I_1 = \langle V_1, T_1 \rangle$ and $I_2 = \langle V_2, T_2 \rangle$ are compatible, that is, an output variable of one module is not a shared or output variable of the other module.
2. Their exported transitions $\mathcal{T}_{exp,1}$ and $\mathcal{T}_{exp,2}$ have compatible fairness conditions. That is, for $\tau_1 \in \mathcal{T}_{exp,1}$ and $\tau_2 \in \mathcal{T}_{exp,2}$ with the same label, we require that $\tau_1 \in \mathcal{J}_1 \leftrightarrow \tau_2 \in \mathcal{J}_2$ and $\tau_1 \in \mathcal{C}_1 \leftrightarrow \tau_2 \in \mathcal{C}_2$.

In the definition of the operators we frequently will have to rename part or all of the variables. In the definitions we will use the following convention. Given an expression $\mathcal{E}(v_1, \ldots, v_n)$ and a variable renaming function $\alpha : \mathcal{V} \mapsto \mathcal{V}$, we denote by $\alpha(\mathcal{E}(v_1, \ldots, v_n)$ the expression $\mathcal{E}(\alpha(v_1), \ldots, \alpha(v_2))$. We assume that for every $v \in \mathcal{V}$ if α maps v into $\hat{v}$, then it also maps v' into $\hat{v}'$. We will occasionally write $\alpha(\mathcal{T})$ to represent the set of transitions such that all variables and primed variables in the transition relation are renamed according to α.

Case distinction Let $\mathcal{E}_1 \ldots \mathcal{E}_n$ be well-defined module expressions denoting the modules

$$[\![\mathcal{E}_1]\!]_{\mathbf{Menv},\mathbf{Venv}}, \ldots, [\![\mathcal{E}_n]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}_1, \ldots, \mathbf{M}_n$$

and $g_1, \ldots, g_n$ be first-order formulas. The expression $g_1 : \mathcal{E}_1 \ldots g_n : \mathcal{E}_n$ is well-defined if

- $\mathbf{M}_1 \ldots \mathbf{M}_n$ have identical interfaces, and
- the free variables of $g_1 \ldots g_n$ do not appear in the input, output, shared or private variables of $\mathbf{M_1} \ldots \mathbf{M_n}$, and
- for every variable environment **Venv** there exists exactly one i, $1 \leq i \leq n$ such that $[\![g_i]\!]_{\mathbf{Venv}}$ is true.

If well-defined, the module expression $g_1 : \mathcal{E}_1 \ldots g_n : \mathcal{E}_n$ denotes the module

$$[\![g_1 : \mathcal{E}_1 \ldots g_n : \mathcal{E}_n]\!]_{\mathbf{Menv},\mathbf{Venv}} = \begin{cases} \mathbf{M}_1 & \text{if } [\![g_1]\!]_{\mathbf{Venv}} \text{ is true} \\ \ldots & \\ \mathbf{M}_n & \text{if } [\![g_n]\!]_{\mathbf{Venv}} \text{ is true} \end{cases}$$

Parallel composition Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two well-defined module expressions denoting

$$[\![\mathcal{E}_1]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}_1 = \langle\langle V_1, T_1\rangle, \langle V_{p,1}, \Theta_1, \mathcal{T}_1, \lambda_1, \mathcal{J}_1, \mathcal{C}_1\rangle\rangle$$
$$[\![\mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}} = \mathbf{M}_2 = \langle\langle V_2, T_2\rangle, \langle V_{p,2}, \Theta_2, \mathcal{T}_2, \lambda_2, \mathcal{J}_2, \mathcal{C}_2\rangle\rangle$$

Then $[\![\mathcal{E}_1 \mid\mid \mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}}$ is well-defined if the interfaces $\langle V_1, T_1\rangle$ and $\langle V_2, T_2\rangle$ are compatible. If well-defined, the expression $[\![\mathcal{E}_1 \mid\mid \mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}}$ denotes

$$[\![\mathcal{E}_1 \mid\mid \mathcal{E}_2]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

where

- *Interface Variables:* $V = V_1 \cup V_2$. The partitioning of V into input, output, and shared variables is shown in Figure 3, where **X** denotes combinations that are not allowed.

		$v_2 \in$		
		$V_{i,2}$	$V_{o,2}$	$V_{s,2}$
$v_1 \in$	$V_{i,1}$	V_i	V_o	V_s
	$V_{o,1}$	V_o	**X**	**X**
	$V_{s,1}$	V_s	**X**	V_s

Fig. 3. Combination of interface variables.

- *Exported transitions:* $T = T_1 \cup T_2$
- *Private variables:* Since we do not require $V_{p,1}$ and $V_{p,2}$ to be disjoint, we have to rename private variables to ensure that private variables of different modules are not identified with each other. To do so we let V_p be the disjoint union of $V_{p,1}$ and $V_{p,2}$ and define α_1 to be the mapping that maps every variable from $V_{p,1}$ into the corresponding variable of V_p, and maps all other variables to themselves; α_2 is defined similarly. We assume that $V_p \cap V = \emptyset$. So we have
$$V_p = V_{p,1} \dot{\cup} V_{p,2} = \alpha_1(V_{p,1}) \cup \alpha_2(V_{p,2})$$
- *Initial Condition:* Let α_i be the renaming functions defined before. The initial condition of the composition is the conjunction of the two initial conditions after appropriately renaming the private variables:
$$\Theta = \alpha_1(\Theta_1) \wedge \alpha_2(\Theta_2)$$
- *Transitions:* The new set of transitions is given by
$$\mathcal{T} = \mathcal{T}_{1,p} \cup \mathcal{T}_{2,p} \cup \mathcal{T}_{syn}$$
where $\mathcal{T}_{1,p}$, or $\mathbf{M}_1$'s private transitions, are the transitions from $\mathbf{M_1}$ that do not synchronize with transitions of module $\mathbf{M}_2$, and similarly for $\mathcal{T}_{2,p}$, and

$\mathcal{T}_{syn}$ contains the result of synchronizing those transitions in $\mathbf{M}_1$ and $\mathbf{M}_2$ whose labels appear in both interfaces. Variables in the transition relations are renamed according to the renaming functions α_i as before. For internal transitions, a conjunct is added to the transition relation stating that the transition does not modify the private variables originating from the other module. Formally, for $(i,j) = (1,2), (2,1)$:

$$\mathcal{T}_{i,p} = \left\{ \tau \;\middle|\; \exists \tau^* \in \mathcal{T}_i \,.\, \exists id \in \mathcal{T}_{id} \,.\, ((\tau^*, id) \in \lambda_i \wedge id \notin id_{syn}) \wedge \rho_\tau = \rho^{i,j}_{\tau^*} \right\}$$

where $\rho^{i,j}_{\tau^*}$ is the new transition relation, taking into account the preservation of the private variables of the other module, that is, $\rho^{i,j}_{\tau^*} = \alpha_i(\rho_{\tau^*}) \wedge \alpha_j(\bigwedge_{v \in V_{j,p}} v' = v)$, and where id_{syn} is the set of labels that are exported by both modules, that is

$$id_{syn} = T_1 \cap T_2$$

The set of synchronized transitions is described by

$$\mathcal{T}_{syn} = \left\{ \tau \;\middle|\; \begin{array}{c} \exists id \in id_{syn}, \tau_1 \in \mathcal{T}_1, \tau_2 \in \mathcal{T}_2 \,. \\ (\tau_1, id) \in \lambda_1 \wedge (\tau_2, id) \in \lambda_2 \wedge \rho_\tau = (\alpha_1(\rho_{\tau_1}) \wedge \alpha_2(\rho_{\tau_2})) \end{array} \right\}$$

Note that if a transition τ has a label that synchronizes and a label that does not synchronize, the composed module will contain the synchronized transition as well as the unsynchronized version.

– *Labeling function:* A synchronized transition has the same label as its constituent transitions, that is, for $id \in syn$

$$\lambda_{syn} = \left\{ (\tau, id) \;\middle|\; \exists \tau_1, \tau_2 \,.\, \left[\begin{array}{c} (\tau_1, id) \in \lambda_1 \wedge (\tau_2, id) \in \lambda_2 \\ \wedge \\ \rho_\tau = (\alpha_1(\rho_{\tau_1}) \wedge \alpha_2(\rho_{\tau_2})) \end{array} \right] \right\}$$

and unsynchronized transitions keep the same label, that is, for $id \notin syn$

$$\lambda_{un} = \left\{ (\tau, id) \;\middle|\; \exists \tau^* \,.\, \left[\begin{array}{c} ((\tau^*, id) \in \lambda_1 \wedge \rho_\tau = \rho^{1,2}_{\tau^*}) \\ \vee \\ ((\tau^*, id) \in \lambda_2 \wedge \rho_\tau = \rho^{2,1}_{\tau^*}) \end{array} \right] \right\}$$

Finally,

$$\lambda = \lambda_{syn} \cup \lambda_{un}$$

– *Fairness conditions:* Since we are assuming that transitions can synchronize only if their fairness conditions are the same, we can take the union of the two sets, accounting for the renaming of the transition relations:

$$\begin{aligned} \mathcal{J} &= \alpha_1(\mathcal{J}_1) \cup \alpha_2(\mathcal{J}_2) \\ \mathcal{C} &= \alpha_1(\mathcal{C}_1) \cup \alpha_2(\mathcal{C}_2) \end{aligned}$$

Hiding Let $\mathcal{E}$ be a well-defined module expression denoting

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle I = \langle V, T\rangle, B = \langle V_p, \ldots\rangle\rangle$$

and X a set of variables. Then

$$[\![\mathbf{Hide}(X, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V - X, T\rangle, \langle V_p \cup X, \ldots\rangle\rangle$$

If X is a set of transition labels then

$$[\![\mathbf{Hide}(X, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T - X\rangle, B\rangle$$

Renaming Let $\mathcal{E}$ be a well-defined module expression denoting

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

and $\beta : \mathcal{V} \mapsto \mathcal{V}$ a function that maps variables into variables. α is a renaming on the private variables that ensures that V_p and V are still disjoint after the renaming. If for some interface variable $v \in V$ and private variable $w \in V_p$, $\beta(v) = w$, then $\alpha(w) = z$, where z is a new variable, not present in the interface or in the private variables. Then

$$[\![\mathbf{Rename}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle\beta(\alpha(V)), T\rangle, \langle V_p, \beta(\alpha(\Theta)), \beta(\alpha(\mathcal{T})), \lambda, \beta(\alpha(\mathcal{J})), \beta(\alpha(\mathcal{C}))\rangle\rangle$$

If $\beta : \mathcal{T}_{id} \mapsto \mathcal{T}_{id}$ is a function that maps transition labels into transition labels, then

$$[\![\mathbf{Rename}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, \beta(T)\rangle, \langle V_p, \Theta, \mathcal{T}, \beta(\lambda), \mathcal{J}, \mathcal{C}\rangle\rangle$$

where $(\tau, id) \in \beta(\lambda)$ iff $\exists id^* \,.\, id = \beta(id) \wedge (\tau, id^*) \in \lambda$.

Augmentation Let $\mathcal{E}$ be a well-defined module expression denoting

$$[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$$

and β a partial function mapping variables into expressions over output variables. Again, α is a renaming of the private variables that keeps V and V_p disjoint.

$$[\![\mathbf{Augment}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V \cup dom(\beta), T\rangle, \langle\alpha(V_p), \Theta^*, \mathcal{T}^*, \lambda, \mathcal{J}^*, \mathcal{C}^*\rangle\rangle$$

where the variables in $dom(\beta)$ are added to the output variables. A constraint on the new variables is added to the initial condition:

$$\Theta^* = \alpha(\Theta) \wedge \bigwedge_{v \in dom(\beta)} v = \beta(v)$$

and all transition relations are augmented to update of the newly added variables, that is

$$\mathcal{T}^* = \left\{ \tau^* \;\middle|\; \exists \tau \in \mathcal{T} \,.\, \rho_{\tau^*} = \left(\alpha(\rho_\tau) \wedge \bigwedge_{v \in dom(\beta)} v' = \beta(v') \right) \right\}$$

$\mathcal{J}^*$ and $\mathcal{C}^*$ are defined analogously.

Restriction Let $\mathcal{E}$ be a module expression denoting $[\![\mathcal{E}]\!]_{\mathbf{Menv},\mathbf{Venv}} = \langle\langle V, T\rangle, \langle V_p, \Theta, \mathcal{T}, \lambda, \mathcal{J}, \mathcal{C}\rangle\rangle$, V_n a set of fresh variables, and β a partial function, mapping input variables into expressions over variables in V_n. Then

$$[\![\mathbf{Restrict}(\beta, \mathcal{E})]\!]_{\mathbf{Menv},\mathbf{Venv}} = \\ \langle\langle (V - dom(\beta) \cup V_n, T\rangle, \langle \alpha(V_p), \beta(\alpha(\Theta)), \beta(\alpha(\mathcal{T})), \lambda, \beta(\alpha(\mathcal{J})), \beta(\alpha(\mathcal{C}))\rangle\rangle$$

where the variables in V_n are added to the input variables. β is applied to the initial condition and all transition relations. As before, α denotes the renaming of the private variables necessary to keep V_p and V disjoint.

Module Systems

A module system is described by a list of equations of the form $id(P_i) = \mathcal{E}_i$ defining the modular environment, followed by an expression $\mathcal{E}_{\text{main}}$ defining the *main* module. The modular environment **Menv** is defined as follows,

$$\mathbf{Menv} = \text{lfp}\left(\lambda\, \mathbf{Menv}^* \,.\, \left\{ \begin{array}{l} id_1 \mapsto \lambda X \,.\, [\![\mathcal{E}_1]\!]_{\mathbf{Menv}^*, \mathbf{Venv}[P_1 \backslash X]} \\ id_2 \mapsto \lambda X \,.\, [\![\mathcal{E}_2]\!]_{\mathbf{Menv}^*, \mathbf{Venv}[P_2 \backslash X]} \\ \ldots \end{array} \right.\right)$$

where lfp denotes the least fixpoint. The main module $\mathbf{M}_{\text{main}}$ is the interpretation of $\mathcal{E}_{\text{main}}$ in this environment:

$$\mathbf{M}_{\text{main}} = [\![\mathcal{E}_{\text{main}}]\!]_{\mathbf{Menv},\mathbf{Venv}}$$

In the remainder we assume that a given module system is well-defined, that is, the environment has a unique least fixpoint.

3.7 Example: Arbiter

Continuing the arbiter example, we now describe the full hierarchical arbiter. The `Arbiter` module is composed from an `ArbiterTree` and a module named `Top` that gives and takes grants. `ArbiterTree` is a tree of `ArbiterNode`s that were defined in Figure 2. Both `Arbiter` and `ArbiterTree` are parameterized by their height h.

An `ArbiterTree` of height h communicates with 2^h clients, who can each request access by setting a *request* bit. One client at a time will be given access to the resource, and the `Arbiter` informs the client about its granted access by setting the client's grant bit. The leafs of the tree are defined by an expression over `ArbiterNode` (`++` denotes bit-vector concatenation):

```
Leafnode = Hide(grL, grR,
               Augment(grants = grL ++ grR,
                  Restrict(reqL = requests[0], reqR = requests[1],
                     ArbiterNode))
```

The **Restrict** operation instantiates its input variables `reqL` and `reqR` with the actual request bits of the clients, and the `Augment` operation combines the two output variables `grL` and `grR` into a single variable `grants`. After the augmentation, the output variables `grL` and `grR` can be hidden. Thus the interface of `LeafNode` is

V_i:	`gr`: **bool** `requests`: **bitvector**[**2**]
V_o:	`req`: **bool** `grants`: **bitvector**[**2**]

The parameterized module `ArbiterTree` is described by the module expression

```
ArbiterTree(h)=

    h = 1: LeafNode

    h > 1: Hide(grantsL, grantsR, grL, grR, reqL, reqR,
             Augment(grants = grantsL ++ grantsR,
                (Restrict(requests = requests [0 : 2^(h-1) − 1],
                   Rename(gr = grL, req = reqL, grants = grantsL,
                      ArbiterTree(h-1)))
                ||
                ArbiterNode
                ||
                Restrict(requests = requests [2^(h-1) : 2^h − 1],
                   Rename(gr = grR, req = reqR, grants = grantsR,
                      ArbiterTree(h-1))))))
```

Each instance has the interface

V_i:	`gr`: **bool** `requests`: **bitvector**[$0..2^h - 1$]
V_o:	`req`: **bool** `grants`: **bitvector**[$0..2^h - 1$]

For any given h, the parameterized module `ArbiterTree` describes a tree of height h. The module expression is illustrated by Figure 4, which shows the three modules that are composed and their input and output variables. Note that the **Augment** and **Restrict** operations are necessary to obtain identical interfaces for the cases $h = 1$ and $h > 1$.

We complete our description of a hierarchical arbiter by defining the `Arbiter` module, the main module of the system. An `ArbiterTree` of height h guarantees mutual exclusion among the 2^h clients, but the tree will only give a grant to

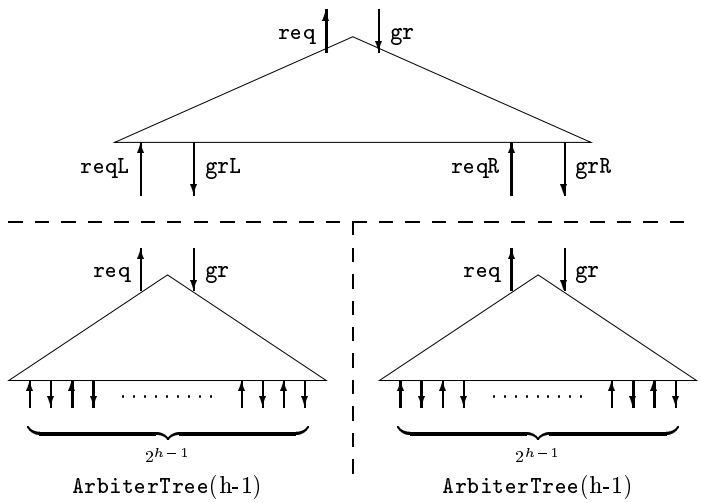


Fig. 4. Composition of `ArbiterNode` and a left and right subtree.

```
Module Top:

out gr : bool where gr=false
external in req :bool

Transition Grant Just:
    enable !gr /\ req
    assign gr:=true

Transition Retract Just:
    enable gr /\ !req
    assign gr:=false

EndModule
```

Fig. 5. Module `Top`.

some client if it has received the grant from its parent entity. This parent entity is represented by the module `Top`, shown in Figure 5.

`Top`'s only actions are to award a grant when one is requested and retract a grant when one is released.

The main module is described as an instance `Arbiter`(h) of the parameterized `Arbiter` module, which is defined as follows:

$$\texttt{Arbiter(h)} = \quad \textbf{Hide}(\texttt{req, gr, (ArbiterTree}(h) \parallel \texttt{Top}))$$

and has interface

V_i:	requests: **bitvector**$[2^h]$
V_o:	grants: **bitvector**$[2^h]$

4 Deductive Verification

In the previous sections we have developed a formalism for modular system descriptions. In this section we now move on to the verification of such systems. We begin with an introduction to available formalisms for the *global* verification of transition systems in Section 4.1. Next, we extend the notion of (global) program validity of temporal formulas to *modular* validity. For systems with non-recursive descriptions we give a proof rule in Section 4.2. While this is a feasible approach to establish the modular validity, we are interested in methods that make use of the structure given by module descriptions. In Section 4.3 we discuss how modular properties can be *inherited* by other modules, and in Section 4.4 we define module abstraction. Finally, in Section 4.5, we discuss the verification of recursively described modules.

4.1 Verification of Transition Systems

The classical deductive framework for the verification of fair transition systems is based on *verification rules*, which reduce temporal properties of systems to first-order or simpler temporal premises [MP95b].

For a past formula φ,

1. $\mathcal{S} \models \Theta \rightarrow \varphi$
2. $\mathcal{S} \models \{\varphi\}\, \mathcal{T}_S\, \{\varphi\}$

$$\mathcal{S} \models \Box\varphi$$

Fig. 6. Invariance rule INV.

Figure 6 presents the *invariance rule*, INV, which can be used to establish the $\mathcal{S}$-validity of formulas of the form $\Box p$, where p is a past formula. Here $\{\varphi\}\, \mathcal{T}_S\, \{\varphi\}$ stands for $\Box(\rho_\tau \wedge \varphi \rightarrow \varphi')$ for all transitions $\tau \in \mathcal{T}_S$. An invariant φ may not be inductive (that is, it may not be preserved by all transitions), in which case it can be necessary to find a stronger, inductive invariant that implies φ, and prove it first. An alternative approach is to first prove a set of

simpler invariants $p_1, ..., p_k$ and then use them to establish the more complicated invariant φ.

Graphical formalisms can facilitate the task of guiding and understanding a deductive proof. *Verification diagrams* [MP94, BMS95] provide a graphical representation of the verification conditions needed to establish a particular temporal formula over a transition system. In this paper we will use generalized verification diagrams [BMS95, BMS96] to prove response properties.

Generalized Verification Diagrams A verification diagram is a graph, with nodes labeled by assertions and propositions and edges labeled by sets of transitions, that represents a proof that a transition system $\mathcal{S}$ satisfies a temporal property φ. A subset of the nodes is marked as initial. First-order verification conditions associated with the diagram prove that the diagram faithfully represents all computations of $\mathcal{S}$. The edge labeling is used to express the fairness properties of the transitions relevant to the property.

Some of the first-order verification conditions generated by the diagram are as follows (for a full description, see [BMS95]):

- *Initiation*: At least one initial node satisfies the initial condition of $\mathcal{S}$.
- *Consecution*: Any τ-successor of a state that satisfies the assertion of a node n, must satisfy the assertion of some successor node of n.
- *Fairness*: If an edge is labeled with a transition, that transition is guaranteed to be enabled.

To show that the diagram satisfies φ can be checked algorithmically, by viewing the diagram as an automaton (considering its propositional labeling only) and checking that its language is included in the language of the formula. Multiple diagrams can be combined such that the intersection of their (propositional) languages is included in the language of the formula [BMS96].

4.2 Verification of Modular Properties

The goal of modular verification is to reduce the task of verifying a system as a whole to the verification of modules. In this section we will define *modular validity* and we will describe a *proof rule* to establish modular properties.

We use the notion of associated transition systems, introduced in Section 3.3, to define the modular validity of a temporal property:

Definition 8 Modular Validity. We say that a property φ is *modularly valid*, or $\mathbf{M}$-valid for a module $\mathbf{M}$, denoted by

$$\mathbf{M} \models \varphi,$$

if φ is valid over the transition system $\mathcal{S}_{\mathbf{M}}$ associated with $\mathbf{M}$.

The set of computations of the associated transition system includes any computation of a system that contains the module. A modular property is therefore valid over any system that contains the module.

Definition 9 Module descriptions in normal form. A module description is in *normal form* if it is either a direct description or a case distinction where all subexpressions are direct descriptions.

Figure 7 presents a proof rule that reduces modular validity to a set of system validities, based on a case distinction on the guards. The rule requires the module description to be in normal form. Note that any non-recursive description can be transformed into normal form, by first expanding the references and then reducing module expressions to direct descriptions.

For a module $\mathbf{M}$, described in normal form
$$g_1 : M_1 \ldots g_n : M_2$$
and a temporal formula φ,

$$\frac{\mathcal{S}_{[\![M_i]\!]} \models g_i \rightarrow \varphi \qquad \text{for } i = 1 \ldots n}{\mathbf{M} \models \varphi}$$

Fig. 7. Modular validity rule MOD.

4.3 Property Inheritance

Rule MOD of Figure 7 allows us to prove the modular validity of properties. The obvious limitation of the rule lies in the requirement that the module description be in normal form: transforming the description into normal form means that all structural information is lost. The *inheritance* proof rules shown in Figure 8, by contrast, make explicit use of this structure. Property inheritance allows us to use properties that were previously proven to be valid over other modules as lemmas in a modular proof.

Example: In the `Arbiter` example, assume we have shown that `ArbiterNode` establishes mutual exclusion between its two clients:

$$\texttt{ArbiterNode} \models \Box\,\neg(\texttt{grL} \wedge \texttt{grR})$$

`LeafNode` is described in terms of `ArbiterNode`. It *inherits* the corresponding property

$$\texttt{LeafNode} \models \Box\,\neg(\texttt{grants[0]} \wedge \texttt{grants[1]})$$

which is in turn inherited by `ArbiterTree`(1):

$$\texttt{ArbiterTree}(1) \models \Box\,\neg(\texttt{grants[0]} \wedge \texttt{grants[1]})$$

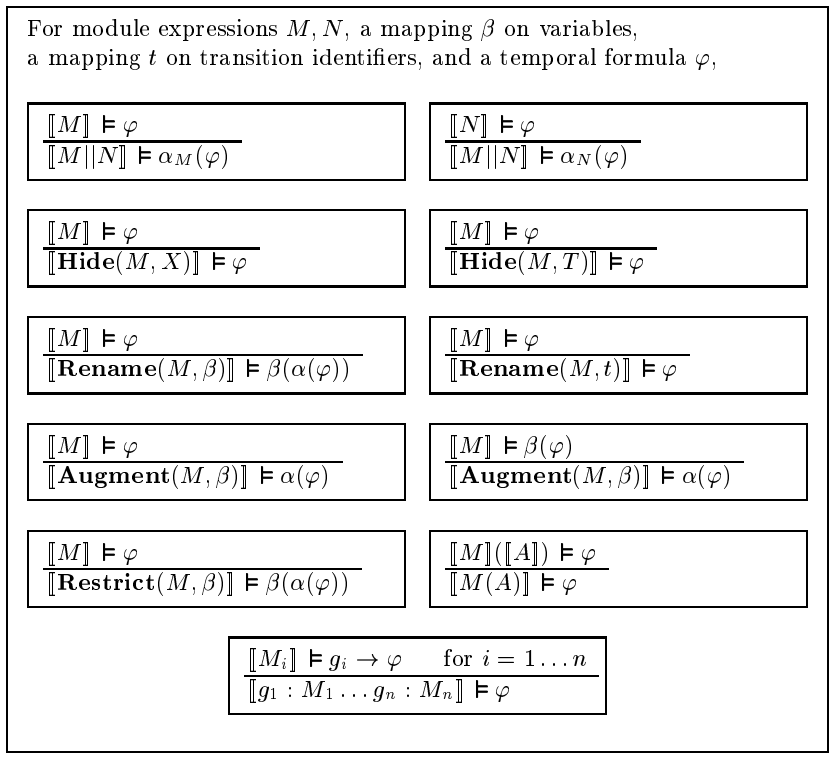


Fig. 8. Property inheritance for various operators.

The inheritance rules for the different module operators in Figure 8 can be justified by showing that a refinement mapping [AL88] exists between the transition systems associated with the modules given in the premise and those in the conclusion. We consider refinement mappings that are induced by a substitution relation:

Definition 10 Refinement. Let S^A and S^C be two transition systems, called the *abstract* and *concrete* transition system, respectively, and $\alpha : V^A \mapsto E(V^C)$ a substitution relation mapping variables from S^A to expressions over variables in S^C. The transition system S^C is an α-refinement of S^A, denoted $S^C \sqsubseteq_\alpha S^A$, if for every computation σ^C of S^C there exists a computation σ^A of S^A such that $\sigma^C = \alpha(\sigma^A)$, where α is extended to a mapping on computations in the obvious way.

The proof rule in Figure 9 states that if S^C is an α-refinement of S^A, properties of S^A are inherited by S^C.

Justification: Assume $S^A \models \varphi$, and $S^C \sqsubseteq_\alpha S^A$. Let σ^C be a computation of S^C. By the definition of refinement, there exists a computation σ^A of S^A

For two transition systems $\mathcal{S}_1$, $\mathcal{S}_2$ and a temporal formula φ,

IH1. $\mathcal{S}^C \sqsubseteq_\alpha S^A$
IH2. $\mathcal{S}^A \models \varphi$

$\mathcal{S}^C \models \alpha(\varphi)$

Fig. 9. General inheritance rule G-INH.

such that $\sigma^C = \alpha(\sigma^A)$. Because $S^A \models \varphi$ we have in particular $\sigma^A \models \varphi$, and thus $\alpha(\sigma^A) \models \alpha(\varphi)$ as required.

Figure 10 presents a proof rule to establish a refinement relation between two transition systems. The rule assumes the existence of a surjective transition mapping $\gamma : \mathcal{T}^C \mapsto \mathcal{T}^A$ that maps each concrete transition to a corresponding abstract transition with the same or weaker fairness condition.

For two transition systems S^C, S^A and a surjective function $\gamma : \mathcal{T}^C \mapsto \mathcal{T}^A$, such that $\tau^c \in \mathcal{J}^C$ implies $\gamma(\tau^c) \notin \mathcal{C}^A$, and $\tau^c \in \mathcal{T}^C - (\mathcal{J}^C \cup \mathcal{C}^C)$ implies $\gamma(\tau^c) \notin \mathcal{J}^A \cup \mathcal{C}^A$,

R1. $\Theta^C \rightarrow \alpha(\Theta^A)$
R2. $\rho_{\tau^C} \rightarrow \alpha(\rho_{\gamma(\tau^C)})$ for every $\tau^C \in \mathcal{T}^C$
R3. $\alpha(En_{\gamma(\tau^C)}) \rightarrow En_{\tau^C}$ for every $\tau^C \in \mathcal{T}^C$ with $\gamma(\tau^C) \in \mathcal{J}^A \cup \mathcal{C}^A$

$S^C \sqsubseteq_\alpha S^A$

Fig. 10. Basic refinement rule B-REF.

Justification: Assume that $\sigma^C = s_o, s_1, \ldots$ is a computation of S^C. We have to show that the premises ensure there exists a corresponding computation $\sigma^A = \alpha(s_0), \alpha(s_1), \ldots$ of the abstract system S^A. By R1 and $s_0 \models \Theta^C$, we have $\alpha(s_0) \models \Theta^A$. For every two consecutive states s_i, s_{i+1} in σ^C, the transition relation of some concrete transition τ_c must be satisfied; by R2, the transition relation of the corresponding abstract transition $\gamma(\tau^C)$ is satisfied for states $\alpha(s_i), \alpha(s_{i+1})$. It remains to show that σ^A is fair. Since γ is onto, there exists for every fair transition τ^A a transition τ^C with equal or stronger fairness, and thus by R2 and R3, τ^A's fairness conditions can only be violated if τ^C's fairness conditions are violated.

Clearly, refinement under a transition mapping γ, denoted by $\sqsubseteq^\gamma$, is a stronger property than refinement alone. However, it suffices for our purposes, and results in simpler verification conditions than, for example, the more gen-

eral proof rule presented in [KMP94], where proving refinement is reduced to the proof of a number of temporal properties.

4.4 Module Abstraction

It is often the case that some components of a module are irrelevant to the validity of a property to be proven. Module abstraction allows us to ignore some or all of the details of those components in the proof, thus simplifying the proof.

The idea is that in a module expression, subexpressions can be replaced by other expressions denoting simpler modules, provided those modules *modularly simulate* the original module, that is, the new module can simulate the original module in any expression.

Definition 11 Modular simulation. A module $[\![M^A]\!]$ *simulates* a module $[\![M^C]\!]$, denoted by $[\![M^C]\!] \sqsubseteq [\![M^A]\!]$, if for all modular expressions $\mathcal{E}(M)$,

$$S_{[\![\mathcal{E}(M^C)]\!]} \sqsubseteq S_{[\![\mathcal{E}(M^A)]\!]}$$

A proof rule to establish modular simulation between two modules is shown in Figure 11.

Justification: Consider two modules $[\![M^C]\!]$ and $[\![M^A]\!]$ with identical interface I. Assume a surjective transition mapping $\gamma : \mathcal{T}^C \to \mathcal{T}^A$ between the sets of transitions that fulfills the condition S1 - S3 (which are identical to the premises R1-R3 in rule B-REF, with α being the identity) and that is consistent with the transition labeling, expressed by premises S4 and S5 respectively: each exported label of a concrete transition τ is also a label of $\gamma(\tau)$, and if a concrete transition τ has an internal label, then so does $\gamma(\tau)$.

For two modules $\mathbf{M}^C, \mathbf{M}^A$ with a common interface,
and a surjective function $\gamma : \mathcal{T}^C \mapsto \mathcal{T}^A$,
such that $\tau^c \in \mathcal{J}^C$ implies $\gamma(\tau^c) \notin \mathcal{C}^A$,
and $\tau^c \in \mathcal{T}^C - (\mathcal{J}^C \cup \mathcal{C}^C)$ implies $\gamma(\tau^c) \notin \mathcal{J}^A \cup \mathcal{C}^A$,

S1. $\Theta^C \to \Theta^A$

S2. $\rho_{\tau^C} \to \rho_{\gamma(\tau^C)}$ for every $\tau^C \in \mathcal{T}^C$

S3. $En_{\gamma(\tau^C)} \to En_{\tau^C}$ for every $\tau^C \in \mathcal{T}^C$ with $\gamma(\tau^C) \in \mathcal{J}^A \cup \mathcal{C}^A$

S4. $\forall l \in T \,.\, \lambda^C(\tau^C, l) \to \lambda^A(\gamma(\tau^C), l)$ for every $\tau^C \in \mathcal{T}^C$

S5. $\left(\begin{array}{c} \exists l^C \in T_{id} - T \,.\, \lambda^C(\tau^A, l^C) \\ \to \\ \exists l^A \in T_{id} - T \,.\, \lambda^A(\gamma(\tau^C), l^A) \end{array}\right)$ for every $\tau^C \in \mathcal{T}^C$

$\mathbf{M}^C \sqsubseteq \mathbf{M}^A$

Fig. 11. Modular simulation rule M-SIM.

We show by induction on expressions that $S_{[\![\mathcal{E}(M^C)]\!]} \sqsubseteq S_{[\![\mathcal{E}(M^A)]\!]}$. For the base case we have to show $S_{[\![M^C]\!]} \sqsubseteq S_{[\![M^A]\!]}$. As the modules $[\![M_A]\!]$ and $[\![M_C]\!]$ have identical interfaces, we can extend γ to a transition mapping γ' on the associated transition systems as follows:

$$\gamma'(\tau) = \begin{cases} \tau \text{ if } \tau = \tau_{env} \\ \tau' \text{ if } \rho_\tau = \rho_{\tau_0} \wedge \bigwedge_{v \in V_i} v = v' \text{ and } \rho_{\tau'} = \rho_{\gamma(\tau_0)} \wedge \bigwedge_{v \in V_i} v = v' \\ \quad \text{for some transition } \tau_0 \in \mathcal{T}_{[\![M^C]\!]}. \\ \tau' \text{ if } \rho_\tau = \rho_{\tau_0} \text{ and } \rho_{\tau'} = \rho_{\gamma(\tau_0)} \\ \quad \text{for some transition } \tau_0 \in \mathcal{T}_{[\![M^C]\!]}. \end{cases}$$

For the inductive step, for each of the operations we can show that given $[\![M^C]\!] \sqsubseteq^\gamma [\![M^A]\!]$, there is a transition mapping γ', such that $[\![\mathcal{E}(M^C)]\!] \sqsubseteq^{\gamma'} [\![\mathcal{E}(M^A)]\!]$. Hence, $[\![M^C]\!] \sqsubseteq^\gamma [\![M^A]\!]$ implies that there is a transition mapping γ'', such that $S_{[\![\mathcal{E}(M^C)]\!]} \sqsubseteq^{\gamma''} S_{[\![\mathcal{E}(M^A)]\!]}$.

The proof rule M-INH in Figure 12, a specialization of the general proof inheritance rule shown in Figure 9, allows us to replace modules in an expression by simpler modules that simulate them.

For modular expressions M, N, and $\mathcal{E}(M)$,

M1. $[\![\mathcal{E}(N)]\!] \models \varphi$
M2. $[\![M]\!] \sqsubseteq [\![N]\!]$

$[\![\mathcal{E}(M)]\!] \models \varphi$

Fig. 12. Modular inheritance proof rule M-INH.

Interface Abstraction For each class of modules with the same interface there is a largest element with respect to the simulation preorder, called the *interface abstraction*, which can be generated automatically.

Definition 12 Interface Abstraction. Let $\mathbf{M} = \langle I, B \rangle$ be a module with interface $I = \langle V, T \rangle$. The *interface abstraction* of $\mathbf{M}$ is the module $\mathbf{A}_\mathbf{M} = \langle I, B^* \rangle$ where $B^* = \langle V_p^* = \emptyset, \Theta^* = true, \mathcal{T}^* = \{\tau_a\}, \lambda^* = \{\tau_a\} \times (T \cup \{l_{proxy}\}), \mathcal{J}^* = \emptyset, \mathcal{C}^* = \emptyset \rangle$

The interface abstraction relies solely on information given by the interface. Using the transition mapping

$$\gamma(\tau) = \begin{cases} \tau_a & \text{if } \exists l \in T_M \,.\, \lambda_M(\tau, l) \\ \tau^* \text{ with } \rho_{\tau^*} = \rho_{\tau_a} \wedge \bigwedge_{v \in V_i} v = v' & \text{otherwise} \end{cases}$$

it is easy to show that $\mathbf{M} \sqsubseteq \mathbf{A}_M$. The transition τ_a can simulate any transition in B. The labeling function covers all exported transitions, and the *proxy* label, which is not exported, ensures that in any composition there is a non-synchronizing transition.

4.5 Induction for Recursive Descriptions

A natural way to prove properties over a recursively defined module is by induction on the parameter value. Let $\prec$ be a well-founded order over the domain $\mathcal{D}$ of the parameters of a module $\mathbf{M}$. The principle of well-founded induction is formulated as follows:

> To show that $\mathbf{M}(X) \models \varphi(X)$ for all $X \in \mathcal{D}$, it suffices to show that for an arbitrary value $X \in \mathcal{D}$
>
> $$\mathbf{M}(A) \models \varphi(A) \text{ for all } A \prec X \text{ (IH)}$$
> $$\text{implies}$$
> $$\mathbf{M}(X) \models \varphi(X)$$
>
> The antecedent is called the inductive hypothesis, and the consequent is called the conclusion of the *inductive step*.

For unknown parameter values the transition system associated with a recursively described module cannot be computed directly. Module abstraction, e.g., interface abstraction, can be used to derive an abstraction with a non-recursive description.

5 Verification of the Arbiter

The two properties we want to prove about the arbiter system are *mutual exclusion:* no two clients can hold the grant simultaneously, expressed by

$$mux(h): \ \Box(\forall i, j : [0..2^h - 1] \,.\, (\texttt{grants}[i] \wedge \texttt{grants}[j]) \rightarrow i = j)$$

and *eventual access:* any client who requests a grant will eventually get a grant, expressed by

$$acc(h): \ \Box(\forall i : [0..2^h - 1] \,.\, \texttt{requests}[i] \rightarrow \Diamond\, \texttt{grant}[i])$$

5.1 Mutual exclusion

The `Arbiter` system was formally specified in Section 3.7, as a composition of `ArbiterTree` and `Top`. By the property inheritance rules, to prove

$$\texttt{Arbiter}(h) \models mux(h)$$

it is sufficient to prove

$$\texttt{ArbiterTree}(h) \models mux(h)$$

The recursive description of `ArbiterTree` suggests a proof by induction. For the base case we have to show

$$\texttt{ArbiterTree}(1) \models mux(1)$$

which, using the definition of `ArbiterTree` for $h = 1$, the definition of `LeafNode` and the property inheritance rules can be reduced to the proof of

$$\texttt{ArbiterNode} \models \Box\neg(\texttt{grL} \wedge \texttt{grR})$$

This property is easily established by applying the invariance rule to the associated transition system of `ArbiterNode`.

By the induction principle, to prove for $h > 1$

$$\texttt{ArbiterTree}(h) \models mux(h)$$

we may make use of the inductive hypothesis, for any $h^* \leq h$, in particular $h^* = h - 1$:

$$\texttt{ArbiterTree}(h-1) \models mux(h-1)$$

and thus, by the property inheritance rules, we inherit

$$\texttt{ArbiterTree}(h) \models \Box(\forall i, j : [0..2^{(h-1)} - 1] \,.\, (\texttt{grants}[i] \wedge \texttt{grants}[j]) \rightarrow i = j)$$

$$\texttt{ArbiterTree}(h) \models \Box(\forall i, j : [2^{(h-1)}..2^h - 1] \,.\, (\texttt{grants}[i] \wedge \texttt{grants}[j]) \rightarrow i = j)$$

for the left and right subtree respectively. The two properties express that each subtree establishes mutual exclusion among its set of clients. Unfortunately, they do not establish mutual exclusion for the tree itself: they do not prohibit the case in which both subtrees simultaneously have given out a grant. To rule out this case, we establish two additional properties. The first property states that no client holds a grant unless the tree holds a grant. This property only holds of the `ArbiterTree` if we assume that its environment does not retract a grant before the `ArbiterTree` releases the grant. Thus we formulate this as an assumption-guarantee property:

$$\texttt{ArbiterTree}(h) \models \begin{array}{l} \text{Assumption:} \\ \Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \\ \text{Guarantee:} \\ \Box\neg\texttt{gr} \rightarrow (\forall i : [0..2^h - 1] \,.\, \neg\texttt{grants}[i]) \end{array} \tag{1}$$

The second property states that only one of the two subtrees can hold the grant, expressed by

$$\texttt{ArbiterTree}(h) \models \Box\neg(\texttt{grL} \wedge \texttt{grR})$$

The latter property is inherited directly from the same property proven earlier for the `ArbiterNode`. From the first property, by the property inheritance rules, we inherit

$$\texttt{ArbiterTree}(h) \models \left(\begin{array}{c} \Box(\ominus(\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{grL}) \\ \rightarrow \\ \Box\neg\texttt{grL} \rightarrow (\forall i : [0..2^{h-1} - 1] \,.\, \neg\texttt{grants}[i]) \end{array}\right)$$

$$\texttt{ArbiterTree}(h) \models \left(\begin{array}{c} \Box(\ominus(\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{grR}) \\ \rightarrow \\ \Box\,\neg\texttt{grR} \rightarrow (\forall i : [2^{h-1}..2^h - 1]\,.\,\neg\texttt{grants}[i]) \end{array} \right)$$

The assumptions are discharged by proving

$$\texttt{ArbiterNode} \models \Box(\ominus(\texttt{grL} \wedge \texttt{reqL})) \rightarrow \texttt{grL} \tag{2}$$

and

$$\texttt{ArbiterNode} \models \Box(\ominus(\texttt{grR} \wedge \texttt{reqR})) \rightarrow \texttt{grR} \tag{3}$$

using the invariance rule. These properties are inherited directly by ArbiterTree(h).

It remains to prove (1). This property is again established by induction. The case $h = 1$ is proved using the invariance rule. For the case $h > 1$ we need, in addition to (2) and (3), the auxiliary property that a client of a node can have the grant only if the node owns the grant and has not released it yet, expressed by

$$\texttt{ArbiterNode} \models \begin{array}{l} \text{Assumption:} \\ \Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \\ \text{Guarantee:} \\ \Box((\texttt{grL} \vee \texttt{grR}) \rightarrow (\texttt{gr} \wedge \texttt{req})) \end{array}$$

This property is readily established using the invariance rule, and the assumptions are again discharged using (2) and (3).

5.2 Eventual Access

To show accessibility for all clients of the arbiter system, we expect we have to make some assumptions on the environment, for example, that clients will eventually release a grant. However, rather than trying to identify these assumptions up front, we choose to discover them in the course of the proof, and we will add them to the property as appropriate. As it is more convenient to do the proof at the level of the ArbiterTree rather than for the arbiter system, assumptions about the parent of the ArbiterTree (called the *server*) are added as well. These will be discharged at the end by the Top module.

To show

$$\texttt{Arbiter}(h) \models acc(h)$$

it suffices to show

$$\texttt{ArbiterTree}(h) \models acc(h) \tag{4}$$

A proof by induction yields as the base case

$$\texttt{ArbiterTree}(1) \models acc(1)$$

which, by the property inheritance rules can be reduced to

$$\texttt{ArbiterNode} \models \Box(\texttt{reqL} \rightarrow \Diamond\, \texttt{grL}) \tag{5}$$

and

$$\texttt{ArbiterNode} \models \Box(\texttt{reqR} \rightarrow \Diamond\, \texttt{grR}) \tag{6}$$

Figure 13 shows a generalized verification diagram to prove (5). It represents the desired flow of the module that establishes the property. The initiation conditions (all initial states are covered by the diagram) and the fairness conditions (the assertions on nodes with outgoing labeled edges imply the enabling condition of the corresponding transition) are readily established. However, consecution (every τ-successor of an assertion is covered) does not hold, for example it fails for node n_1 for the environment transition. Since `gr` and `reqL` are input variables of `ArbiterNode`, the environment transition may modify them arbitrarily. However, if we assume that the parent (server) does not retract a grant before it is released, expressed by

$$\Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \quad (\textit{server}) \tag{7}$$

and that the (right) client does not retract a request before it receives a grant,

$$\Box(\ominus(\neg\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{reqR}) \quad (\textit{client}) \tag{8}$$

the consecution condition holds for n_1. For the consecution condition of n_2 we need to assume that the right client does not request a grant before the previous one is retracted,

$$\Box(\ominus(\texttt{grR} \wedge \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}) \quad (\textit{client}) \tag{9}$$

and for the consecution of node n_4 we assume that the parent does not give a grant unless it is requested.

$$\Box(\ominus(\neg\texttt{gr} \wedge \neg\texttt{req}) \rightarrow \neg\texttt{gr}) \quad (\textit{server}) \tag{10}$$

Additional assumptions are necessary to ensure progress. The fairness of `ReleaseRight`, `RequestGrant` and `GrantLeft` ensure progress from nodes n_2, n_4 and n_6, however no progress is guaranteed from nodes n_1, n_3 and n_5 (note that no progress is required from n_0 or n_7, because in n_0 the antecedent of our property is false, while in n_7 the goal is true). Progress from n_1 requires the (right) client to eventually release the grant:

$$\Box(\texttt{grL} \rightarrow \Diamond(\neg\texttt{reqL} \vee \neg\texttt{grL})) \quad (\textit{client}) \tag{11}$$

To guarantee progress from n_3 and n_5 we have to assume that the server will eventually retract the grant when it is released,

$$\Box(\texttt{req} \rightarrow \Diamond(\texttt{gr} \vee \neg\texttt{req})) \quad (\textit{server}) \tag{12}$$

and that the server will eventually give a grant when one is requested,

$$\Box(\neg\texttt{req} \rightarrow \Diamond(\neg\texttt{gr} \vee \texttt{req})) \quad (\textit{server}) \tag{13}$$

The proof of (6) is similar; it generates the same assumptions for the server and the symmetrical asssumptions for the clients.

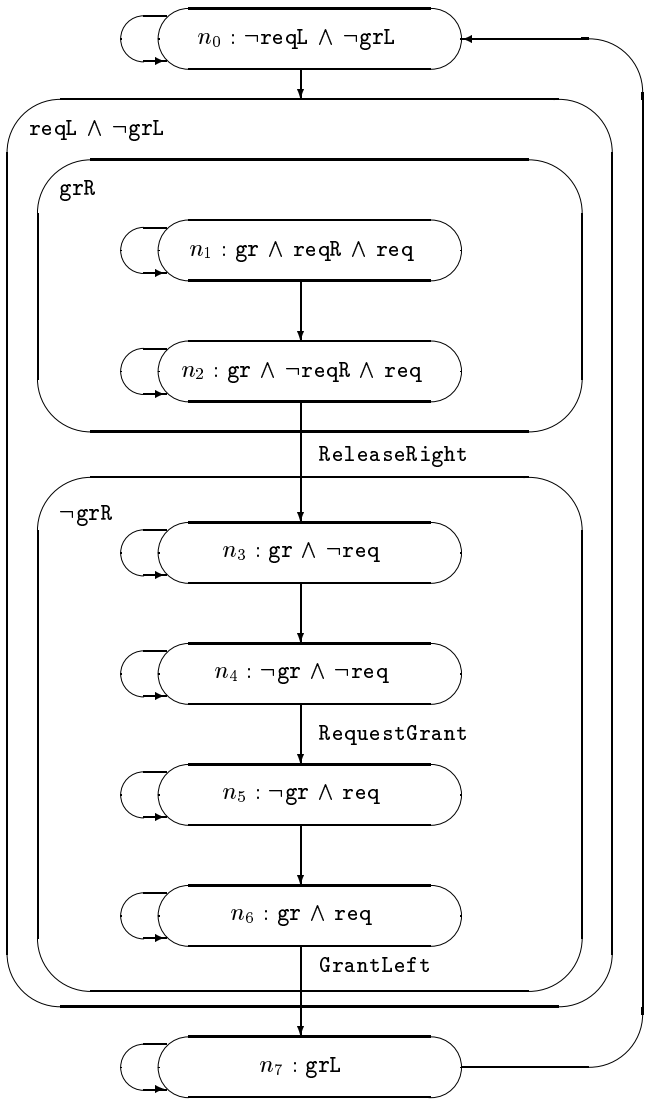


Fig. 13. Verification diagram for property (5).

Generalizing these assumptions about the `ArbiterNode` clients to the clients of an `ArbiterTree` of height h we get,

$$
\begin{array}{l}
\mathit{clients}(h): \\
\quad \forall i : [0..2^h - 1]\ .\ \Box(\ominus(\neg \mathtt{grants}[i] \wedge \mathtt{requests}[i]) \rightarrow \mathtt{requests}[i]); \\
\quad \forall i : [0..2^h - 1]\ .\ \Box(\ominus(\mathtt{grants}[i] \wedge \neg \mathtt{requests}[i]) \rightarrow \neg \mathtt{requests}[i]); \\
\quad \forall i : [0..2^h - 1]\ .\ \Box(\mathtt{grants}[i] \rightarrow \Diamond(\neg \mathtt{requests}[i] \vee \neg \mathtt{grants}[i]))
\end{array}
$$

We now weaken our accessibility property to include the assumptions:

$$\mathtt{ArbiterTree}(h) \models \mathit{server} \wedge \mathit{clients}(h) \rightarrow \mathit{acc}(h) \tag{14}$$

where *server* stands for the conjunction of assumptions made about the parent:

$$
\mathit{server}: \begin{array}{l}
\Box(\ominus(\mathtt{gr} \wedge \mathtt{req}) \rightarrow \mathtt{gr}) \wedge \\
\Box(\ominus(\neg \mathtt{gr} \wedge \neg \mathtt{req}) \rightarrow \neg \mathtt{gr}) \wedge \\
\Box(\neg \mathtt{req} \rightarrow \Diamond(\neg \mathtt{gr} \vee \mathtt{req})) \wedge \\
\Box(\mathtt{req} \rightarrow \Diamond(\mathtt{gr} \vee \neg \mathtt{req}))
\end{array}
$$

To prove the induction step, we make use of the inductive hypothesis for $h^* = h - 1$,

$$\mathtt{ArbiterTree}(h-1) \models \mathit{server} \wedge \mathit{clients}(h-1) \rightarrow \mathit{acc}(h-1)$$

which is inherited by $\mathtt{ArbiterTree}(h)$ as

$$\mathtt{ArbiterTree}(h) \models \mathit{server}_L \wedge \mathit{clients}_L(h) \rightarrow \mathit{acc}_L(h)$$

and

$$\mathtt{ArbiterTree}(h) \models \mathit{server}_R \wedge \mathit{clients}_R(h) \rightarrow \mathit{acc}_R(h)$$

where acc_L and acc_R stand for accessibility for the clients $0 \ldots 2^{(h-1)} - 1$ and $2^{h-1} \ldots 2^h - 1$, respectively,

$$\mathit{acc}_L(h) : \forall i : [0..2^{h-1} - 1]\ .\ \Box(\mathtt{requests}[i] \rightarrow \Diamond\, \mathtt{grants}[i])$$

$$\mathit{acc}_R(h) : \forall i : [2^{h-1}..2^h - 1]\ .\ \Box(\mathtt{requests}[i] \rightarrow \Diamond\, \mathtt{grants}[i])$$

Similarly, $\mathit{clients}_L(h)$ and $\mathit{clients}_R$ refer to the assumptions made about the clients $0 \ldots 2^{(h-1)} - 1$ and $2^{h-1} \ldots 2^h - 1$, respectively; server_L stands for the server assumptions made by the left subtree:

$$
\mathit{server}_L : \mathtt{ArbiterTree}(h) \models \begin{array}{l}
\Box(\ominus(\mathtt{grL} \wedge \mathtt{reqL}) \rightarrow \mathtt{grL}) \wedge \\
\Box(\ominus(\neg \mathtt{grL} \wedge \neg \mathtt{reqL}) \rightarrow \neg \mathtt{grL}) \wedge \\
\Box(\neg \mathtt{reqL} \rightarrow \Diamond(\neg \mathtt{grL} \vee \mathtt{reqL})) \wedge \\
\Box(\mathtt{reqL} \rightarrow \Diamond(\mathtt{grL} \vee \neg \mathtt{reqL}))
\end{array}
$$

that is, `req` and `gr` are replaced by `reqL` and `grL`. Similarly, server_R stands for the server assumptions made by the right subtree. It is easy to see that

$$\mathtt{ArbiterTree}(h) \models \mathit{clients}(h) \rightarrow \mathit{clients}_L(h) \wedge \mathit{clients}_R(h)$$

and

$$\texttt{ArbiterTree}(h) \models acc_L(h) \wedge acc_R(h) \rightarrow acc(h)$$

Thus it remains to discharge $server_L$ and $server_R$. For $server_L$ we show

$$\texttt{ArbiterTree}(h) \models \Box(\ominus(\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{grL}) \tag{15}$$

$$\texttt{ArbiterTree}(h) \models \Box(\ominus(\neg\texttt{grL} \wedge \neg\texttt{reqL}) \rightarrow \neg\texttt{grL}) \tag{16}$$

$$\texttt{ArbiterTree}(h) \models server \rightarrow \Box(\neg\texttt{reqL} \rightarrow \Diamond(\neg\texttt{grL} \vee \texttt{reqL})) \tag{17}$$

$$\texttt{ArbiterTree}(h) \models server \wedge clients(h) \rightarrow \Box(\texttt{reqL} \rightarrow \Diamond\,\texttt{grL}) \tag{18}$$

Property (15) is identical to (2), which was established before. We show (16) by applying INV to the corresponding `ArbiterNode` property. Property (17) can be reduced to

$$\texttt{ArbiterNode} \models \Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr}) \rightarrow \Box(\neg\texttt{reqL} \rightarrow \Diamond(\neg\texttt{grL} \vee \texttt{reqL})) \tag{19}$$

which is proven by the generalized verification diagram shown in Figure 14. The (server) assumption $\Box(\ominus(\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{gr})$ is necessary to ensure that in node n_0 `gr` is preserved by the environment transition.

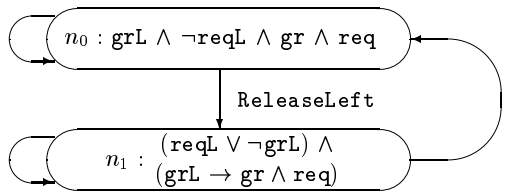


Fig. 14. Verification diagram for property (19).

To show property (18) we make use of property (5), which was proven under the assumptions (7)-(13). The server assumptions, (7), (12), and (13) are discharged immediately by *server* in the antecedent. Thus it remains to show

$$\texttt{ArbiterTree}(h) \models server \wedge clients(h) \rightarrow \Box(\ominus(\neg\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{reqR}) \tag{20}$$

$$\texttt{ArbiterTree}(h) \models server \wedge clients(h) \rightarrow \Box(\ominus(\texttt{grR} \wedge \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}) \tag{21}$$

$$\texttt{ArbiterTree}(h) \models server \wedge clients(h) \rightarrow \Box(\texttt{grL} \rightarrow \Diamond(\neg\texttt{reqL} \vee \neg\texttt{grL})) \tag{22}$$

Property (20) is shown by case analysis. For $h = 1$ the consequent is directly implied by $clients(1)$. For the case $h > 1$ we prove

$$\texttt{ArbiterNode} \models \Box(\ominus(\neg\texttt{gr} \wedge \texttt{req}) \rightarrow \texttt{req}) \tag{23}$$

using the invariance rule; the desired property is then inherited from the right child. Property (21) is proven in the same way, by proving

$$\texttt{ArbiterNode} \models \Box(\ominus(\texttt{grR} \wedge \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}) \tag{24}$$

The proof of (22) proceeds in a similar fashion except here we need to use induction, where both the base case ($h = 1$) and the inductive step ($h > 1$) rely on the following `ArbiterNode` property

$$\begin{array}{ll} & \text{Assumption:} \\ & \Box(\ominus(\neg\texttt{grL} \wedge \texttt{reqL}) \rightarrow \texttt{reqL}); \\ & \Box(\ominus(\neg\texttt{grR} \wedge \texttt{reqR}) \rightarrow \texttt{reqR}); \\ & \Box(\ominus(\texttt{grL} \wedge \neg\texttt{reqL}) \rightarrow \neg\texttt{reqL}); \\ \texttt{ArbiterNode} \models & \Box(\ominus(\texttt{grR} \wedge \neg\texttt{reqR}) \rightarrow \neg\texttt{reqR}); \\ & \Box(\texttt{grL} \rightarrow \Diamond(\neg\texttt{reqL} \vee \neg\texttt{grL})); \\ & \Box(\texttt{grR} \rightarrow \Diamond(\neg\texttt{reqR} \vee \neg\texttt{grR})) \\ & \text{Guarantee:} \\ & \Box(\texttt{gr} \rightarrow \Diamond(\neg\texttt{req} \vee \neg\texttt{gr})) \end{array} \tag{25}$$

which is proven by the generalized verification diagram shown in Figure 15. The first four assumptions are necessary to ensure the consecution requirement for nodes n_0, n_3, n_2, and n_5 respectively. The last two assumptions are used to ensure progress from node n_1 and n_4.

In the base case, when $\texttt{ArbiterTree}(1)$ inherits this property, all assumptions are discharged by $clients(1)$. For the case $h > 1$, the first four assumptions are discharged by (23) and (24) and the corresponding properties for the left side, and the last two properties are discharged by the inductive hypothesis inherited from the left and right subtree. This concludes the proof of (14).

We now finish the proof of accessibility for the `Arbiter` system. It is easy to show

$$\texttt{Top} \models server$$

and therefore we can discharge the *server* assumption for the `Arbiter` system, and we have

$$\texttt{Arbiter}(h) \models clients(h) \rightarrow acc(h)$$

Thus, as expected, accessibility for the arbiter system relies on the cooperation of the clients.

6 Conclusions

We have presented a formal framework for the modular description and verification of fair transition systems, and demonstrated its use on an example. We proposed several deductive proof techniques to establish and re-use modular properties.

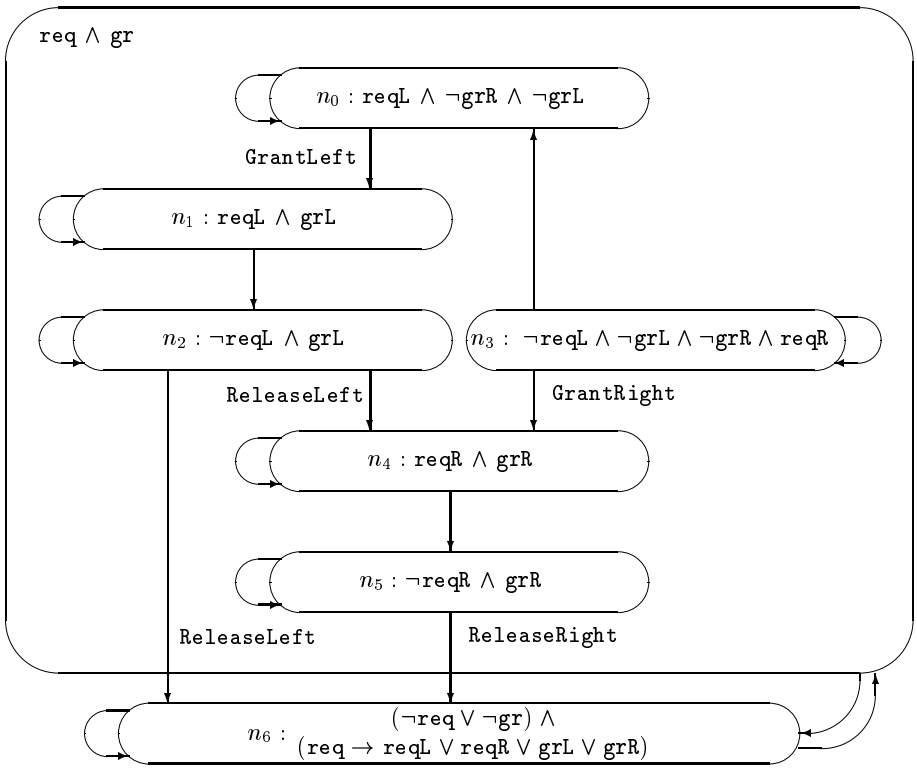


Fig. 15. Verification diagram for property (25).

- A *modular verification rule* to prove properties over modules with non-recursive descriptions.
- A property *inheritance* mechanism that provides an incremental proof method: properties of a module A can be reused in any module B whose description refers to A.
- *Modular abstraction*, which allows us to focus the proof on relevant components.
- The *induction rule*, which makes the methodology applicable to recursive designs.

We illustrated our techniques by verifying an arbiter system. In particular we demonstrated that our framework allows the use of assumption-guarantee reasoning without suffering from its main disadvantage of having to identify sufficiently strong guarantee properties up front. In the verfication of the arbiter system we showed how assumptions are generated naturally in the course of the proof. Diagrams were constructed representing the intended flow of the module, and verification conditions involving input variables were added as assumptions to the property we set out to prove. These assumptions were then carried along

until they either could be discharged by properties proven over other modules, or they could be proven directly over the larger inheriting module.

Although not considered in this paper, the verification methodology can be adapted to other verification techniques, such as *deductive model checking* [SUM96]. It is also straightforward to extend the framework to real-time and hybrid systems, modeled by clocked and hybrid transition systems [MP95a]. In these systems fair, clocked and hybrid parameterized transition modules can be freely combined into one module system. Extra care has to be taken to ensure that time steps synchronize for all parallel modules.

Acknowledgements: We thank Nikolaj Bjørner, Mark Pichora and Tomás Uribe for their careful reading and many helpful suggestions.

References

[AH96] R. Alur and T.A. Henzinger, editors. *Proc. 8^{th} Intl. Conference on Computer Aided Verification*, vol. 1102 of *LNCS*. Springer-Verlag, July 1996.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd IEEE Symp. Logic in Comp. Sci.*, pages 165–175. IEEE Computer Society Press, 1988.

[AL93] M. Abadi and L. Lamport. Conjoining specifications. Technical Report SRC-118, DEC-SRC, December 1993.

[BBC+95] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, November 1995.

[BBC+96] N.S. Bjørner, A. Browne, E.S. Chang, M. Colón, A. Kapur, Z. Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [AH96], pages 415–418.

[BK84] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In *Seminar on Concurrency*, vol. 197 of *LNCS*, pages 35–61. Springer-Verlag, 1984.

[BMS95] A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, vol. 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[BMS96] A. Browne, Z. Manna, and H.B. Sipma. Hierarchical verification using verification diagrams. In 2^{nd} *Asian Computing Science Conf.*, vol. 1179 of *LNCS*, pages 276–286. Springer-Verlag, December 1996.

[BMSU97] N.S. Bjørner, Z. Manna, H.B. Sipma, and T.E. Uribe. Deductive verification of real-time systems using STeP. In *4th Intl. AMAST Workshop on Real-Time Systems*, vol. 1231 of *LNCS*, pages 22–43. Springer-Verlag, May 1997.

[Cha93] E.S. Chang. *Compositional Verification of Reactive and Real-Time Systems.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, 1993. Tech. Report STAN-CS-TR-94-1522.

[Dil88] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, Carnegie-Mellon Univ., 1988. Available as Technical Report CMU-CS-88-119.

[GGS88] S. Garland, J. Guttag, and J. Staunstrup. Verification of vlsi circuits using lp. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 329–345. Elsevier Science Publishers B.V. (North Holland), 1988.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Prog. Lang. Sys.*, 16(3):843–871, May 1994.

[Jon83] C. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[JT95] B. Jonsson and Y.K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. In *TAPSOFT '95*, pages 262–276, 1995.

[KMP94] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.P. de Roever, and G. Rosenberg, editors, *A Decade of Concurrency*, vol. 803 of *LNCS*, pages 273–346. Springer-Verlag, 1994.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, 1987.

[LT89] N.A. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MP94] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J.C. Mitchell, editors, *Proc. International Symposium on Theoretical Aspects of Computer Software*, vol. 789 of *LNCS*, pages 726–765. Springer-Verlag, 1994.

[MP95a] Z. Manna and A. Pnueli. Clocked transition systems. In *Proc. of the Intl. Logic and Software Engineering Workshop*, August 1995. Beijing, China.

[MP95b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, sub-series F: Computer and System Science, pages 123–144. Springer-Verlag, 1985.

[Sei80] C.L. Seitz. Ideas about arbiters. *Lambda*, pages 10–14, 1980.

[Sha93] N. Shankar. A lazy approach to compositional verification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, December 1993.

[Sha98] N. Shankar. Lazy compositional verification. In *this volume*, 1998.

[Sta94] J. Staunstrup. *A Formal Approach to Hardware Design*. Kluwer Academic Publishers, 1994.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In Alur and Henzinger [AH96], pages 208–219.

Compositional Verification of Real-Time Applications

Jozef Hooman

Dept. of Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: hooman@win.tue.nl
http://www.win.tue.nl/win/cs/tt/hooman/

Abstract. To support top-down design of distributed real-time systems, a framework of mixed terms has been incorporated in the verification system PVS. Programs and assertional specifications are treated in a uniform way. We focus on the timed behaviour of parallel composition and hiding, presenting several alternatives for the definition of a denotational semantics. This forms the basis of compositional proof rules for parallel composition and hiding. The formalism is applied to an example of a hybrid system, which also serves to illustrate our ideas on platform-independent programming.

1 Introduction

The aim of this work is to devise a formal framework for the top-down design of distributed real-time systems. Important ingredient of such a framework is a set of compositional proof rules for the programming constructs. That is, it should be possible to reason with the specifications of components without knowing their implementation [dR85,HdR86].

In Sect. 1.1 we introduce the framework of mixed terms. To obtain mechanized support we use the verification system PVS, presented in Sect. 1.2. Section 1.3 addresses applications and the topic of this paper. The structure of the rest of the paper can be found in Sect. 1.4.

1.1 Mixed Terms

Our specifications are based on assertions, i.e., logical formulae that express desired properties of (part of) a system. To be able to formalize intermediate stages during the top-down design of a system, we aim at a framework where specifications and programming constructs can be mixed freely. This is inspired by similar work on untimed systems [Old85,Old91,Zwi89] and related to recent work on timed systems [SO98].

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 276-300, 1998.
Springer-Verlag Berlin Heidelberg 1998

Example 1. As a simple example, consider a top-level specification $TLSpec$, including timing constraints. Suppose we decide to implement this as the parallel composition of two components specified by $SpecComp_1$ and $SpecComp_2$. This refinement step is denoted by $(SpecComp_1 \parallel SpecComp_2) \Rrightarrow TLSpec$. It should be justified by a compositional rule for parallel composition. Often a set of internal events, say $IntEvents$, is introduced to synchronize or communicate between the two components. These internal events can be encapsulated by a hiding construct, denoted by "$-$". By means of a compositional hiding rule we then show $((SpecComp_1 \parallel SpecComp_2) - IntEvents) \Rrightarrow TLSpec$.
Next each component can be developed in isolation. For instance,
$SpecComp_{11} \; ; \; SpecComp_{12} \Rrightarrow SpecComp_1$ and
$SpecComp_{21} \parallel SpecComp_{22} \Rrightarrow SpecComp_2$.
Then, by means of transitivity of $\Rrightarrow$ and so-called monotonicity rules for parallel composition and hiding, this leads to
$(((SpecComp_{11} \; ; \; SpecComp_{12}) \parallel$
$(SpecComp_{21} \parallel SpecComp_{22})) - IntEvents) \Rrightarrow TLSpec$.
Again the specifications can be refined further, leading to a (real-time) program for each software component of the system.

Traditionally, real-time programs are based on constructs such as delays, time-outs, periods, and priorities. Such a program usually depends heavily on the underlying platform, the scheduling policy, and also on the other components of the system (e.g., also the priority of totally unrelated parallel components is important). Consequently, program verification requires many assumptions about the execution platform and compositionality is difficult to achieve.

Our approach aims at a platform-independent program activity, postponing platform considerations as long as possible. This is achieved by extending conventional (untimed) programming languages with so-called timing annotations that only specify the relevant timing constraints [HvR97]. As a separate activity, these platform-independent programs are then scheduled on a particular execution platform.

1.2 Mechanized Proof Support

To obtain mechanized support for our formal framework, we use the verification system PVS (Prototype Verification System) [ORS92,ORSvH95]. Our mixed framework has been formulated in the language of PVS, a typed higher-order logic. Type-checking might generate proof obligations (Type Check Conditions), requiring a proof that expressions indeed have the proper type. In general, properties can be verified by means of the interactive proof checker of PVS.

PVS provides not only convenient support during the specification and verification of applications, but it is equally useful for the development of formal theory. The use of so-called putative theorems, expressing properties that ought to hold, also frequently reveals errors during theoretical studies. For instance, in the work described here, several errors have been detected by the formulation of alternative definitions and attempts to prove equivalence.

1.3 Applications and Topic of this Paper

In previous work, our assertional framework has been used to verify several protocols with PVS, such as part of the ACCESS.bus protocol [Hoo95] and a membership protocol with local clocks and a dynamically changing network [Hoo97]. Another class of applications concerns the design of hybrid systems, containing both discrete and continuous components. For instance, a steam boiler control system has been specified and verified [VH96].

These applications are based on a formalization of the framework in PVS as described in [Hoo94]. There the proof rules for sequential programming constructs have been proved sound, but the rule for parallel composition was stated as an axiom, based on a manual soundness proof. Main topic of the current paper is to formalize this soundness proof and to investigate several alternatives for the formalization of parallel composition. Additionally we investigate the hiding construct and prove soundness of the required monotonicity rules.

We aim at a framework which allows reasoning about local clocks and hybrid systems; it should be possible to deal with both discrete and continuous time. Hence the framework developed here will be parameterized by a time domain and can be instantiated with various notions of time.

Focusing on parallel composition and hiding, we only describe the externally visible behaviour of a component in terms of the events that occur at any point of time. This clarifies the exposition, leaving extensions of the approach to deal with a local state along the lines of [Hoo94] to future work.

1.4 Overview

The basic semantic primitives are defined in Sect. 2. Several equivalent formulations for the semantics of parallel composition can be found in Sect. 3. Moreover, we prove the soundness of a compositional rule for parallel composition. An alternative framework, aiming at the definition of parallel composition by intersection is presented in Section 4. We show that the two approaches are isomorphic. Semantics and proof rules for the hiding construct are given in Sect. 5, again investigating both approaches. To illustrate the use of the framework and our ideas on platform-independent design, we consider in Sect. 6 a simple example of a hybrid system. Section 7 contains concluding remarks.

2 Semantic Primitives

The PVS specification language allows us to structure the framework into a number of parameterized theories. Theory TimePrim contains a few simple notations to express timing properties. The time domain `Time` is a parameter of this theory, which makes it possible to instantiate it for concrete examples with, e.g., a discrete or a continuous notion of time. The semantic framework and the

proof rules presented here are independent of this choice. We only assume given a strict order $<$ and a partial order $\leq$ on this domain, assuming the usual relation between these two, as expressed by the ASSUMING clause below. Importing theory TimePrim with a particular time domain then leads to a so-called Type Check Condition which requires a proof that the particular orders indeed satisfy the formula named leq_less. Of course, we could use only one of the orders as a parameter and define the other in terms of it, but the version presented here allows us to import the theory with, for instance, the real numbers and then make optimal use of the built-in decision procedures of PVS for the reals and the corresponding relations.

We use three dots (...) to indicate that some obvious details are omitted. Observe that TimePrim imports theory Connectives which is presented below.

```
TimePrim[Time : TYPE, < : (strict_order?[Time]),
                     ≤ : (partial_order?[Time])] : THEORY
  BEGIN
  ASSUMING
    leq_less : ASSUMPTION ∀ (t1, t2 : Time) : t1 ≤ t2 ⇔ t1 < t2 ∨ t1 = t2
  ENDASSUMING
  t, t0, t1 : VAR Time

  [t0, t1] : setof[Time] = {t | t0 ≤ t ∧ t ≤ t1}
  [t0, t1) : setof[Time] = {t | t0 ≤ t ∧ t < t1}
  ...
  P : VAR pred[Time]      % pred[Time] = [Time → bool]
  I : VAR setof[Time]     % setof[Time] = [Time → bool]

  P in I : bool = ∃ t : t ∈ I ∧ P(t)

  P during I : bool = ∀ t : t ∈ I ⇒ P(t)

  IMPORTING Connectives[Time]
END TimePrim
```

Theory Connectives is copied from [Sha98]; it lifts the boolean connectives to the domain of predicates over a given type.

```
Connectives[T : TYPE] : THEORY
  BEGIN
  P, Q : VAR pred[T]
  t : VAR T

  ¬(P)(t) : bool = ¬P(t);
  ...
  TRUE(t) : bool = TRUE
  FALSE(t) : bool = FALSE
END Connectives
```

The basic semantic primitives are defined in theory SemPrim. Since theory TimePrim is imported and no more restrictions are imposed on the time domain, we use the same parameters and also copy the assuming clause to be able to prove the generated Type Check Condition. This construction is used in all subsequent theories, except for concrete examples, and is omitted henceforth.

We declare a nonempty type of events and define the notion of an *observation function* which is a function from the time domain to a set of events. The intention is that such a function represents an externally observable timing behaviour of a component by describing the set of events that occur at each point of time. Events can be shared by several components, e.g., for synchronization or to observe the same physical event.

The basic semantic structure is defined by type `CompInfo` which is a record with two fields:

- Field α represents the *alphabet* of the component, that is, the set of events that might occur during the behaviour of the component.
- Field *obs* describes the observable behaviour of the component. Since we allow nondeterministic components, it is represented by a set of observation functions.

Note that components are not restricted to (parts of) computer programs; also physical components can be represented in this way, as will be illustrated by an example of a hybrid system in Sect. 6.

In subsequent sections, additional restrictions will be imposed on the type CompInfo, to represent choices concerning the semantic representation. Further note that the exposition here only concerns the externally observable behaviour and a representation of an internal state is omitted.

SemPrim[Time : TYPE, $<$: (strict_order?[Time]),
$\leq$: (partial_order?[Time])] : THEORY
BEGIN
ASSUMING
leq_less : ASSUMPTION $\forall\ (t_1, t_2 :$ Time$) :\ t_1 \leq t_2 \Leftrightarrow t_1 < t_2 \vee t_1 = t_2$
ENDASSUMING
IMPORTING TimePrim[Time, $<$, $\leq$]

Events : NONEMPTY_TYPE

ObsFuncts : TYPE = [Time $\rightarrow$ setof[Events]]

CompInfo : TYPE = [# α : setof[Events], obs : setof[ObsFuncts] #]

The usual operations on sets (of events) are lifted to observation functions.

o, o_1, o_2, o_3 : VAR ObsFuncts
$t, t_0, t_1, t_2, t_3, t_4$: VAR Time

ci : VAR CompInfo
Eset, Eset0, Eset1, Eset2 : VAR setof[Events]
e : VAR Events

$o_1 \cup o_2$: ObsFuncts $= \lambda\ t :\ o_1(t) \cup o_2(t)$
$o \cap \text{Eset}$: ObsFuncts $= \lambda\ t :\ o(t) \cap \text{Eset}$
$o \setminus \text{Eset}$: ObsFuncts $= \lambda\ t :\ o(t) \setminus \text{Eset}$
$o \subseteq \text{Eset}$: bool $= \forall\ t :\ o(t) \subseteq \text{Eset}$

obs_union_comm : LEMMA $o_1 \cup o_2 = o_2 \cup o_1$
obs_union_assoc : LEMMA $(o_1 \cup o_2) \cup o_3 = o_1 \cup (o_2 \cup o_3)$
intersection_difference : LEMMA $o \cap (\text{Eset1} \setminus \text{Eset2}) = (o \setminus \text{Eset2}) \cap \text{Eset1}$

With the current definitions, $o(t)$(read) expresses that a event read occurs at time t (note that a set of events is represented as a predicate on events). To be able to use the notations from theory TimePrim, one should be able to write $o(\text{read})(t)$ and then, for instance, read **in** $[3, 7)$. This can be achieved by defining a conversion At which makes it possible to interpret o as $\text{At}(o)$ when convenient.

$\text{At}(o)(e)(t)$: bool $= o(t)(e)$

CONVERSION At
END SemPrim

Henceforth, declarations of variables are not repeated, so below o, o_1, t, t_0, etc., are used without declaration.

3 Parallel Composition

The parallel composition of real-time components is studied, where we consider in this section a representation where events outside the alphabet of a process, i.e. events of the environment, are ignored. In Sect. 3.1, a denotational semantics is presented. To increase the confidence in the definition, several equivalent formulations are studied. Section 3.2 contains a general framework to denote assertional specifications. A compositional rule for parallel composition can be found in Sect. 3.3.

3.1 Denotational Semantics of Parallel Composition

Theory Sem defines the type Comps of components by imposing the restriction that any observation function of a component contains only events of its alphabet. So an observation function of a component describes only the behaviour that is relevant for the component itself.

```
Sem[Time : TYPE ...] : THEORY
  BEGIN
  ASSUMING...
  IMPORTING SemPrim[Time, <, ≤]

  ObsInAlpha(ci) : bool = ∀ o : obs(ci)(o) ⇒ o ⊆ α(ci)

  Comps : TYPE = {ci | ObsInAlpha(ci)}

END Sem
```

Parallel composition of two components can be defined by taking the union of the alphabets and the (pointwise) union of the observation functions, with the additional restriction that these observation functions agree (i.e., are equal) on the events that are in both alphabets.

```
SemPar[Time : TYPE ...] : THEORY
  BEGIN
  ASSUMING...
  IMPORTING Sem[Time, <, ≤]

  comp, comp0, comp1, comp2, comp3, comp4 : VAR Comps

  par(comp1, comp2) : Comps =
    (# α := α(comp1) ∪ α(comp2),
       obs := λ o : (∃ o1, o2 : obs(comp1)(o1) ∧ obs(comp2)(o2) ∧ o = o1 ∪ o2 ∧
                      o1 ∩ α(comp1) ∩ α(comp2) = o2 ∩ α(comp1) ∩ α(comp2)) #)
```

Since par(comp1, comp2) should be of type Comps, type-checking leads to a Type Check Condition requiring a proof that the definition indeed satisfies ObsInAlpha. The proof uses the fact that this property already holds for the components.

Moreover, ObsInAlpha has been used to show that an equivalent definition is obtained if $o_1 \cap \alpha(\text{comp1}) \cap \alpha(\text{comp2}) = o_2 \cap \alpha(\text{comp1}) \cap \alpha(\text{comp2})$ is replaced by $o_1 \cap \alpha(\text{comp2}) = o_2 \cap \alpha(\text{comp1})$.

Example 2. Consider a component P_1 which performs a *read* event at time 3 and a $comm(1)$ event between 7 and 9. Component P_2 has the following behaviour; if a $comm(v)$ event occurs at time t, for some value v, then it performs a $write(v+1)$ between $t+5$ and $t+10$. Note that this behaviour is represented by a possibly infinite set of observation functions; for all values of t and v and any time point between $t+5$ and $t+10$ there exists an observation function.

To obtain a behaviour of $\text{par}(P_1, P_2)$ we take the union of a behaviour of P_1 and one of P_2, provided they agree on the joint *comm* event. Hence we can only use observation functions of P_2 that contain a $comm(1)$ event between 7 and 9, and thus have a $write(2)$ event between 12 and 19. Hence the observation functions of $\text{par}(P_1, P_2)$ contain a *read* at 3, a $comm(1)$ event between 7 and 9, and a $write(2)$ between 12 and 19.

To increase the confidence in the definition of parallel composition, an alternative definition is formulated and shown to be equivalent with the previous one. This alternative definition is similar to the trace-based semantics of parallel composition [Hoa85], which turned out to be useful for the formulation of a compositional rule for (untimed) parallel composition [Zwi89]. Basic idea is that the projection of an observation of the parallel construct onto the alphabet of one of the components leads to an observation of this component. Projection is here represented by intersection. Additionally, any observation function of the composition should only contain events of its alphabet. Note that // can be used as an infix operator in PVS.

```
//(comp1, comp2) : Comps =
  (# α := α(comp1) ∪ α(comp2),
     obs := λ o : (∃ o1, o2 : obs(comp1)(o1) ∧ obs(comp2)(o2) ∧
                              o ∩ α(comp1) = o1 ∧ o ∩ α(comp2) = o2 ∧
                              o ⊆ α(comp1) ∪ α(comp2)) #)

AltSemEquiv : THEOREM comp1 // comp2 = par(comp1, comp2)
```

Example 3. Consider again the processes P_1 and P_2 of Example 2. Then any observation function of $P_1//P_2$ should lead to an observation function of P_1 if the intersection (projection) is taken with the events of P_1, viz. *read* and *comm*. Hence any observation function of $P_1//P_2$ should have a *read* event at 3 and a $comm(1)$ event between 7 and 9. Since the projection onto events of P_2, viz. *comm* and *write* should lead to an observation function of P_2, this implies that there is a $write(2)$ between 12 and 19.

As a further test of the definitions, it can be shown that parallel composition boils down to a simple union of observation functions if there are no shared events. Moreover, we show that // is idempotent.

```
NoSharedEvents : LEMMA α(comp1) ∩ α(comp2) = ∅ ⇒
  comp1 // comp2 =
    (# α := α(comp1) ∪ α(comp2),
       obs := λ o : (∃ o1, o2 : obs(comp1)(o1) ∧ obs(comp2)(o2) ∧ o = o1 ∪ o2) #)

  Idempotence : FACT comp // comp = comp

END SemPar
```

3.2 Specifications

Common to the two versions of the semantic framework presented here, is a notion of refinement between components. Component ci1 refines (implements) components ci2, denoted by ci1 $\Rrightarrow$ ci2, if the alphabet of "implementation" ci1 is an extension of that of "specification" ci2 and the observable behaviour of ci1 is included in that of ci2.

SpecsPrim[Time : TYPE ...] : THEORY
BEGIN
ASSUMING...
IMPORTING SemPrim[Time, $<, \leq$]

ci1 $\Rrightarrow$ ci2 : bool $= \alpha(\text{ci2}) \subseteq \alpha(\text{ci1}) \wedge \text{obs}(\text{ci1}) \subseteq \text{obs}(\text{ci2})$

RefRefl : THEOREM ci $\Rrightarrow$ ci
RefTrans : THEOREM (ci0 $\Rrightarrow$ ci2) $\Leftrightarrow$ ($\exists$ ci1 : (ci0 $\Rrightarrow$ ci1) $\wedge$ (ci1 $\Rrightarrow$ ci2))

Instead of a semantic description it is often convenient to specify components using assertions, which are simply predicates over observation functions.

Assertion : TYPE = pred[ObsFuncts]
IMPORTING Connectives[ObsFuncts]
A, A_0, A_1, A_2 : VAR Assertion
Valid(A) : bool = $\forall\, o$: $A(o)$
END SpecsPrim

In general, assertional specifications can have a certain structure with, for instance pre and post conditions or rely/guarantee pairs. To illustrate the basic concepts, here a specification simply consists of an alphabet and a single assertion.

To allow mixed terms, combining specifications and programming constructs in a uniform way, specifications also have type Comps. Since this requires a proof of ObsInAlpha, an observation function satisfying the specification should only contain events of the alphabet.

Specs[Time : TYPE ...] : THEORY
BEGIN
ASSUMING...
IMPORTING Sem[Time, $<, \leq$], SpecsPrim[Time, $<, \leq$]

spec(Eset, A) : Comps = (# α := Eset, obs := $\lambda\, o$: $o \subseteq$ Eset $\wedge\, A(o)$ #)

END Specs

Example 4. Assume given read and write events.

read, write : Events
ReadWriteDiff : AXIOM read $\neq$ write

Observe that component C_1

C_1 : Comps = spec($\{e \mid e = \text{write}\}$, $\lambda\, o$: $o(7)$(write))

specifies observation functions that contain only write events. Component C_2

C_2 : Comps = spec({e | $e = \text{read} \vee e = \text{write}$}, $\lambda\, O$: $O(7)(\text{write})$)

additionally allows arbitrary read events. Lemma ActionFalse below expresses that we obtain an empty set of observation functions is we restrict the alphabet to read events. On the other hand, assertion $\lambda\ o$: $\neg o(7)(\text{write})$ does not impose any additonal restriction.

ActionFalse : LEMMA spec({e | $e = \text{read}$}, $\lambda\ o$: $o(7)(\text{write})$) = spec({e | $e = \text{read}$}, FALSE)

NoActionTrue : LEMMA spec({e | $e = \text{read}$}, $\lambda\ o$: $\neg o(7)(\text{write})$) = spec({e | $e = \text{read}$}, TRUE)

3.3 Compositional Proof Rule for Parallel Composition

Main topic of this chapter is the formulation of a compositional rule for parallel composition. First, however, we prove the soundness of a consequence rule, which allows weakening of assertions.

RulePar[Time : TYPE ...] : THEORY
BEGIN
ASSUMING...
IMPORTING SemPar[Time, $<$, $\leq$], Specs[Time, $<$, $\leq$]

ConsRule : THEOREM Eset0 = Eset $\wedge$ Valid($A_0 \Rightarrow A$) $\Rightarrow$ (spec(Eset0, A_0) $\Rrightarrow$ spec(Eset, A))

The next example shows that the condition Eset0 = Eset of the consequence rule cannot be weakened to Eset $\subseteq$ Eset0.

Example 5. Component C_2 is not a refinement of C_1, because C_1 specifies, by its alphabet, that there are no read events, whereas C_2 allows arbitrary read events.

C2notrefC1 : FACT $\neg(C_2 \Rrightarrow C_1)$

The monotonicity rule is important for the composition of refinement steps.

MonoPar : THEOREM (comp1 $\Rrightarrow$ comp3) $\wedge$ (comp2 $\Rrightarrow$ comp4) $\Rightarrow$ (comp1 // comp2 $\Rrightarrow$ comp3 // comp4)

Next, the aim is to formulate a rule where the parallel composition of specifications of components corresponds to the conjunction of the assertions:

spec(Eset1, A_1) // spec(Eset2, A_2) $\Rrightarrow$ spec(Eset1 $\cup$ Eset2, $A_1 \wedge A_2$). (*)

The following example shows, however, that this is not valid if, e.g., assertion A_1 refers to events of Eset2 that are not in Eset1.

Example 6. Formula (*) would lead to

$$\begin{array}{l}\mathrm{spec}(\{e \mid e = \mathrm{read}\},\ \lambda\, o :\ \neg o(7)(\mathrm{write})) \\ \quad // \ \mathrm{spec}(\{e \mid e = \mathrm{write}\},\ \lambda\, o :\ o(7)(\mathrm{write})) \Rrightarrow \\ \qquad \mathrm{spec}(\{e \mid e = \mathrm{read} \vee e = \mathrm{write}\}, \mathrm{FALSE})\end{array}$$

and hence, using Example 4,

$$\begin{array}{l}\mathrm{spec}(\{e \mid e = \mathrm{read}\}, \mathrm{TRUE}) \\ \quad // \ \mathrm{spec}(\{e \mid e = \mathrm{write}\},\ \lambda\, o :\ o(7)(\mathrm{write})) \Rrightarrow \\ \qquad \mathrm{spec}(\{e \mid e = \mathrm{read} \vee e = \mathrm{write}\}, \mathrm{FALSE})\end{array}$$

To rule out such counter examples, it is required that validity of the assertion of a component only depends on the events of its alphabet. Hence we define $\mathrm{OnlyDepEve}(A, \mathrm{Eset})$ to express that validity of A only depends on the events in Eset. Again, to increase the confidence, we prove the equivalence with an alternative formulation which also turns out to be convenient in proofs.

$$\mathrm{OnlyDepEve}(A, \mathrm{Eset}) :\ \mathrm{bool} \ =\ \forall\, o_1, o_2 : A(o_1) \wedge o_1 \cap \mathrm{Eset} = o_2 \cap \mathrm{Eset} \Rightarrow A(o_2)$$

$$\mathrm{OnlyDepEveEquiv} :\ \textsc{lemma}\ \mathrm{OnlyDepEve}(A, \mathrm{Eset}) \Leftrightarrow (\forall\, o :\ A(o) \Leftrightarrow A(o \cap \mathrm{Eset}))$$

$$\begin{array}{l}\mathrm{ParCompRule} :\ \textsc{theorem}\ \mathrm{OnlyDepEve}(A_1, \mathrm{Eset1}) \wedge \mathrm{OnlyDepEve}(A_2, \mathrm{Eset2}) \Rightarrow \\ \quad \mathrm{spec}(\mathrm{Eset1}, A_1)\ //\ \mathrm{spec}(\mathrm{Eset2}, A_2) \Rrightarrow \mathrm{spec}(\mathrm{Eset1} \cup \mathrm{Eset2}, A_1 \wedge A_2)\end{array}$$

Theorem ParCompRule, expressing soundness of the parallel composition rule, has been proved as follows. Assume that we have $\mathrm{OnlyDepEve}(A_1, \mathrm{Eset1})$ and $\mathrm{OnlyDepEve}(A_2, \mathrm{Eset2})$. Let $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset1}, A_1)\ //\ \mathrm{spec}(\mathrm{Eset2}, A_2))$. Then there exist, e.g., an o_1 such that $o \cap \mathrm{Eset1} = o_1$ and $A_1(o_1)$, and hence we have $A_1(o \cap \mathrm{Eset1})$. By OnlyDepEveEquiv this leads to $A_1(o)$. By symmetry, we obtain $o \in \mathrm{obs}(\mathrm{spec}(\mathrm{Eset1} \cup \mathrm{Eset2}, A_1 \wedge A_2))$.

4 Semantics of Parallel Composition by Intersection

In this section we aim at a framework where the semantics of parallel composition can be formulated as the intersection of the sets of behaviours of the two components. To achieve this, here the semantic representation of a component contains arbitrary events of the environment. The relation with the semantics of the previous section is studied in Sect. 4.1.

First define a type of components, called CompsEnv, where we require that any arbitrary behaviour outside the alphabet is included.

$$\begin{array}{l}\mathrm{ArbOutAlpha}(\mathrm{ci}) :\ \mathrm{bool} \ = \forall\, o_1, o_2 : \\ \quad \mathrm{obs}(\mathrm{ci})(o_1) \wedge o_1 \cap \alpha(\mathrm{ci}) = o_2 \cap \alpha(\mathrm{ci}) \Rightarrow \mathrm{obs}(\mathrm{ci})(o_2)\end{array}$$

$$\mathrm{CompsEnv} :\ \textsc{type} \ =\ \{\mathrm{ci} \mid \mathrm{ArbOutAlpha}(\mathrm{ci})\}$$

Parallel composition is defined by the intersection of the sets of behaviours.

$$\text{ce, ce0, ce1, ce2, ce3} : \text{VAR CompsEnv}$$

$$//(\text{ce1, ce2}) : \text{CompsEnv} = (\#\ \alpha := \alpha(\text{ce1}) \cup \alpha(\text{ce2}),\ \text{obs} := \text{obs(ce1)} \cap \text{obs(ce2)}\ \#)$$

Example 7. Consider again the two components of Example 2: P_1 which does a *read* at time 3 and a $comm(1)$ between 7 and 9, P_2 responds to a $comm(v)$ at time t with a $write(v+1)$ between $t+5$ and $t+10$. Now the observation functions of P_1 contain arbitrary events outside its alphabet (*read* and *comm*), including arbitrary *write* events. Similarly, the observable behaviour of P_2 contains observation functions with arbitrary *read* events. Taking the intersection of these sets of functions, we obtain all observation functions that contain a *read* at 3, a $comm(1)$ event between 7 and 9, and a $write(2)$ between 12 and 19.

In our mixed framework, specifications are also components, and hence the definition of specifications has to be adapted to obtain arbitrary behaviour outside the alphabet.

$$\text{spec(Eset}, A) : \text{CompsEnv} = (\#\ \alpha := \text{Eset},\ \text{obs} := \lambda\ o : (\exists\ o_1 : A(o_1) \wedge o \cap \text{Eset} = o_1 \cap \text{Eset})\ \#)$$

Example 8. Observe that for the components C_1 and C_2 of Example 4, which are now of type CompsEnv, we have

$$\text{C2refC1} : \text{FACT}\ C_2 \Rrightarrow C_1$$

since C_1 allows more arbitrary behaviour (all non-write events) than C_2 (all non-write and non-read events).

As before we have NoActionTrue, i.e. a specification expressing no activity outside the alphabet is equivalent to true. But now also assertions expressing the occurrence of events outside the alphabet are ignored, as expressed by ActionTrue

$$\text{ActionTrue} : \text{LEMMA spec}(\{e \mid e = \text{read}\},\ \lambda\ o :\ o(7)(\text{write})) = \text{spec}(\{e \mid e = \text{read}\}, \text{TRUE})$$

$$\text{NoActionTrue} : \text{LEMMA spec}(\{e \mid e = \text{read}\},\ \lambda\ o :\ \neg o(7)(\text{write})) = \text{spec}(\{e \mid e = \text{read}\}, \text{TRUE})$$

Nice is that in this framework the consequence rule can be strengthened, since the condition about the alphabets can be weakened to a subset relation.

$$\text{ConsRule} : \text{THEOREM Eset} \subseteq \text{Eset0} \wedge \text{Valid}(A_0 \Rightarrow A) \Rightarrow (\text{spec(Eset0}, A_0) \Rrightarrow \text{spec(Eset}, A))$$

The rules for monotonicity and parallel composition remain unchanged. The proof of ParCompRule now proceeds as follows. Again assume that we have OnlyDepEve(A_1, Eset1) and OnlyDepEve(A_2, Eset2). Consider an observation $o \in$ obs(spec(Eset1, A_1) // spec(Eset2, A_2)). Then, e.g., $o \in$ obs(spec(Eset1, A_1)) and hence there exists an o_1 such that $A_1(o_1)$ and $o \cap$ Eset1 $= o_1 \cap$ Eset1. Hence OnlyDepEve leads to $A_1(o)$. By symmetry, $o \in$ obs(spec(Eset1 $\cup$ Eset2, $A_1 \wedge A_2$)).

Finally observe that Example 6 can also be used here to show that the OnlyDepEve conditions are needed for the soundness of the rule.

4.1 Relating the Frameworks

To relate the two approaches and the corresponding definitions of parallel composition, two functions are defined to transform one representation into the other. They are each others inverse.

```
AddEnv(comp) : CompsEnv =
  (# α := α(comp),
     obs := λ o1 : (∃ o2 : obs(comp)(o2) ∧ o1 ∩ α(comp) = o2 ∩ α(comp)) #)

RemEnv(ce) : Comps =
  (# α := α(ce),
     obs := λ o1 : (∃ o2 : obs(ce)(o2) ∧ o1 = o2 ∩ α(ce)) #)

AddRemProp : LEMMA AddEnv(RemEnv(ce)) = ce
RemAddProp : LEMMA RemEnv(AddEnv(comp)) = comp
```

Theorems RemRel and AddRel relate the two versions of parallel composition. Note that AddRel can be proved easily from RemRel and the properties AddRemProp and RemAddProp.

```
RemRel : THEOREM RemEnv(ce1 // ce2) = RemEnv(ce1) // RemEnv(ce2)
AddRel : THEOREM
           AddEnv(comp1 // comp2) = AddEnv(comp1) // AddEnv(comp2)
```

We have not yet addressed commutativity and associativity of parallel composition in the framework of Sect. 3. The reason is that it is much easier to prove these properties first in the framework with an arbitrary environment, where we can simply use the properties of union and intersection from the PVS prelude.

```
ParCommEnv : LEMMA ce1 // ce2 = ce2 // ce1
ParAssocEnv : LEMMA (ce1 // ce2) // ce3 = ce1 // (ce2 // ce3)
```

By RemAddProp and theorem AddRel these results can be tranformed easily to the framework without environment events.

```
ParComm : LEMMA comp1 // comp2 = comp2 // comp1
ParAssoc : LEMMA (comp1 // comp2) // comp3 = comp1 // (comp2 // comp3)
```

5 Hiding of Internal Events

Besides parallel composition, also the possibility to encapsulate internal events is important during top-down design. In Sect. 5.1 a hiding operator is defined in the framework of Sect. 3. The other framework, with an arbitrary enironment, is considered in Sect. 5.2.

5.1 Hiding in the Framework without Environment Events

To hide a set of events Eset from component comp, denoted by "comp - Eset", the elements of Eset are removed from the alphabet and the observation functions.

```
Hiding[Time : TYPE ...] : THEORY
  BEGIN
  ASSUMING...
  IMPORTING RulePar[Time, <, ≤]

  ⇔(comp, Eset) : Comps =
      (# α := α(comp) \ Eset,
         obs := λ o : (∃ o1 : obs(comp)(o1) ∧ o = o1 \ Eset) #)
```

To show that the definition indeed leads to an element of type Comps, type-checking leads to the requirement to prove that it satisfies ObsInAlpha.

We also prove a monotonicity rule and a hiding rule, which requires that the assertion of the specification only depends on the remaining, not hidden, events.

```
  HideMono : THEOREM (comp1 ⇛ comp2) ⇒ (comp1 ⇔ Eset ⇛ comp2 ⇔ Eset)

  HideRule : THEOREM OnlyDepEve(A, Eset \ Eset0) ⇒
                        spec(Eset, A) ⇔ Eset0 ⇛ spec(Eset \ Eset0, A)
END Hiding
```

Example 9. For the components C_1 and C_2 of Example 4 we can prove

```
  C1hideC2 : LEMMA C1 = C2 ⇔ {e | e = read}
```

5.2 Hiding in the Framework with an Arbitrary Environment

In the alternative framework we cannot simply remove internal events from the observation function. On the contrary, hiding is achieved by including any arbitrary behaviour concerning the events to be hidden.

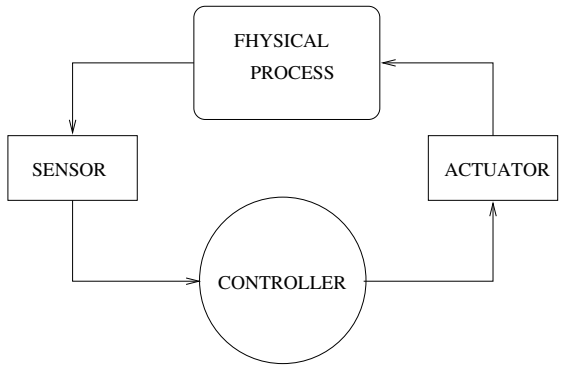
```
⇔(ce, Eset) : CompsEnv =
    (# α := α(ce) \ Eset,
       obs := λ o : (∃ o1 : obs(ce)(o1) ∧ o \ Eset = o1 \ Eset) #)
```

The rules are identical to the previous section. Similar to parallel composition (Sect. 4.1), it is reassuring that the two versions are isomorphic.

```
HideRemRel : LEMMA RemEnv(ce ⇔ Eset) = RemEnv(ce) ⇔ Eset
HideAddRel : LEMMA AddEnv(comp ⇔ Eset) = AddEnv(comp) ⇔ Eset
```

6 Hybrid Systems

As an application of the theory above, we present a simple example of a hybrid system, i.e. a system with discrete and continuous components. As a typical example, we consider in Sect. 6.3 a simple process control application with the following structure.



The general outline of our approach for such systems is described in Sect. 6.2. We will use the framework of Sect. 3 and Sect. 5.1 (i.e., without environment events), to show a few steps towards the design of a discrete controller. Further, the example serves to illustrate recent ideas on platform-independent development of real-time systems [HvR97], as indicated in the next section.

6.1 Platform-Independence

The goal is to postpone platform-dependent design decisions as long as possible. To achieve this, we distinguish two activities:

- A platform-independent programming activity, where a program is developed independent of any particular architecture on which it has to be executed. To achieve this, a notation has been devised to annotate programs with timing specifications. By programming the functional behaviour and only specifying the timing behaviour, we obtain a context and platform independent way of describing algorithms, similar to untimed system design.
- An activity where the program, including timing annotations, is realized on a particular platform. This involves the transformation of the annotated program into scheduling blocks plus timed precedence constraints, and the design of a schedule for the execution platform. Different from the usual compilation phase for untimed programs is that schedulability analysis might indicate that it is not possible to find a schedule and either the program or the platform has to be adapted.

This apprach should be contrasted with the traditional practice where in an early phase of the design internal deadlines, periods, and priorities are chosen. Thus making the design both context dependent (e.g., even depending on priorities of unrelated tasks) and implementation dependent (e.g., depending on the duration of basic statements and the scheduling policy).

6.2 Design of Process Control Systems

To design a real-time computer system which controls physical processes, we propose the following approach.

1. Formulate the requirements specification of the complete system, including continuous components.
2. Formalize the assumptions about the physical processes in the system.
3. Specify the control component, in general using continuous quantities (as in the previous steps).
4. Verify the control algorithm of the previous step, i.e. show that the specifications of 2. and 3. lead to the properties specified in step 1.
5. Step-wise transformation of the continuous control strategy (of step 3) into a specification in terms of a discrete interface. Usually this is done by means of sensors and actuators, assuming formal specifications of these components.
6. Design a program satisfying the discrete specification, obtained in the previous step, of the control component.

6.3 Example Hybrid System

The example presented here is a simplified version of a mine pump example [MJ96]. Consider a mine with a certain engine (e.g., a pump to evacuate water) which should not be operating when a certain amount of gas is present, because that might lead to an explosion. Following the steps above, first the requirements specification of the mine system is formulated.

Requirements Specification As a first specification of the system, we simply require that no explosion should occur at any point of time, where we choose the real numbers as our (continuous) time domain.

ExplEx : THEORY
  BEGIN
  Time : TYPE = real
  NonNegTime : TYPE = $\{t : \text{Time} \mid t \geq 0\}$
  IMPORTING Hiding[Time, $<, \leq$]

  expl : Events

$\mathrm{EMS} : \mathrm{setof[Events]} = \{e \mid e = \mathrm{expl}\}$

$\mathrm{AMS} : \mathrm{Assertion} = \lambda\, o : \forall\, t : \neg o(t)(\mathrm{expl})$

$\mathrm{MineSystem} : \mathrm{Comps} = \mathrm{spec}(\mathrm{EMS}, \mathrm{AMS})$

Specify Physical Environment Next we specify the assumption that an explosion can only occur if the engine is operating and there has been gas for at least, say, CritGasPeriod time units. For simplicity, suppose there is an event *engine* which occurs iff the engine is operating. An alternative is to use start and stop events for the engine, and to define a predicate expressing that the engine is on if the last event was a start event.

$\mathrm{gas}, \mathrm{engine} : \mathrm{Events}$

$\mathrm{ContEventsDiffer} : \textsc{axiom}\ \mathrm{gas} \neq \mathrm{expl} \wedge \mathrm{gas} \neq \mathrm{engine} \wedge \mathrm{expl} \neq \mathrm{engine}$

$\mathrm{EM} : \mathrm{setof[Events]} = \{e \mid e = \mathrm{expl} \vee e = \mathrm{gas} \vee e = \mathrm{engine}\}$

$\mathrm{CritGasPeriod} : \mathrm{NonNegTime}$

$\mathrm{AM} : \mathrm{Assertion} = \lambda\, o :$
$\forall\, t : o(\mathrm{expl})(t) \Rightarrow o(\mathrm{engine})(t) \wedge o(\mathrm{gas})\ \mathbf{during}\ [t \Leftrightarrow \mathrm{CritGasPeriod}, t]$

$\mathrm{Mine} : \mathrm{Comps} = \mathrm{spec}(\mathrm{EM}, \mathrm{AM})$

Specify Control Component The aim is to specify a control strategy Control, in terms of the physical events gas and engine, such that together with specification Mine we obtain the required specification MineSystem. Hence the alphabet is defined by EC.

$\mathrm{EC} : \mathrm{setof[Events]} = \{e \mid e = \mathrm{gas} \vee e = \mathrm{engine}\}$

As a first attempt, we specify that if there is gas at some point in time, then the engine is switched off within, say, StopDelay time units;

$\lambda\, o : \forall\, t : o(t)(\mathrm{gas}) \Rightarrow \neg o(\mathrm{engine})\ \mathbf{in}\ [t, t + \mathrm{StopDelay}]$

Assuming StopDelay $\leq$ CritGasPeriod this indeed leads to MineSystem. But the specification is stronger than necessary and imposes an unrealistic condition on the implementation; gas needs to be observed at any point in time, since a response is needed for any single point in time where gas occurs. Therefore, as a next attempt, we weaken the assertion, requiring that if there is gas during a certain period then the engine should be off after StopDelay.

$\lambda\, o : \forall\, t_1, t_2 : o(\mathrm{gas})\ \mathbf{during}\ [t_1, t_2] \Rightarrow \neg o(\mathrm{engine})\ \mathbf{during}\ [t_1 + \mathrm{StopDelay}, t_2]$

Observe that the assertion is trivially true if $t_2 < t_1 + \mathrm{StopDelay}$, and hence a

response is only needed if there is gas during StopDelay time units. Again we can derive MineSystem, provided StopDelay $\leq$ CritGasPeriod.

However, aiming at platform-independence, we try to avoid internal reaction times, such as StopDelay, and corresponding conditions on them. This leads to the following specification, which only refer to the given constant CritGasPeriod.

$$\text{AC} : \text{Assertion} = \lambda\, o : \\ \forall\, t : o(\text{gas})\ \textbf{during}\ [t, t + \text{CritGasPeriod}] \Rightarrow \neg o(\text{engine})(t + \text{CritGasPeriod})$$

$$\text{Control} : \text{Comps} = \text{spec}(\text{EC}, \text{AC})$$

Verification of the Control Algorithm Main part of the verification is the proof that the assertions of the mine and the control algorithm lead to the assertion of the system, as expressed by lemma MSAssert.

$$\text{MSAssert} : \textsc{lemma}\ \text{Valid}(\text{AM} \wedge \text{AC} \Rightarrow \text{AMS})$$

To apply the parallel composition rule we first prove OnlyDepEve(AM, EM) and OnlyDepEve(AC, EC). Then the rules ParCompRule and ConsRule, together with transitivity of refinement (theorem RefTrans) lead essentially to the required specification, except that the alphabet still contains the events gas and engine.

$$\text{MSIntEve} : \text{setof[Events]} = \{e \mid e = \text{gas} \vee e = \text{engine}\}$$

$$\text{MSPar} : \textsc{theorem}\ \text{Mine} \,//\, \text{Control} \Rrightarrow \text{spec}(\text{EMS} \cup \text{MSIntEve}, \text{AMS})$$

To remove the gas and engine events, we apply monotonicity of hiding (HideMono), transitivity of refinement (RefTrans), and hiding (HideRule).

$$\text{MSRef} : \textsc{theorem}\ (\text{Mine} \,//\, \text{Control}) \Leftrightarrow \text{MSIntEve} \Rrightarrow \text{MineSystem}$$

END ExplEx

6.4 Transformation into a Discrete Interface

To obtain a discrete interface, we assume here that the control program communicates with sensors and actuators by means of shared registers. This communication mechanism is axiomatized in theory RegisterComm, expressing that reading a register yields the last written value.

RegisterComm[Registers : TYPE, Values : NONEMPTY_TYPE, Time : TYPE ...] : THEORY
BEGIN
ASSUMING...
IMPORTING SemPrim[Time, $<$, $\leq$]

reg : VAR Registers
val, val0, val1, val2 : VAR Values

read(reg, val) : Events
write(reg, val) : Events

LastWrite(reg, val, o, t) : bool $= \exists\ t_0 : t_0 \leq t \wedge o(\text{write}(\text{reg}, \text{val}))(t_0) \wedge$
$(\forall\ \text{val1} : \text{val1} \neq \text{val} \Rightarrow \neg o(\text{write}(\text{reg}, \text{val1}))\ \textbf{during}\ (t_0, t])$

NoWrite(reg, o, t) : bool $= \forall\ t_0, \text{val} :\ t_0 \leq t \Rightarrow \neg o(\text{write}(\text{reg}, \text{val}))(t_0)$

ReadWriteAx : AXIOM
$o(\text{read}(\text{reg}, \text{val}))(t) \Rightarrow \text{LastWrite}(\text{reg}, \text{val}, o, t) \vee \text{NoWrite}(\text{reg}, o, t)$

Observe that in case of simultaneous writes a non-deterministic choice is made. Any arbitrary value is allowed if there are no preceding writes. Finally we introduce a convenient abstraction from the values read or written.

read(reg) : Events
write(reg) : Events
AbstrReadAx : AXIOM $o(\text{read}(\text{reg}))(t) \Leftrightarrow (\exists\ \text{val} :\ o(\text{read}(\text{reg}, \text{val}))(t))$
AbstrWriteAx : AXIOM $o(\text{write}(\text{reg}))(t) \Leftrightarrow (\exists\ \text{val} :\ o(\text{write}(\text{reg}, \text{val}))(t))$
END RegisterComm

Gas Sensor Assume given a gas sensor which updates a boolean register GasPresent at least once every WritePeriod time units. Moreover, assume each update reflects the current state of the gas.

ExplReg : THEORY
BEGIN
IMPORTING ExplEx
Registers : NONEMPTY_TYPE
RegValues : TYPE $=$ bool
IMPORTING RegisterComm[Registers, RegValues, Time, $<$, $\leq$]

GasPresent : Registers

EGS : setof[Events] $= \{e \mid e = \text{gas} \vee e = \text{write}(\text{GasPresent}) \vee$
$\exists\ \text{val} :\ e = \text{write}(\text{GasPresent}, \text{val})\}$

WritePeriod : NonNegTime

AGS1 : Assertion $= \lambda\ o :\ \forall\ t :\ o(\text{write}(\text{GasPresent}))\ \textbf{in}\ [t, t + \text{WritePeriod}]$

$$\text{AGS2} : \text{Assertion} = \lambda\, o : \forall\, t, \text{val} : o(\text{write}(\text{GasPresent}, \text{val}))(t) \Rightarrow \text{val} = o(\text{gas})(t)$$

$$\text{AGS} : \text{Assertion} = \text{AGS1} \wedge \text{AGS2}$$

$$\text{GasSensor} : \text{Comps} = \text{spec}(\text{EGS}, \text{AGS})$$

Gas Sensor Control The goal is to specify a component which reads register GasPresent and controls the engine. Hence it has the following alphabet.

$$\text{ESC} : \text{setof[Events]} = \{e \mid e = \text{engine} \vee e = \text{read}(\text{GasPresent}) \vee \exists\, \text{val} : e = \text{read}(\text{GasPresent}, \text{val})\}$$

A possible specification of the control component could express that register GasPresent is read at least once every ReadPeriod time units.

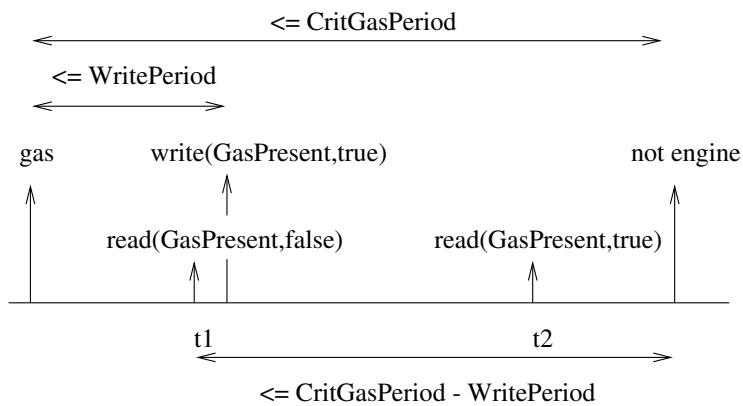
$\lambda\, o : \forall\, t : o(\text{read}(\text{GasPresent}))$ **in** $[t, t + \text{ReadPeriod}]$

When reading a value TRUE from this register, the engine is stopped within SensorControlDelay time units, i.e., given o, for all t,

$o(\text{read}(\text{GasPresent}, \text{TRUE}))(t) \Rightarrow \neg o(\text{engine})$ **in** $[t, t + \text{SensorControlDelay}]$

Then we can prove that together with GasSensor this implies Control, assuming a number of constraints on timing constants ReadPeriod and SensorControlDelay.

Again, however, we prefer to avoid the choice of internal reading periods and assumptions about internal response times. Here it is indeed possible to give a specification without introducing new timing constants. The key idea is that the response deadline should not depend on the time when GasPresent was found to be true, but on the previous read event. Observe that when gas becomes present, it might take at most WritePeriod before GasPresent is changed accordingly, and this might happen just after the previous read. Hence we have to switch the engine off within CritGasPeriod − WritePeriod after the previous read event.



In the specification below, the previous read is not mentioned explicitly, but only the preceding non-read period. To be able to stop the engine in time, the register has to be read at least once every CritGasPeriod − WritePeriod time units.

$$\text{reg}: \text{VAR Registers}$$

$$\text{OnceReadSince}(o, \text{reg}, \text{val})(t_1, t_2): \text{bool} = t_1 < t_2 \wedge o(\text{read}(\text{reg}, \text{val}))(t_2) \wedge \neg o(\text{read}(\text{reg})) \textbf{ during } (t_1, t_2)$$

$$\text{OnceReadSince}(o, \text{reg})(t_1, t_2): \text{bool} = \exists\ \text{val}: \text{OnceReadSince}(o, \text{reg}, \text{val})(t_1, t_2)$$

$$\text{ASC1}: \text{Assertion} = \lambda\ o : \forall\ t : (\exists\ t_1, t_2 : t_1 < t \wedge t_2 \in [t, t + \text{CritGasPeriod} \Leftrightarrow \text{WritePeriod}) \wedge \text{OnceReadSince}(o, \text{GasPresent})(t_1, t_2))$$

We specify that if a value TRUE is read at t_2 and any previous read event occured before t_1, then the engine is switched off before $t_1 + \text{CritGasPeriod} - \text{WritePeriod}$ and it remains off as long as no value FALSE has been read.

$$\text{ASC2}: \text{Assertion} = \lambda\ o : \forall\ t_1, t_2, t_3 : \text{OnceReadSince}(o, \text{GasPresent}, \text{TRUE})(t_1, t_2) \wedge \neg o(\text{read}(\text{GasPresent}, \text{FALSE})) \textbf{ in } (t_2, t_3] \Rightarrow (\exists\ t : t \in [t_2, t_1 + \text{CritGasPeriod} \Leftrightarrow \text{WritePeriod}] \wedge \neg o(\text{engine}) \textbf{ during } [t, t_3])$$

$$\text{ASC}: \text{Assertion} = \text{ASC1} \wedge \text{ASC2}$$

$$\text{SensorControl}: \text{Comps} = \text{spec}(\text{ESC}, \text{ASC})$$

Verification of this Design Step To verify the design step, the most difficult part is to show that the assertions of the gas sensor and the sensor control lead to control assertion AC. This is expresed by lemma GSAssert, assuming WritePeriod $\leq$ CritGasPeriod. After proving OnlyDepEve(AGS, EGS) and OnlyDepEve(ASC, ESC), the parallel composition rule leads to the control specification with the exception of a number of internal events.

$$\text{GSAssert}: \text{LEMMA WritePeriod} \leq \text{CritGasPeriod} \Rightarrow \text{Valid}(\text{AGS} \wedge \text{ASC} \Rightarrow \text{AC})$$

$$\text{GSIntEve}: \text{setof[Events]} = \{e \mid e = \text{write}(\text{GasPresent}) \vee (\exists\ \text{val}: e = \text{write}(\text{GasPresent}, \text{val})) \vee e = \text{read}(\text{GasPresent}) \vee (\exists\ \text{val}: e = \text{read}(\text{GasPresent}, \text{val}))\}$$

$$\text{GSPar}: \text{THEOREM WritePeriod} \leq \text{CritGasPeriod} \Rightarrow (\text{GasSensor} \ //\ \text{SensorControl} \Rrightarrow \text{spec}(\text{EC} \cup \text{GSIntEve}, \text{AC}))$$

By the hiding rule the internal events can be removed and finally we can combine this design step with the previous one, viz. MSRef, leading to TLGSRef.

GSRef : THEOREM WritePeriod $\leq$ CritGasPeriod $\Rightarrow$
((GasSensor // SensorControl $\Leftrightarrow$ GSIntEve $\Rrightarrow$ Control))

TLGSRef : THEOREM WritePeriod $\leq$ CritGasPeriod $\Rightarrow$
((Mine // ((GasSensor // SensorControl) $\Leftrightarrow$ GSIntEve)) $\Leftrightarrow$ MSIntEve $\Rrightarrow$
MineSystem)

END ExplReg

Similarly, we could specify an actuator for the engine and develop a discrete specification of a control component. Here we only show the basic ideas of the last step where a program satisfying this specification is developed.

Platform-Independent Program Design To illustrate the basic ideas of our approach to platform-independent program design, we extend a simple imperative programming language with so-called *timing annotations*, written between brackets "[...]". These annotations contain expressions with special timing variables $m, m_1, m_2, \ldots$ which are used to record and restrict the *execution moment* of statements. The execution moment of a statement is a relevant point in time during the execution of the statement.

Example 10. Consider, as a simple example, the program

$$read(r_1, x)[\, m? \,] \; ; \; \ldots \quad y := f(x) \quad \ldots \; ; \; write(r_2, y)[\, > m + L \; ; < m + U \,].$$

Annotation $m?$ expresses that the execution moment of $read(r_1, x)$ is assigned to m. The annotation of $write(r_2, y)$ expresses that its execution moment should be after $m + L$ and before $m + U$. Hence the timing annotations express the timing relations between relevant events, without further assumptions about the timing of intermediate parts.

In our simplified mine pump example, this leads to the following program.

Let $d = CritGasPeriod - WritePeriod$.

$[\, m_1 := 0 \,]$;

while $true$

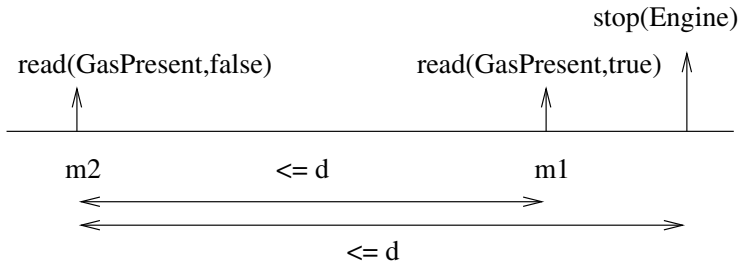
do $[\, m_2 := m_1 \,]$;

$read(GasPresent, b)[\, \leq m_2 + d, ?m_1 \,]$;

if b **then** $stop(Engine)[\, \leq m_2 + d \,]$ **fi**

od

Observe that m_2 records the previous read event (0 if there is no such event). The timing annotations require that a read event occurs within at most d time units after the previous one, and a possible stop action also takes place within this deadline, as the next picture shows.



With timing annotations there is no need to choose a reading period and, correspondingly, to determine an internal response time. Note that such a choice should depend on the platform; a short reading period gives a heavy periodic load, but a long period implies a short remaining response time and hence imposes strong requirements on the timing of the response actions.

In general, correctness of a program with timing annotations does not require any assumption about the execution platform such as the duration of statements, the number of processors, the mapping of processes to processors, and the particular scheduling policy. Extracting a timed precedence graph from the program and scheduling it on a particular platform is considered as a separate activity.

7 Concluding Remarks

We have presented two approaches for a timed semantics of parallel components:

- In the first framework (Sect. 3) an observation function of a component contains only events of its alphabet. The semantics of parallel composition can either be formulated using the union of observations plus some synchronisation constraint, or by requiring that an observation function of the parallel construct can be projected onto behaviours of the components.
- In the second framework (Sect. 4) any arbitrary behaviour outside the alphabet is included in the semantics. Then the semantics of parallel composition is defined using a simple intersection of the sets of behaviours.

A straightforward correspondence has been established between the two semantics approaches, using functions that add or remove the arbitrary events of the environment.

Observe that these approaches also have their consequences for the allowed refinements, although we used the same refinement relation for both approaches (a simple set inclusion, similar to the trace inclusion in e.g. [Old85]). For instance, in the second framework the condition in the consequence rule can be weakened.

Interesting is comparison with the refinement relation defined in [AH97], which requires that if an observation of the implementation is projected onto the alphabet of the specification, an observation of the specification is obtained.

In our framework this could be defined as

$$\mathrm{ci1} \preceq \mathrm{ci2} : \ \mathrm{bool} \ = \alpha(\mathrm{ci2}) \subseteq \alpha(\mathrm{ci1}) \wedge (\forall\, o : \ \mathrm{obs}(\mathrm{ci1})(o) \Rightarrow \mathrm{obs}(\mathrm{ci2})(o \cap \alpha(\mathrm{ci2})))$$

Observe that for reflexivity of $\preceq$ it is required that an observation function contains only events of the afphabet, i.e. property ObsInAlpha should hold.

As mentioned in [AH97], this relation implies $\mathrm{P} \ // \ \mathrm{Q} \preceq \mathrm{P}$. This does not hold in our first framework without environment events, but there we can hide the external events of the second component and get

$$((\mathrm{comp1} \ // \ \mathrm{comp2}) \ - \ (\alpha(\mathrm{comp2}) \setminus \alpha(\mathrm{comp1}))) \Rightarrow \mathrm{comp1}$$

In the second framework, which contains arbitrary environment events, we have

$$\mathrm{ce1} \ // \ \mathrm{ce2} \Rightarrow \mathrm{ce1}$$

since adding a parallel component simply restricts the set of behaviours.

Clearly more work is needed to investigate various notions of refinement in combination with the real-time frameworks developed here and to study their mutual relations. Also relevant is a comparison of extensions to sequential programs with a local state, similar to [Hoo94]. Another topic of future research concerns the specifications which are kept simple here, but might be more structured with, for instance, pre/post conditions or rely/guarantee (assumption/commitment) pairs.

Current work includes research on the ideas of platform-independent design of real-time systems. Although an axiomatic semantics has been given for a simple language with timing annotations [HvR97], more work is needed to obtain a formal semantics of a more realistic programming language and to incorporate this into PVS. Also the design process, supported by compositional proof rules, requires further study.

Acknowledgement

The design of the hybrid system example is strongly influenced by joint work with Onno van Roosmalen on platform-independent design.

References

[AH97] R. Alur and T. Henzinger. Modularity for timed and hybrid systems. In *Proc. Conf. on Concurrency Theory (CONCUR '97)*, pages 74–88. LNCS 1243, Springer-Verlag, 1997.

[dR85] W.P. de Roever. The quest for compositionality - a survey of assertion-based proof systems for concurrent programs, Part I: Concurrency based on shared variables. In *Proc. IFIP Working Conference 1985: The role of abstract models in computer science*, pages 181–207. North-Holland, 1985.

[HdR86] J. Hooman and W.P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current Trends in Concurrency*, pages 343–395. LNCS 224, Springer-Verlag, 1986.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoo94] J. Hooman. Correctness of real time systems by construction. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–40. LNCS 863, Springer-Verlag, 1994.

[Hoo95] J. Hooman. Verifying part of the ACCESS.bus protocol using PVS. In *Proceedings 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, pages 96–110. LNCS 1026, Springer-Verlag, 1995.

[Hoo97] J. Hooman. Verification of distributed real-time and fault-tolerant protocols. In *Algebraic Methodology and Software Technology (AMAST'97)*, to appear. LNCS 1349, Springer-Verlag, 1997.

[HvR97] J. Hooman and O. van Roosmalen. Platform-independent verification of real-time programs. In *Proceedings of the Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 183–192. IEEE Computer Society Press, 1997.

[MJ96] M. Joseph, editor. *Real-time Systems: Specification, Verification and Analysis*. Prentice Hall, 1996.

[Old85] E.-R. Olderog. Process theory: Semantics, specification and verification. In *Current Trends in Concurrency*, pages 442–509. LNCS 224, Springer-Verlag, 1985.

[Old91] E.-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Camebridge Tracts in Theoretical Computer Science*. Camebridge University Press, 1991.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Sha98] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy, editor, *Mathematical Programming Methodology*, to appear. 1998.

[SO98] M. Schenke and E.-R. Olderog. Transformational design of real-time systems – Part I: From requirements to program specification. *Acta Informatica*, to appear, 1998.

[VH96] J. Vitt and J. Hooman. Assertional specification and verification using PVS of the steam boiler control system. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, pages 453–472. LNCS 1165, Springer-Verlag, 1996.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*. LNCS 321, Springer-Verlag, 1989.

Compositional Proofs for Concurrent Objects*

Jerry James and Ambuj Singh

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, California 93106

Abstract. Objects are a convenient representation for building compositional open systems. Many object models exist in the literature and building a new proof system for each is infeasible. Instead of constructing a new proof system from first principles, we show how proof methodologies for non-object-oriented systems can be adapted. We give a sample object model that includes inheritance, active objects, and unbounded creation of both objects and threads. We show how a proof system for this model can be built from a modular concurrent logic. We also discuss the reuse of proofs during the construction of subclasses.

1 Introduction

The great promise of object-oriented technology is that it makes truly open systems possible. In a system where reusable and replaceable components are encapsulated as objects, upgrades can be effected by replacing individual components. Other system components are unaffected by such changes, as long as each replacement meets the specification of the original component. The dream of building applications by plugging together off-the-shelf software components is realizable in such a system.

The choice of an object-oriented approach to open system construction impacts the way in which system specification and verification is carried out. A compositional approach is natural, due to the encapsulated nature of the system components. However, a number of design choices must be made when creating a proof system. For example, the underlying object model must be both simple enough to support compositional reasoning, and rich enough to model actual systems. Also, the proof system should support both proof and code reuse in subclasses.

Many object models have been explored in the literature. Creating a proof system is difficult and error-prone work, so we want to avoid creating a new proof system for each model variation. We show how a non-object-oriented compositional proof system can be adapted to an object-oriented setting. To do so, we first describe a simple object-oriented model containing a synthesis of ideas from the literature. The model is founded on *atomic code fragments*, pieces of executable code that execute atomically with respect to the rest of the system.

* This work was supported in part by NSF grant CCR–9505807.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 301-326, 1998.
Springer-Verlag Berlin Heidelberg 1998

Methods are composed of such code fragments. Synchronization constraints on code fragments are expressed with *guards*.

We consider the form in which object properties are expressed. Such properties are put to two very different uses. The user of an object needs its abstract interface only, but the object implementor needs information about the concrete implementation of an object to create subclasses. For example, when creating a priority queue subclass from a generic queue parent class it is very important to know how the queue is stored internally. This leads us to distinguish between *abstract* and *concrete* properties of an object. Although this distinction is very language-dependent and perhaps somewhat arbitrary, it has been shown to be useful in structuring object class development [22].

We have chosen to express object properties in TLA [11] (Temporal Logic of Actions), although other concurrent logics could be used. Our proof methodology is founded on the abstract/concrete property dichotomy, and TLA's Composition Theorem [2]. We use this theorem to prove an object's abstract properties from its concrete properties, to prove properties of systems of objects, and to prove properties of a subclass while reusing proofs from the parent class where possible.

The overall proof methodology is illustrated in Figure 1. We first prove concrete properties of an object's methods from its source code. This step is language-dependent, and receives little attention in this paper. Next, the abstract property of the class is proved from the concrete properties via TLA's Decomposition Theorem. Classes which have had their abstract properties proved in this manner are then composed to prove the property of the system as a whole.

Code reuse is achieved through local reasoning and composition, but global reasoning is often difficult to avoid. An example of this problem is given in [15]. The authors demonstrate that Segall's PIF algorithm [20] (which they call the "gossip" algorithm) is very hard to reason about compositionally. We use this algorithm to illustrate our ideas, and give a compositional proof of its correctness.

The rest of the paper is organized as follows. We first describe Segall's PIF algorithm. We describe the object model in Section 2. In Section 3, we give the translation from the object model to TLA. We also present the proof methodology, showing the roles of both composition and concrete and abstract properties. Section 4 describes related work. We conclude with a few remarks on future research.

1.1 Example: Segall's PIF algorithm

The PIF (propagation of information with feedback) algorithm sends information (the "gossip") across the network from a root node (the "initiator"), then collects responses (the "feedback") back at the root. It is used for both information scattering and gathering operations, such as a distributed summation algorithm [24]. The property provided by this algorithm is that the root knows that all nodes in the network have received the gossip, and that the feedback it receives has been collected from all nodes in the network.

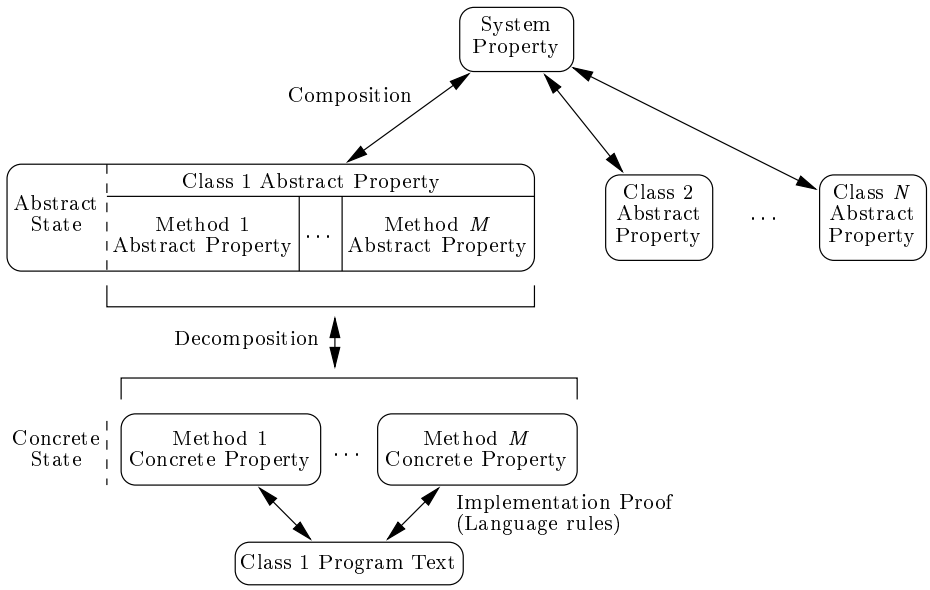


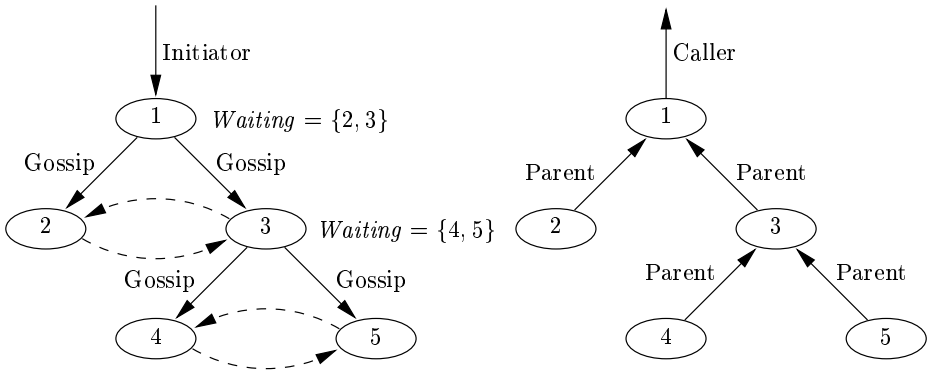
Fig. 1. Proof Methodology

The initiator begins the algorithm by sending the gossip to its immediate neighbors, as shown in Figure 2(a). It then waits for its neighbors to respond. If node j first receives the gossip from node i, then we say that i is the *parent* of j, as shown in Figure 2(b). Each node sends the gossip to all neighbors but its parent, then waits to receive a message from all nodes but the parent. Once this has occurred, it sends an acknowledgment to its parent. Once the initiator has received acknowledgments from all neighbors, it terminates the algorithm.

Our formulation of this algorithm in object-oriented terms is very simple. We employ a single class of objects, *Node*, with a private method *Gossip* and a public method *Start*. Correctness of the algorithm is dependent on a number of environment assumptions, which we state explicitly. Segall's PIF algorithm has been mechanically verified [8]. In this paper, we show how our general compositional technique can be applied to its proof.

2 Object Model

In this section, we describe an object-oriented programming model, for which we will construct a proof system in Section 3. Our object model is intended to reflect trends we expect to see in real concurrent systems of the future. It is similar to other concurrent object models, such as Orca [3] and Java [13].



(a) Outward phase (b) Logical tree

Fig. 2. Segall's PIF Algorithm

2.1 Basic Concepts

An *object* is a named, encapsulated state container. The state cannot be accessed directly; the only external access is through *method* calls, or *operations*. An object is a concurrent entity. That is, objects can have multiple *threads of control* (or just *threads*) running in them simultaneously. For full generality, we place no a priori upper bound on the number of threads or the number of objects in the system.

Atomic actions are expressed with *atomic code fragments* (or just *fragments*), which may range from individual machine instructions (or parts thereof) to large units of code. Since we can express arbitrary units of atomicity, arbitrarily complex actions can take place in a single atomic step. A *method* consists of a set of fragments with some control structure imposed on it. The control structure can use any of the usual constructs; e.g., sequencing and iteration.

Synchronization constraints (both mutual exclusion and condition synchronization) are expressed with a *guard* on each code fragment. The meaning of the guard is that, if the associated code fragment executes, it does so atomically starting in a state satisfying the guard. The guard may only reference the state of the associated object and the executing thread[1]. A *trivial* guard is true in all states. The guard plays much the same role as in Owicki and Gries' system [17], ensuring that the local state meets some criterion before the following atomic step takes place. However, our construction differs from their **await B then** S construction in allowing method calls inside an atomic fragment. We describe method call semantics in Section 2.3. Like Owicki and Gries, we assume that each

[1] However, since object references may be part of such state, a guard can contain method calls. A guard may be evaluated an arbitrary number of times, so we do not want guard evaluation to change the state of any object. Hence, method calls in guards may only be to "read-only" methods.

fragment is terminating, as a non-terminating atomic action can never have any visible effect on the system state.

Guards may become satisfied, and then be falsified again before a waiting thread has a chance to execute. To deal with starvation issues, we make two kinds of fairness assumptions about the system: *weak fairness* (a thread waiting on a continuously enabled guard eventually executes), and *strong fairness* (a thread waiting on an infinitely often enabled guard eventually executes). Although the set of threads in the system can be dynamic and unbounded, we show in section 3 that existing techniques for fixed sets of processes can be adapted.

For convenience in reasoning, we group objects with identical implementations into *classes*. Thus, proofs constructed for one member of a class apply to all instances of that class. For the moment, we assume that class implementations are fixed in advance, so that we may reason from the program text. This assumption does not hold for all languages. For example, Common Lisp [23] has a Meta-Object Protocol (MOP) that allows the user to construct and alter classes at runtime. Such systems introduce complexities that we do not address in this paper.

2.2 Segall's PIF Algorithm Revisited

Figure 3 illustrates the concepts we have introduced so far. It is pseudocode, in a graphical form, for a class named *Node* that implements Segall's PIF algorithm. The heavy lines denote atomic fragments. Arrows leading into atomic fragments are labeled with guard conditions. For convenience, each fragment is assigned a name, which is given at the top of the box denoting the fragment. Note that the control structure and the fragment structure can be viewed independently of one another. The dashed line and extra names in fragment *SetParent* will be explained shortly.

The state of objects of class *Node* consists of two variables, *Parent* and *Waiting*. The *Parent* variable holds a node reference, thereby implementing the tree shown in Figure 2(b). If $Parent = self$, then the node is the initiator, shown in Figure 2(b) as pointing to "Caller". The *Waiting* variable holds a set of node references. This is the set of neighbors to which the node has sent the gossip, but from whom it has not yet received the gossip. Initially, *Parent* is null and *Waiting* is empty. A non-null *Parent* and a nonempty *Waiting* imply that the node is participating in the outward phase of the gossip algorithm.

The public interface to the *Node* class is the *Start* method. It has two fragments, *Initiate* and *Done*, each with nontrivial guards. The guard on fragment *Initiate* requires the calling thread to wait if this node is already participating in an execution of the gossip algorithm. If it is not, then this node is marked as the initiator by setting *Parent* to *self*. The *Waiting* set is initialized to all neighbors, and the gossip is sent to all neighbors. The thread then waits on the guard of fragment *Done* until the *Waiting* set is empty, signifying that the algorithm has run to completion. At that point, it executes a trivial atomic fragment and exits, signaling to the caller that the gossip synchronization is complete.

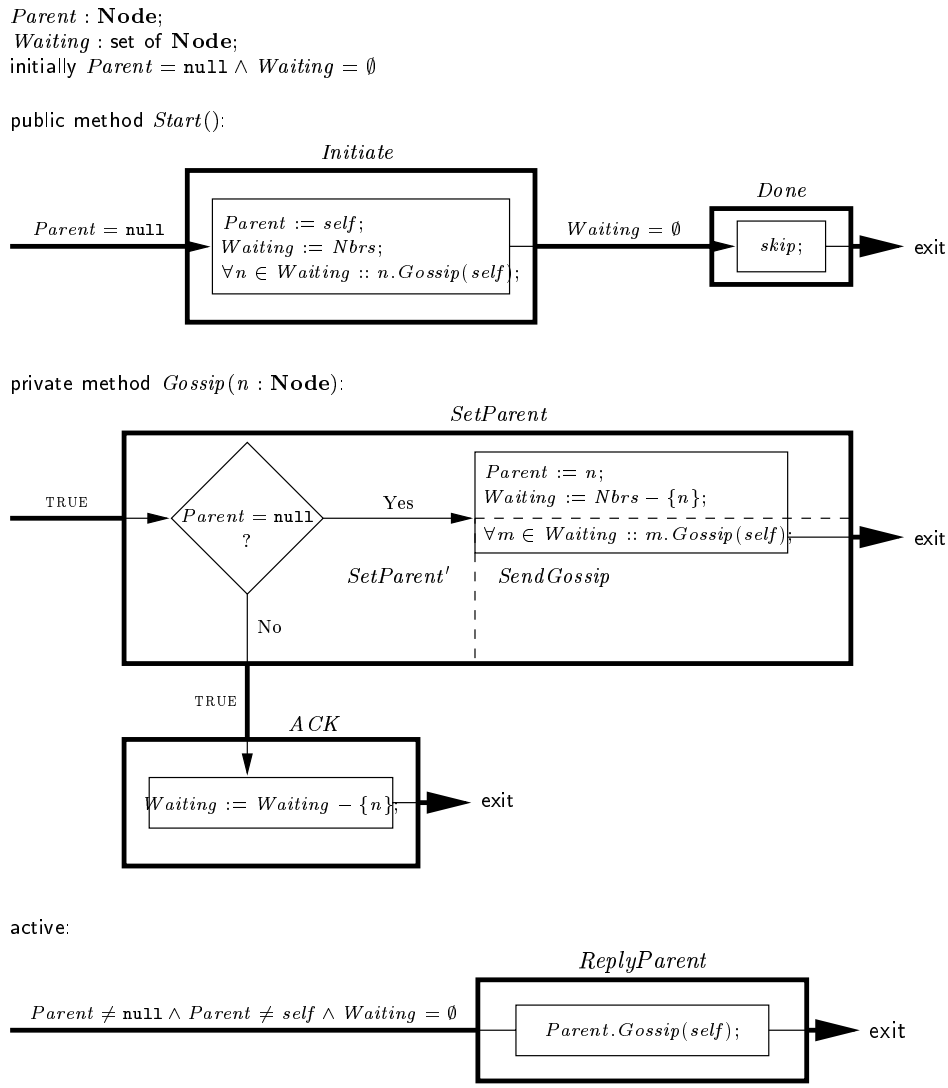


Fig. 3. Gossip algorithm implementation for class *Node*

The *Gossip* method is private, meaning that objects not of class *Node* do not have access to it. The argument to the method is of class *Node*, intended to be the caller sending the gossip. This method consists of two atomic code fragments, *SetParent* and *ACK*, each with a trivial guard. Fragment *SetParent* tests whether the gossip has been heard before, which is the case if *Parent* is non-null. If the gossip has not been heard, then *Parent* is set to the caller, and the *Waiting* set is initialized to all neighbors but *Parent*. Then the gossip is

sent to all nodes in *Waiting* (if any). If the node already had *Parent* set, then fragment *ACK* is executed. In that case, the caller must be some element of *Waiting*. Hence, we remove it from *Waiting*. The method then terminates.

Objects of class *Node* are *active objects*. That is, a thread of control is created when the object is created. The thread waits on the guard of fragment *ReplyParent*. For the initiating node, the thread never terminates, since *Parent* = *null UNTIL Parent* = *self* is satisfied. For all other *Node* objects, the guard becomes true when all neighboring nodes have sent the gossip. Then fragment *ReplyParent* can execute, which causes the node to call *Gossip* on its parent, signaling that it is done. The active thread then exits. In other types of objects, the active thread never terminates. Such objects exhibit a looping structure in the active code.

Notice that trivial alterations to guards or the structure of the atomic fragments can result in an incorrect algorithm. For example, consider the effect of splitting fragment *SetParent* into fragments *SetParent'* and *SendGossip* as indicated by the dashed rectangle in Figure 3. Suppose that *SetParent'* retains *SetParent*'s trivial guard, and *SendGossip* also has a trivial guard. Let thread t_1 execute fragment *SetParent'*, then be interupted before it can execute fragment *SendGossip*. Let thread t_2 execute fragment *SetParent'*, thereby removing the calling node, p, from *Waiting*. When thread t_1 resumes, it will fail to call *Gossip* on p, since $p \notin$ *Waiting*. Then p will never send an acknowledgement to its *Parent*, and by induction up the *Parent* tree, the algorithm never terminates.

Another small alteration that affects the correctness of the algorithm can be illustrated with the nontrivial guards of method *Start*. The guard of fragment *Initiate* can be removed under the guarantee that only one thread is executing in *Start* over all nodes at any one time. However, the guard of fragment *Done* is vital. It causes the thread executing *Start* to suspend awaiting responses from all neighbors. This guard ensures that the synchronization promised by the algorithm actually takes place: the initiating method does not terminate until all nodes have received the gossip.

2.3 Method Calls

We now consider the semantics of a method call inside an atomic fragment. The problem we must consider is this: what happens when a method call is made in the middle of a fragment, but the called method consists of multiple fragments? Worse yet, what if the execution of those methods results in more method calls, causing a cascade of calls to take place? What is the meaning of atomicity in this situation?

The answer lies in two mechanisms. First, method calls are composed of two distinct events: the *call* and the *return*. We allow these events to be separated in the code. The method call returns a unique handle which can be used to test for completion of the call, and to retrieve return values upon completion. Due to the test for completion, the handle can be used in a guard to block until the called method returns. A synchronous method call is then one where the calling

thread blocks awaiting completion immediately after making the call; otherwise, the call is asynchronous.

Second, if, inside an atomic fragment, a thread makes a method call and waits for its completion, then the call is *nested* inside the fragment. It must appear that the calling fragment and all events caused by the method call take place in a single atomic step. That is, the effect must be the same as though all other threads were suspended during execution of the method call. Nested atomic fragments are a way of implementing *multi-object operations*, atomic steps that access more than one object. They are closely related to nested transactions in concurrent databases. Note that such a fragment is not enabled unless all associated fragments can complete in a single atomic step. Hence, in general it is impossible to determine whether a nesting fragment is enabled solely from the state of its associated object.

3 Proof Methodology

Our goal is to construct a compositional proof system for the object model described in Section 2. However, we would like to avoid constructing it from first principles, as that is a time-consuming and error-prone task. Instead, we show how to construct a proof system from an existing compositional, though not object-oriented, concurrent proof system. We expect that other object models can have proof systems constructed in a similar manner.

Our model and proof methodology are relatively independent of the underlying logic. However, various logics may be easier or harder to work with in the context of this model. State-based models such as TLA [11] and Unity [5] favor a shared variable model. Action-based models such as I/O Automata [14] favor a message-passing model. A concurrent object system, however, draws from both kinds of models. Methods share state of the enclosing object; objects send messages to (make method calls on) one another. Hence, either kind of logic may prove awkward for dealing with some constructs. In spite of this difficulty, we have chosen TLA (described in [2, 11] and elsewhere in this volume) as our underlying logic. This choice was somewhat arbitrary. Our example could be reworked, for example, to use UNITY [5] with the compositional techniques of Collette and Knapp [6].

Our general method is shown in Figure 1. We begin with the program text, using language rules to prove that each method implements a concrete property. In our case, this means that we give a TLA representation of each method, based on the language semantics [2]. These concrete properties are expressed relative to some concrete representation of the object state. We then use TLA's Decomposition Theorem to prove that the concrete properties collectively implement the abstract properties of the object's methods. These abstract properties are expressed relative to some abstract representation of the object state. Then we

[2] We do not use any specific language in this paper, but assume that some language supporting the object model is used.

show, using compositional techniques, that the objects in a system collectively satisfy some desired property.

While this proof methodology may seem unnecessarily complicated, it has certain advantages. The concrete properties need only be proved once, from the program text. They can then be reused in subclasses. The abstract properties are closely related to the corresponding concrete properties. When creating a subclass, one simply shows that the changes made to the parent class do not violate the environment assumptions of unchanged methods. When that is shown, the abstract properties can be reused as well. Global reasoning can be postponed to the composition step, when the various object classes are glued together to form a system.

3.1 TLA Representations

Our first task is to choose a TLA representation for the entities in our object model (see Figure 4). An easy choice is to make each object class a TLA module. TLA differentiates between *internal variables*, those not accessible to the environment, *input variables*, those manipulated by the environment and read by the module, and *output variables*, those which give information to the environment. Hence, the state of an object is represented by internal variables, since encapsulation makes such variables inaccessible to the environment. Because we place no a priori bound on the number of instances of each class, the state variables are actually infinite arrays, indexed by a number uniquely assigned to each object[3].

Object Model	TLA Representation
class	module
class variables	module internal variables
instance variables	arrays of module internal variables
thread	arrays of module input and internal variables
guarded atomic fragments	atomic steps
method	collection of atomic steps
method call/return	wait sets
nested fragment	atomic step with environment assumption

Fig. 4. Object model—TLA correspondence

Instead of input and output variables, we have method calls and returns. Since they perform similar functions, we expect there to be a close relationship between the two. The obvious approach is to have a set of input variables for each method, corresponding to the parameters of the method, and output variables corresponding to return values. However, there are no bounds on the number of

[3] The number need only be unique over all classes of which the object is a member.

threads in the system, so we must resort to making such variables infinite arrays again, assigning each thread a unique number[4].

By choosing an interleaving representation in TLA, we make TLA actions atomic. Therefore, we can represent each atomic fragment as a TLA action. However, such actions can operate on an unbounded number of instances of the class, and an unbounded number of threads in an instance. Hence, references to object and thread variables must be indexed with the unique numbers discussed above. Atomic fragment guards are represented as predicates over the initial (unprimed) state of the object and thread.

We need a thread representation for recording internal thread states, including a "program counter". We employ the infinite array solution again, changing variable names as needed to avoid collisions, and introduce the notion of a *wait set* for the program counter. Specifically, we associate a set $\mathcal{Q}_M$ of *states* with each method M. All threads suspended on a guard in M are in one of these states. The set of states for method M is assumed to contain the distinguished states M^{call} and M^{return}. Method calls result in the creation of a new thread, which is placed into M^{call}. When a thread completes execution, it is placed in M^{return}. Hence, the *inputs* of an object are its associated *call* sets, and its *outputs* are its associated *return* sets.

Finally, we turn our attention to nested fragments. Encapsulation makes these difficult, as we cannot be sure when such fragments are enabled. The solution is to assume that the called method has some property P when called atomically. This assumption is used to compute the effect of the calling fragment, and is added to the environment assumption for that method. When classes are composed, we verify such assumptions. Note that TLA's Composition Theorem is able to cope with mutually recursive nested calls, in much the same way that it can be used to prove properties of mutually recursive modules.

To apply TLA's methods, we first identify system actions. The actions $\mathcal{A}_M$ associated with a method M are its set of fragments. That is, executing a fragment (taking an action) moves a thread from one state to another. We write $\mathcal{A}(t)$ to indicate the actions that are enabled for thread t; we write ENABLED (t) to indicate that $\mathcal{A}(t)$ is nonempty; that is, thread t is enabled. Figure 9 shows the set of actions (*NSTART*, *NDONE*, *SETPAR*, *ACK*, *REPLY*) for a *Node* object. The set of actions corresponds to the set of guards, or fragments, shown in Figure 2.

We introduce some shorthand to avoid redundant expressions in our properties. When moving a thread from one wait set to another, we use the following notation:

$$move\left(t\left(\bar{x}\right), q, s\right) \triangleq t\left(\bar{x}\right) = \text{NCHOOSE}\left(q\right) \wedge q' = q - \left\{t\left(\bar{x}\right)\right\} \wedge s' = s \cup \left\{t\left(\bar{x}\right)\right\}$$

The CHOOSE operator was used by Lamport [11] equivalently to Hilbert's ϵ operator [12]; that is, it represents a fixed but arbitrary choice. Our NCHOOSE

[4] Since a method call spawns a new thread, each thread executes in only a single object. Hence, the thread number need only be unique with respect to threads running on the same object.

operator is defined in terms of the ϵ operator to mean nondeterministic choice from a set (see Appendix A). The meaning of the *move* macro is that some thread t, with local variables $\bar{x}$, is moved from wait set q to wait set s. We will see later, when we consider the problem of unboundedness, that in some instances this is not strong enough; we need *fair* nondeterministic choice from a set.

The second piece of shorthand expresses the creation of a new thread. We create threads at method call time, as follows:

$$\begin{array}{lcl} call\,(t, m\,(\bar{x})) & \triangleq & \wedge\ t' = \text{NCHOOSE}\,(Threads) \\ & & \wedge\ Threads' = Threads - \{t'\} \\ & & \wedge\ m^{call\prime} = m^{call} \cup t'\,(\bar{x}) \end{array}$$

We assume two modules for assigning thread (*Threads*) and object (*Objects*) numbers. Each returns a unique ID, a "ticket", when called. As these modules are extremely simple, we omit them.

3.2 Rely/guarantee Properties

Objects are usually not intended to operate in arbitrary environments. Correctness of an implementation is frequently contingent on the environment using the object in a "well-behaved" manner. For example, Segall's PIF algorithm assumes that the environment will not simultaneously call *Start* on two distinct *Node* objects. We employ the *rely-guarantee* style of writing properties to state such requirements. An object relies on the environment to provide some property; in return, the object guarantees to provide some other property. We use the TLA $\stackrel{+}{\triangleright}$ operator to write such properties: $E \stackrel{+}{\triangleright} M$ means that the machine (guarantee) property M holds for at least one step longer than the environment property E holds.

In Segall's PIF algorithm, we want to specify that the call to *Start* on the root gossiper (or initiator) does not complete until all nodes have heard the gossip. In addition, we want to specify that a gossip step is one in which a node that has heard the gossip tells it to a neighboring node that has not. Finally, we want to say that a gossip step will eventually happen whenever it is possible (liveness).

The variables we need to define our system are given in Figure 5. N is a parameter, indicating the number of nodes in the system. The $Nbrs_i$ sets indicate the geometry of the network. The set $Nbrs_i$ contains all immediate network neighbors of node i. These two values are rigid variables; they are fixed for any given instance of the system.

The flexible variables hold the current state of the system. They are categorized as *internal variables*, those not accessible to the environment, *input variables*, those manipulated by the environment, and *output variables*, those which give information to the environment. The internal variables of most interest are those named $Heard_1, \ldots, Heard_N$. The variable $Heard_i$ is true iff node i has heard the gossip. The other flexible variables are all wait sets associated

Rigid Variables:

N : **Integer**;
$Nbrs_1, \ldots, Nbrs_N$: set of **Node**;

Flexible Variables:

1. Internal Variables
   $Heard_1, \ldots, Heard_N$: **Boolean**;
   $Start_1^{wait}, \ldots, Start_N^{wait}$: set of **Thread**;
2. Input Variables
   $Start_1^{call}, \ldots, Start_N^{call}$: set of **Thread**;
3. Output Variables
   $Start_1^{return}, \ldots, Start_N^{return}$: set of **Thread**;

Abbreviations:

$Path(i, j) = j \in Nbrs_i \vee (\exists\, k :: Path(i, k) \wedge Path(k, j))$
$start = \{Start_1^{call}, \ldots, Start_N^{call}, Start_1^{wait}, \ldots, Start_N^{wait}, Start_1^{return}, \ldots, Start_N^{return}\}$
$i =$ all input variables
$e =$ all internal variables
$o =$ all output variables
$all = i \cup e \cup o$

Fig. 5. Gossip system variables

with method *Start*. Input is given to the system by calling *Start*. Output from the system consists of an indication that method *Start* has finished execution. There is also an internal wait set associated with the method, whose utility we will see shortly.

Finally, we list some abbreviations that are used for convenience in writing. The statement $Path(i, j)$ means that there is some path in the network between nodes i and j. We use the abbreviations i, e, and o to refer to all input, internal, and output variables, respectively. The abbreviation *start* refers to all wait sets associated with method *Start*. Finally, *all* refers to all flexible variables.

We write our specification as shown in Figure 6. The machine specification M has hidden variables, e, the set of all internal variables. Hidden variables aside, the specification is represented in normal form by GM. The initial condition is given by $INIT$. It states that all of the $Heard_i$ variables are false, that all the wait sets are empty, that no node is in its own set of neighbors, that the network is symmetric (i.e., it is always possible to reply to a message), and that there is a path between any two nodes in the system. The next-state relation $NEXT$ states that there are three possible steps of the system: $START$, $GOSSIP$, and $DONE$. A $START$ step can be taken when a call has been made to the *Start* method. This step moves the calling thread into the internal wait set $Start^{wait}$ on the called node and sets its *Heard* variable to true. The $GOSSIP$ step can be taken whenever a node with *Heard* true is adjacent in the network to a node with *Heard* false. When the step is taken, the neighbor's *Heard* variable is set to true. The $DONE$ step is taken when *Heard* is true for all nodes. The calling thread

$$
\begin{array}{rcl}
M & \triangleq & \exists\, e :: GM \\
GM & \triangleq & INIT \wedge \Box[NEXT]_{\langle e,o\rangle} \wedge GF \\
INIT & \triangleq & \wedge \left(\forall\, i : 1 \leq i \leq N :: \neg Heard_n \wedge Start_i^{call} = \emptyset \wedge Start_i^{wait} = \emptyset\right) \\
 & & \wedge\ /\,(\forall\, i : 1 \leq i \leq N :: i \notin Nbrs_i) \\
 & & \wedge\ (\forall\, i,j : 1 \leq i,j \leq N :: (i \in Nbrs_j \iff j \in Nbrs_i) \wedge Path(i,j)) \\
NEXT & \triangleq & START \vee GOSSIP \vee DONE \\
START & \triangleq & \wedge\ \exists\, i : 1 \leq i \leq N :: move\left(t, Start_i^{call}, Start_i^{wait}\right) \wedge \neg Heard_i \wedge Heard_i' \\
 & & \wedge\ \text{UNCHANGED}\,(all \text{ EXCEPT } Heard_i) \\
GOSSIP & \triangleq & \wedge\ \exists\, i,j : 1 \leq i,j \leq N :: Heard_i \wedge \neg Heard_j \wedge j \in Nbrs_i \wedge Heard_j' \\
 & & \wedge\ \text{UNCHANGED}\,(all \text{ EXCEPT } Heard_j) \\
DONE & \triangleq & \wedge\ \exists\, i : 1 \leq i \leq N :: move\left(t, Start_i^{wait}, Start_i^{return}\right) \\
 & & \wedge\ (\forall\, j : 1 \leq j \leq N :: Heard_j) \\
 & & \wedge\ \text{UNCHANGED}\left(all \text{ EXCEPT } Start_i^{wait}, Start_i^{return}\right) \\
GF & \triangleq & WF_{\langle e,o\rangle}\,(NEXT)
\end{array}
$$

Fig. 6. Gossip system specification

is moved into the $Start^{return}$ wait set on its node, signaling to the environment that the operation has completed. Finally, we have the fairness condition GF. It states that $NEXT$ is executed in a weakly fair manner.

$$
\begin{array}{rcl}
E & \triangleq & \exists\, started, thrd, obj :: EINIT \wedge \Box[ENEXT]_{\langle i, started\rangle} \\
EINIT & \triangleq & \neg started \wedge \forall\, i : 1 \leq i \leq N :: Start_i^{return} = \emptyset \\
ENEXT & \triangleq & \wedge\ \neg started \wedge started' \wedge 1 \leq obj \leq N \wedge call\,(thrd, obj.Start()) \\
 & & \wedge\ \text{UNCHANGED}\left(i \text{ EXCEPT } obj.Start^{call}\right)
\end{array}
$$

Fig. 7. Gossip system environment assumptions

The gossip algorithm does not operate in a vacuum; it relies on certain assumptions about its environment, listed in Figure 7. The environment has three hidden variables: *started*, which records whether method *Start* has been called on some node, *thrd* and *obj*, which are junk variables used to set up a call to *Start*. The initial condition is described by *EINIT*: *started* is false, and all of the output variables of the system are empty. At each step, the environment can take some action that does not change the system's input variables and internal

variable *started*, or it can take step *ENEXT*. Due to the variable *started*, this step can be taken at most once. If it is taken, a call to method *Start* is invoked on some node. In this way, we ensure that the system is used at most once. Note that there is no fairness requirement; the environment need not ever call *Start*.

From these specifications, it is possible to prove that the system $E \xrightarrow{+} M$ satisfies the following two desirable properties:

$$(S)\ \Box\ (\exists\, i, t :: t \in Start_i^{return} \Rightarrow (\forall\, j :: Heard_j))$$
$$(P)\ \Box\ \left(\exists\, i, t :: t \in Start_i^{call} \leadsto t \in Start_i^{return}\right)$$

Property (S) is a safety property. It says that, in any state where there is a completed call to *Start*, all nodes have heard the gossip. Property (P) is a progress property. It states that if a call is made to *Start*, then it eventually completes. The proofs are very straightforward, and are left to the reader.

3.3 Concrete/Abstract Properties

We intend to use object properties in two very different ways. First, the user of an object wants to know its *type*; that is, the interface it provides and the guarantees made on that interface. Second, the object designer wants to know implementation details for the purpose of designing subclasses. The difference here is the same as that of Java *interface* and *class*. The first gives information about behavior of the object only; the second gives information about the implementation of that behavior. For the first purpose, the object properties should refer only to the object interface; internal details should be abstracted away. However, in the second case we do want internal details. Multiple implementations are possible for any given interface, so construction of a correct subclass often depends on implementation details of the parent class. (See [22] for a formal approach to making this distinction.)

We call properties of the first type *abstract*. They refer only to the externally visible portions of the object, respecting the encapsulation boundaries. Properties of the second type are called *concrete*. These properties carry details of the internal implementation of the object. It is important that there be a close correspondence between the concrete and abstract properties of an object. In this section, we show how to derive an object's abstract properties from its concrete properties. We do so using the Composition Theorem.

The variables we need to define a node are given in Figure 8, divided, as before, into rigid and flexible variables. The parameter m is the number of nodes in the neighborhood; those nodes are contained in the set *Nbrs*. The variables *Parent* and *Waiting* describe the most significant parts of the state of the node. *Parent* is a **Node**, the parent of this node in the abstract tree of gossipers shown in Figure 2(b). *Waiting* is a set of **Node**, the neighbors that have not yet contacted this node, as shown in Figure 2(a). The variabled named *junk* is needed only for spawning threads during a method call. Its value is never needed again. The other internal, input, and output variables are wait sets for methods *Start* and *Gossip*, both on this node and on its network neighbors.

Rigid Variables:

m : **Integer**;
$Nbrs$: set of **Node**;

Flexible Variables:

1. Internal Variables
   $Parent$: **Node**;
   $Waiting$: set of **Node**;
   $Start^{wait}$: set of **Thread**;
   $junk$: **Thread**
2. Input Variables
   $Start^{call}$, $Gossip^{call}$: set of **Thread**;
   $Start_1^{return}, \ldots, Start_m^{return}, Gossip_1^{return}, \ldots, Gossip_m^{return}$: set of **Thread**;
3. Output Variables
   $Start^{return}$, $Gossip^{return}$: set of **Thread**;
   $Start_1^{call}, \ldots, Start_m^{call}, Gossip_1^{call}, \ldots, Gossip_m^{call}$: set of **Thread**;

Abbreviations:

$ni =$ all input variables
$ne =$ all internal variables
$no =$ all output variables
$start = \{Start^{call}, Start^{wait}, Start^{return}\}$
$gossip = \{Gossip^{call}, Gossip^{return}\}$
$others = \{Start_1^{call}, \ldots, Start_m^{call}, Start_1^{return}, \ldots, Start_m^{return}\}$
$otherg = \{Gossip_1^{call}, \ldots, Gossip_m^{call}, Gossip_1^{return}, \ldots, Gossip_m^{return}\}$
$other = others \cup otherg$
$nall = ni \cup ne \cup no$

Fig. 8. Gossip node variables

Finally, we list some abbreviations that are used for convenience in writing. We use the abbreviations *ni*, *ne*, and *no* to refer to all input, internal, and output variables, respectively. The abbreviation *start* refers to all wait sets associated with method *Start*, and likewise for *gossip* and *Gossip*. We write *others* to refer to all wait sets related to method *Start* on other nodes, and likewise for *otherg* and *Gossip*. We write *other* for all wait sets on other nodes. Finally, *nall* refers to all flexible variables.

The node specification is given in Figure 9, and the environment assumptions are stated in Figure 10. This specification is a straightforward translation of the pseudocode of Figure 3 into TLA notation. We have added only two kinds of items: references to the wait sets, and a fairness condition. Note that we did not need to use wait sets in the *REPLY* step, as there is exactly one thread waiting on that step. We only need wait sets to represent method call and return semantics.

The environment assumptions are related somewhat to those for the entire system. The environment is allowed to make at most one call to method *Start* (due to the *nstarted* variable). That is, step *CSTART* is taken at most once. The

$$
\begin{array}{rcl}
NM & \triangleq & \exists\, ne :: GNM \\
GNM & \triangleq & NINIT \wedge \Box[NNEXT]_{\langle ne, no \rangle} \wedge GNF \\
NINIT & \triangleq & self \notin Nbrs \wedge Parent = \texttt{null} \wedge Waiting = \emptyset \wedge Start^{wait} = \emptyset \\
NNEXT & \triangleq & NSTART \vee NDONE \vee SETPAR \vee ACK \vee REPLY \\
NSTART & \triangleq & \wedge\ move\left(t, Start^{call}, Start^{wait}\right) \wedge Parent = \texttt{null} \wedge Parent' = self \\
 & & \wedge\ Waiting' = Nbrs \wedge (\forall\, n \in Nbrs :: call\,(junk, Gossip_n(self))) \\
 & & \wedge\ \text{UNCHANGED}\left(Start^{return}, gossip, others\right) \\
NDONE & \triangleq & \wedge\ move\left(t, Start^{wait}, Start^{return}\right) \wedge Waiting = \emptyset \\
 & & \wedge\ \text{UNCHANGED}\,(()\ nall\ \text{EXCEPT}\ Start^{wait}, Start^{return}) \\
SETPAR & \triangleq & \wedge\ move\left(t(n), Gossip^{call}, Gossip^{return}\right) \wedge Parent = \texttt{null} \wedge Parent' = n \\
 & & \wedge\ (\forall\, p \in Waiting' :: call\,(junk, Gossip_p(self))) \\
 & & \wedge\ Waiting' = Nbrs - \{n\} \wedge \text{UNCHANGED}\,(start, others) \\
ACK & \triangleq & \wedge\ move\left(t(n), Gossip^{call}, Gossip^{return}\right) \\
 & & \wedge\ Parent \neq \texttt{null} \wedge Waiting' = Waiting - \{n\} \\
 & & \wedge\ \text{UNCHANGED}\,(Parent, start, other\ \text{EXCEPT}\ ack) \\
REPLY & \triangleq & \wedge\ Parent \neq \texttt{null} \wedge Parent \neq self \wedge Waiting = \emptyset \\
 & & \wedge\ call\,(junk, Gossip_{Parent}(self)) \\
 & & \wedge\ \text{UNCHANGED}\left(nall\ \text{EXCEPT}\ reply, Gossip^{call}_{Parent}\right) \\
GNF & \triangleq & WF_{\langle ne, no \rangle}(NNEXT)
\end{array}
$$

Fig. 9. Gossip node specification

$$
\begin{array}{rcl}
NE & \triangleq & \exists\, thrd, nstarted :: NINIT \wedge \Box[NENEXT]_{\langle ni, nstarted \rangle} \\
NINIT & \triangleq & \wedge\ Start^{call} = \emptyset \wedge Gossip^{call} = \emptyset \wedge \neg nstarted \\
 & & \wedge\ (\forall\, n \in Nbrs :: Start^{return}_n = \emptyset \wedge Gossip^{return}_n = \emptyset) \\
NENEXT & \triangleq & CSTART \vee CGOSSIP \vee COTHER \\
CSTART & \triangleq & \wedge\ call\,(thrd, Start()) \wedge \neg nstarted \wedge nstarted' \\
 & & \wedge\ \text{UNCHANGED}\left(ni\ \text{EXCEPT}\ Start^{call}\right) \\
CGOSSIP & \triangleq & \wedge\ call\,(thrd, Gossip(p)) \wedge p \in Nbrs \\
 & & \wedge\ \left(\forall\, t(q) \in Gossip^{call} \cup Gossip^{return} :: p \neq q\right) \\
 & & \wedge\ \text{UNCHANGED}\left(ni, nstarted\ \text{EXCEPT}\ Gossip^{call}\right) \\
COTHER & \triangleq & \wedge\ (\forall\, p \in Nbrs :: p.Start^{call} \subseteq p.Start^{call'} \\
 & & \wedge\ p.Gossip^{call} \subseteq p.Gossip^{call'}) \wedge \text{UNCHANGED}\left(Start^{call}, Gossip^{call}\right)
\end{array}
$$

Fig. 10. Gossip node environment assumptions

environment can make calls to method *Gossip*, subject to certain restrictions. These are, that the parameter passed to that method must be a node in *Nbrs* and that no two calls to *Gossip* pass the same parameter (bounding the number of possible calls to the size of *Nbrs*). The environment is also allowed to call *Start* and *Gossip* on other nodes, without restriction. The abstract properties of a *Node* object should have reference simply to the *Start* method, since the *Gossip* method is not visible.

As noted above, there should be a close correspondence between the concrete and abstract properties of an object. In fact, the concrete properties should *implement* the abstract properties. However, the concrete properties are for each method in isolation, and the abstract properties are for the object as a whole. Environment assumptions on the state variables of an object should disappear, leaving only assumptions about the calling patterns and values of the environment. In short, we need to show that a set of lower level modules (the concrete properties) implement a set of higher level modules (the abstract properties). TLA's Decomposition Theorem is designed for just such a task.

The composition of $Node[1], \ldots, Node[n]$ has variables that do not appear in the system specification, and takes steps that do not appear there. Thus, we apply the refinement of Figure 11 to the composition. In fact, we apply only the refinement on states in the upper half of the figure; the refinement on actions in the bottom half will be shown to follow. All of the ACK, $REPLY$, and $NDONE$ steps of the composition collectively make up a single $DONE$ step of the system.

System Values	$\longleftrightarrow$	Local Values
Variables:		Variables:
$Nbrs_n$		$Nbrs[n]$
$Heard_n$		$Parent[n] \neq \texttt{null}$
$Start_n^{call}$		$Start[n]^{call}$
$Start_n^{wait}$		$Start[n]^{wait}$
$Start_n^{return}$		$Start[n]^{return}$
Steps:		Steps:
$INIT$		$\forall\, n :: NINIT_n$
$START$		$\exists\, n :: NSTART_n$
$GOSSIP$		$\exists\, n :: SETPAR_n$
$DONE$		$\exists\, n :: ACK_n$ $\exists\, n :: REPLY_n$ $\exists\, n :: NDONE_n$

Fig. 11. Refinement: concrete to abstract

Lemma 1. $\mathcal{C}(NM) = NINIT \wedge \Box[NNEXT]_{\langle ne, no \rangle}$

Proof. *NM* exhibits finite invisible nondeterminism, so the result follows from Proposition 2 of [1], and Propositions 1 and 2 of [2]. □

Lemma 2. $\mathcal{C}(M) = INIT \wedge \Box[NEXT]_{\langle e,o \rangle}$

Proof. M exhibits finite invisible nondeterminism, so the result follows from Proposition 2 of [1], and Propositions 1 and 2 of [2]. □

Lemma 3. $\mathcal{C}(E) = E$

Proof. E exhibits finite invisible nondeterminism, so the result follows from Proposition 2 of [1]. □

Lemma 4. *$NM \wedge NE \Rightarrow Parent = n$ is stable, $n \neq null$.*

LET: s, s' be states, $\mathcal{A}$ an action of $\mathcal{C}(NM)$.
PROVE: $(s \models Parent = n \wedge n \neq null) \wedge s\mathcal{A}s' \Rightarrow s' \models Parent = n$
⟨1⟩1. $\mathcal{A} = NSTART \rightarrow Parent = null$
⟨1⟩2. $\mathcal{A} = SETPAR \rightarrow Parent = null$
⟨1⟩3. $\mathcal{A} = NDONE \vee ACK \vee REPLY \rightarrow Parent' = Parent$
⟨1⟩4. $NE \rightarrow Parent' = Parent$
⟨1⟩5. Q.E.D.
⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, and Lemma 1.

□

Lemma 5. *$\mathcal{C}(E) \Rightarrow$ no node takes both an NSTART step and a SETPAR step.*

⟨1⟩1. $\mathcal{C}(E) \Rightarrow$At most one node, say n, takes an $NSTART$ step.
PROOF: Only the environment calls method *Start*(). It does so from $ENEXT$ only. $ENEXT$ is enabled in the initial state, but disables itself. Hence, $ENEXT$ is taken at most once, so at most one node has *Start*() called on it.
⟨1⟩2. Node n never takes a $SETPAR$ step.
⟨2⟩1. $NSTART_n \Rightarrow \Box(Parent_n = n)$
Lemma 4, since $NSTART$ sets $Parent$ to *self*.
⟨2⟩2. $Parent \neq null \Rightarrow \neg$ENABLED $(SETPAR)$
⟨2⟩3. Q.E.D.
PROOF: ⟨2⟩1 and ⟨2⟩2.
⟨1⟩3. No step on any node is enabled until n takes an $NSTART$ step.
⟨1⟩4. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3.

□

Lemma 6. $E \wedge \bigwedge_{i=1}^{N} NM_i \Rightarrow \forall\, i, j :: i$ *calls* $j.Gossip()$ *at most once.*

⟨1⟩1. i takes an $NSTART$ step but never a $GOSSIP$ step $\rightarrow i$ calls $j.Gossip()$ at most once.
⟨2⟩1. i calls all nodes in $Nbrs$ when it takes the $NSTART$ step.
⟨2⟩2. $REPLY$ is never enabled.
⟨2⟩3. Q.E.D.
⟨1⟩2. i takes a $SETPAR$ step but never an $NSTART$ step $\rightarrow i$ calls $j.Gossip()$ at most once.

⟨2⟩1. i calls all nodes in *Nbrs* but *Parent* when it takes the *SETPAR* step.
⟨2⟩2. i calls *Parent* if it takes a *REPLY* step.
⟨2⟩3. Q.E.D.
⟨1⟩3. Q.E.D.
PROOF: ⟨1⟩1, ⟨1⟩2, and Lemma 5.

□

Lemma 7. $\forall\, i :: \mathcal{C}(E) \wedge \bigwedge_{j=1}^{N} \mathcal{C}(NM_j) \Rightarrow NE_i$

PROOF:
LET: $x_k = \langle ne_k, ni_k, no_k \rangle$
$\hat{x}_k = \langle x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_N \rangle$
⟨1⟩1. $\forall\, i :: E \wedge \bigwedge_{j=1}^{N} NINIT_j \wedge \Box[\bigvee_{k=1}^{N} (NNEXT_k \wedge \hat{x}'_k = \hat{x}_k)] \Rightarrow NE_i$
⟨2⟩1. $\forall\, i :: E \wedge \bigwedge_{j=1}^{N} NINIT_j \wedge \Box[\bigvee_{k=1}^{N} (NNEXT_k \wedge \hat{x}'_k = \hat{x}_k)] \Rightarrow NEINIT_i$
PROOF: By the refinement, $\neg started \Rightarrow \bigvee_{i=1}^{N} \neg nstarted \Rightarrow \neg nstarted_i$, and all wait sets are empty.
⟨2⟩2. $\forall\, i :: E \wedge \bigwedge_{j=1}^{N} NINIT_j \wedge \Box[\bigvee_{k=1}^{N} (NNEXT_k \wedge \hat{x}'_k = \hat{x}_k)] \Rightarrow$
$\Box[NENEXT_i]_{\langle ni, i, nstarted_i \rangle}$
⟨3⟩1. $\forall\, i :: E \Rightarrow \Box[NENEXT_i]_{\langle ni, i, nstarted_i \rangle}$
⟨4⟩1. E takes a stuttering step → all input variables (i) and *started* are left unchanged. Hence, the input variables for node i (ni_i) and $nstarted_i$ (by the refinement) are left unchanged. Therefore, this is a stuttering step for NE_i.
⟨4⟩2. $ENEXT$ → a call to *Start* is made on a single node and *started* is set. Hence, this is either a $CSTART_i$ step (if node i was called), or a $COTHER$ step (if some other node was called).
⟨4⟩3. Q.E.D.
PROOF: ⟨4⟩1, ⟨4⟩2, and the definition of E.
⟨3⟩2. $\forall\, i, j :: E \wedge \Box[NNEXT_j \wedge \hat{x}'_j = \hat{x}_j)]_{\langle ne_j, no_j \rangle} \Rightarrow$
$\Box[NENEXT_i]_{\langle ni, i, nstarted_i \rangle}$
⟨4⟩1. NM takes a stuttering step → all output variables (no_i) are left unchanged. Since NM cannot change ni_i or $nstarted_i$, they are left unchanged as well. Hence, this is a stuttering step for NE_i.
⟨4⟩2. $NSTART_j$ → a $COTHER$ step, $CGOSSIP$ step, or both.
PROOF: Node j makes calls on *Gossip* on its neighbors, so this is a $COTHER$ step for node i. It can also be a $CGOSSIP$ step if node i is a neighbor of j. In that case, we must verify the conditions of $CGOSSIP$. Since all calls to *Gossip* are made with argument *self*, that follows by Lemma 6 and $INIT$. If the $NSTART_j$ step corresponds to both a $COTHER_i$ and a $CGOSSIP_i$ step, they can be considered to occur in either order since they operate on disjoint sets of variables.
⟨4⟩3. $NDONE_j, ACK_j$ → a stuttering step for NE_i.
⟨4⟩4. $SETPAR_j$ → Node j makes calls on *Gossip* on its neighbors (except for its parent). Hence this case is equivalent to the $NSTART_j$ case.

⟨4⟩5. $REPLY_j \rightarrow$ Node j makes a single call to *Gossip* on its parent. Hence this is either a $COTHER$ or a $CGOSSIP$ step for NE_i, but not both.

⟨4⟩6. Q.E.D.

PROOF: ⟨4⟩1, ⟨4⟩2, ⟨4⟩3, ⟨4⟩4, and ⟨4⟩5.

⟨3⟩3. Q.E.D.

PROOF: ⟨3⟩1, ⟨3⟩2, and simple propositional logic.

⟨2⟩3. Q.E.D.

PROOF: ⟨2⟩1, ⟨2⟩2, and the definition of NE_i.

⟨1⟩2. $\forall\, i :: E \wedge \bigwedge_{j=1}^{N} NNEXT_j \Rightarrow NE_i$

PROOF: ⟨1⟩1 and Proposition 6 of [2].

⟨1⟩3. Q.E.D.

PROOF: ⟨1⟩2 and Lemmas 3 and 1.

□

Lemma 8. $\mathcal{C}(E)_{+v} \wedge \bigwedge_{j=1}^{N} \mathcal{C}(NM_j) \Rightarrow \mathcal{C}(M)$

PROOF:

⟨1⟩1. $\bigwedge_{j=1}^{N} \mathcal{C}(NM_j) \Rightarrow \mathcal{C}(E) \perp \mathcal{C}(M)$

PROOF: Proposition 5 of [2], with $x = \langle e \rangle$, $e = o$, $y = \langle started, thrd \rangle$, and $m = i$.

⟨1⟩2. $\mathcal{C}(E) \wedge \bigwedge_{j=1}^{N} \mathcal{C}(NM_j) \Rightarrow \mathcal{C}(M)$

LET: $x_k = \langle ne_k, ni_k, no_k \rangle$

$\hat{x}_k = \langle x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_N \rangle$

⟨2⟩1. $E \wedge \bigwedge_{j=1}^{N} NINIT_j \wedge \Box[\bigvee_{k=1}^{N} (NNEXT_k \wedge \hat{x}'_k = \hat{x}_k)] \Rightarrow INIT$

⟨3⟩1. $\bigwedge_{j=1}^{N} NINIT_j \Rightarrow \forall\, i :: \neg Heard_i$

PROOF: $NINIT_J \Rightarrow Parent_j = \texttt{null}$, and by the refinement, $Parent_j = \texttt{null} \Rightarrow \neg Heard_j$.

⟨3⟩2. $\bigwedge_{j=1}^{N} NINIT_j \Rightarrow \forall\, i :: Start_i^{wait} = \emptyset \wedge Start_i^{return} = \emptyset$

PROOF: Definition of $NINIT_j$.

⟨3⟩3. Q.E.D.

PROOF: ⟨3⟩1 and ⟨3⟩2.

⟨2⟩2. $E \wedge \bigwedge_{j=1}^{N} NINIT_j \wedge \Box[\bigvee_{k=1}^{N} (NNEXT_k \wedge \hat{x}'_k = \hat{x}_k)] \Rightarrow \Box[NEXT]_{\langle e, o \rangle}$

⟨3⟩1. E takes a step $\rightarrow$ then e and o are left unchanged, so this is a stuttering step for $\Box[NEXT]_{\langle e, o \rangle}$.

⟨3⟩2. $NSTART_j \rightarrow START$, by the refinement.

⟨3⟩3. $SETPAR_j \rightarrow GOSSIP$, by the refinement, since all calls to *Gossip* are made with argument *self*, the argument is always non−`null`.

⟨3⟩4. $ACK_j, REPLY_j, NDONE_j \rightarrow DONE$, by the refinement.

⟨3⟩5. Q.E.D.

PROOF: ⟨3⟩1, ⟨3⟩2, ⟨3⟩3, and ⟨3⟩4.

⟨2⟩3. Q.E.D.

PROOF: ⟨2⟩1, ⟨2⟩2, and the definition of $\mathcal{C}(M)$.

⟨1⟩3. Q.E.D.

PROOF: $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and Proposition 4 of [2].

□

Lemma 9. $E \wedge \bigwedge_{j=1}^{N} NM_j \Rightarrow M$

PROOF:
$\langle 1 \rangle 1$. $E \wedge \bigwedge_{j=1}^{N} NM_j \Rightarrow \mathcal{C}(M)$
PROOF: Lemma 8, since $E \Rightarrow \mathcal{C}(E)_{+v}$ and $NM_j \Rightarrow \mathcal{C}(NM_j)$.
$\langle 1 \rangle 2$. $E \wedge \bigwedge_{j=1}^{N} NM_j \Rightarrow GF$
PROOF: $\bigwedge_{j=1}^{N} GNF_j \Rightarrow GF$, due to the refinement, as shown in Lemma 8.
$\langle 1 \rangle 3$. Q.E.D.
PROOF: $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and Lemma 2.

□

Now we show that the proof obligations of the Composition Theorem have been met.

Theorem 10. $\models \bigwedge_{j=1}^{n} (NE_j \xrightarrow{+} NM_j) \Rightarrow E \xrightarrow{+} M$

PROOF:
$\langle 1 \rangle 1$. $\forall\, i :: \mathcal{C}(E) \wedge \bigwedge_{j=1}^{n} \mathcal{C}(NM_j) \Rightarrow NE_i$
PROOF: Lemma 7.
$\langle 1 \rangle 2$. $\mathcal{C}(E)_{+v} \wedge \bigwedge_{j=1}^{n} \mathcal{C}(NM_j) \Rightarrow \mathcal{C}(M)$
PROOF: Lemma 8.
$\langle 1 \rangle 3$. $E \wedge \bigwedge_{j=1}^{n} NM_j \Rightarrow M$
PROOF: Lemma 9.
$\langle 1 \rangle 4$. Q.E.D.
PROOF: Steps $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and $\langle 1 \rangle 3$ and the Composition Theorem.

□

Notice how the global reasoning needed in this example (see [15]) is postponed until the final composition step. This is one of the benefits of our approach; low-level (i.e., concrete) properties can be constructed with purely local reasoning. In the absence of private methods and nested objects, the abstract properties of individual objects can also be constructed with purely local reasoning.

In the gossip algorithm case, the abstract properties are the same as the properties of the entire system, given in Figures 6 and 7, since there are no other objects in the system. In a more complex example, we would place other objects in the system, and use the Composition Theorem to show how the synchronization provided by the *Node* objects helps the rest of the system meet its desired properties.

3.4 Fairness

Our translation from the object model to TLA has an undesirable property. A TLA step is enabled iff there is some thread in the associated wait set such that the associated guard is satisfied. However, an unbounded number of threads may

be waiting on the same satisfied guard. In this case, specifying fairness of the TLA step is not equivalent to specifying fairness over threads. The action can be selected fairly, and some thread waiting on that action can starve, if there is always at least one other thread in the same wait set. To express fairness over all threads, we also need fairness of the mechanism for selecting threads from a thread set.

To express such a fair selector, we must take the thread guards into account. If a wait set can simultaneously have threads with both satisfied and unsatisfied guards (this can be the case for guards that reference thread variables), then we want to make sure the selector chooses a thread with a satisfied guard. We refer to such threads as *enabled*. A weakly fair selector will eventually choose any thread that is enabled every time the action is enabled. A strongly fair selector will eventually choose any thread that is infinitely often enabled when the action is enabled.

In TLA$^+$, the CHOOSE operator is used to make a fixed, but arbitrary choice from a set (it is equivalent to Hilbert's ϵ-operator). We introduce the NCHOOSE operator, which nondeterministically selects an element of a set (see Appendix A for details). We introduce two thread-choosing operators based on NCHOOSE, one providing weak fairness and the other providing strong fairness. We assume that only one of the two operators is ever used on any given set of threads.

Definition 11. Let T be a set of threads. WCHOOSE(T) and SCHOOSE(T) each nondeterministically evaluate to some enabled element of T, such that:

$$\forall\, t \in T :: \Box\,(|T| < \infty) \wedge \mathrm{WF}_T\,(\mathit{WCHOOSE}(T)) \Rightarrow \\ \Box\Diamond\,(\neg \textsc{Enabled}\,(t)) \vee \Box\Diamond\,(t = \textsc{Wchoose}(T))$$

and

$$\forall\, t \in T :: \Box\,(|T| < \infty) \wedge \mathrm{SF}_T\,(\mathit{SCHOOSE}(T)) \Rightarrow \\ \Diamond\Box\,(\neg \textsc{Enabled}\,(t)) \vee \Box\Diamond\,(t = \textsc{Schoose}(T))$$

That is, if T is always finite and WCHOOSE(T) is executed with weak fairness, then any given t that is continuously enabled when the WCHOOSE(T) step is enabled will eventually be chosen. If T is always finite and SCHOOSE(T) is executed with strong fairness, then eventually any SCHOOSE(T) step is taken at a time when t is not enabled, or t is eventually chosen. One consequence of these definitions is that WCHOOSE(T) and SCHOOSE(T) steps are only enabled when there is some enabled element of T.

3.5 Inheritance

When deriving subclasses, we want to reuse as much of the correctness proof for the parent class as possible. For example, we have given a one-shot version of Segall's PIF algorithm; consider a subclass that implements the algorithm in a reusable fashion. The subclass would behave just like the parent class, except

that it also has a *reset* method which sets its state back to the initial value. We expect the correctness proofs to be nearly identical. In fact, all we need do is assume that the system has at most one outstanding call to *Start* at any one time. Then we can divide any execution of the subclass into phases such that each phase is identical to an execution of the parent class.

Methods that are identical in the parent class and the subclass may not need to be completely reverified. If the environment assumptions still hold, then the Decomposition Theorem's premises are discharged. Hence, when verifying a subclass, we take the following steps:

1. Verify concrete properties of new/modified methods;
2. Reverify environment assumptions of unmodified methods.
3. Verify abstract properties of new/modified methods.

Step 2 discharges our obligations for unmodified methods; the abstract property still holds.

4 Related Work

Non-object-oriented compositional techniques have been pioneered by Barringer, Kuiper, and Pnueli [4], Pnueli [19], Stark [21], Pandya and Joseph [18], Misra [16], and Abadi and Lamport [2], to name some prominent examples. Some of these techniques ([19] and [21]) are *compositionally incomplete*; that is, some properties of a system cannot be deduced from properties of the modules and the composition rule.

The POOL object-oriented language is given a proof system by de Boer [7] using rely-guarantee properties. The strength of this system is that it handles a dynamic set of processes. However, the composition rule can only prove pure safety properties; there is no composition rule for progress properties. Furthermore, there is no mechanism for dealing with inheritance. The restriction to safety properties simplifies the composition rule. It consists of placing statements that might modify the global context in *bracketed sections*, and then applying a *Cooperation test* to the bracketed sections of each class. If all classes pass the cooperation test with respect to some invariant I, then the composition of all the rely properties guarantees I and the composition of all guarantee properties.

DisCo [9] is an object-oriented specification language for reactive systems based on the joint action model of execution (see [10] for an example of its use). It has guarded multi-object actions instead of single-object methods. Like our model, DisCo has a formal basis in TLA; i.e., it can be considered another example of an object model with a TLA-based proof system. Its notion of object is somewhat different from ours, due to the use of joint actions rather than threads of control.

Manohar and Sivilotti [15] provide a non-object-oriented composition rule based on *modified rely-guarantee properties*. These are rely-guarantee properties which can only refer to local variables (i.e., variables local to the process to which the property refers). They are able to prove both safety and progress properties

with their composition rule. They use channel variables for communication between processes that act very much like our thread sets. Their *next* operator is equivalent to Misra's *co* operator and Abadi and Lamport's $\stackrel{+}{\triangleright}$ operator.

Manohar and Sivilotti found that they were unable to prove all properties of a composition and blamed it on the modified properties. However, the real culprit is their composition rule, which is as follows. Given modules P_1, P_2, ... with rely-guarantee properties (R_1, G_1), (R_2, G_2), ..., their composition has the rely-guarantee property $(R, R_1 \wedge G_1 \wedge R_2 \wedge G_2 \wedge \ldots)$ if:

$$
\begin{aligned}
& R \Rightarrow R_1 \\
& R \wedge R_1 \wedge G_1 \Rightarrow R_2 \\
& R \wedge R_1 \wedge G_1 \wedge R_2 \wedge G_2 \Rightarrow R_3 \\
& \ldots
\end{aligned}
$$

This is a very weak composition rule, and so is unable to handle Segall's PIF algorithm.

5 Conclusion

We have described a simple concurrent object model, and shown how to construct a proof system for it by using an existing modular proof system, TLA. We have shown how compositional reasoning may be applied to facilitate both proof and code reuse. Our approach distinguishes between the *concrete* and *abstract* properties of an object. The concrete properties are closely related to the source code. They carry information about the implementation of the object needed by subclass designers. The abstract properties hide implementation details, and deal only with the properties visible to the object user. We showed how to verify the abstract properties of an object by applying the Decomposition Theorem to the concrete properties of that object. The properties of a system are verified in a similar fashion, by composing the properties of its component objects. We introduced fair thread choice operators to express fairness over a dynamic, unbounded set of threads.

Our object model was designed with trends in actual concurrent systems in mind. We are exploring the model further by building a distributed implementation of Java, with some language modifications to support the model. The system design itself is object-oriented, and we plan to verify the correctness of some components with the methodology outlined in this paper. We expect this process to yield further insights into the structure of such proofs, and the pitfalls that await the verifier of concurrent object systems.

Acknowledgments: The authors wish to thank Wim Hesselink for his many constructive comments and helpful suggestions, as well as the other conference participants whose insights helped improve this paper.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–84, May 1991.
2. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–34, May 1995. Also SRC Research Report 118.
3. Henri E. Bal. *Programming Distributed Systems*. Prentice-Hall, New York, 1991.
4. Howard Barringer, Ruurd Kuiper, and Amir Pnueli. Now you may compose temporal logic specifications. In *STOC '84*, pages 51–63, Washington, D.C., USA, 30 April–2 May 1984.
5. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, USA, 1988. Reprinted with corrections, May 1989.
6. Pierre Collette and Edgar Knapp. A foundation for modular reasoning about safety and progress properties of state-based concurrent programs. *Theoretical Computer Science*, 183(2):253–79, September 1997.
7. Frank S. de Boer. A proof system for the parallel object-oriented language POOL. In M. S. Paterson, editor, *ICALP '90*, volume 443 of *Lecture Notes in Computer Science*, pages 572–85, Warwick University, England, 16–20 July 1990. Springer-Verlag.
8. Wim H. Hesselink. A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9(2):208–26, 1997.
9. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proc. 12th Int. Conf. on Software Eng.*, pages 63–71, 1990.
10. Reino Kurki-Suonio. Incremental specification with joint actions: The RPC-memory specification problem. In Manfred Broy, Stephan Merz, and Katharina Spies, editors, *Formal Systems Specification: The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*, pages 375–404. Springer, Berlin, 1996.
11. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
12. A. C. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol*. Gordon and Breach, New York, 1969.
13. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1997.
14. Nancy A. Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC '87*, pages 137–51, Vancouver, British Columbia, Canada, 10–12 August 1987.
15. Rajit Manohar and Paolo A. G. Sivilotti. Composing processes using modified rely-guarantee specifications. Technical Report CS-TR–96–22, California Institute of Technology, Pasadena, CA 91125, 12 June 1996.
16. Jayadev Misra. New UNITY. Unpublished book.
17. Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
18. Paritosh K. Pandya and Mathai Joseph. P-A logic—a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.
19. Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series F*, pages 123–44. Springer-Verlag, Heidelberg, 1985.

20. Adrian Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT–29(2):23–35, January 1983.
21. Eugene W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *FST&TCS '85*, volume 206 of *Lecture Notes in Computer Science*, pages 369–91, New Delhi, India, 16–18 December 1985. Springer-Verlag.
22. Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA '95*, pages 200–14, Austin, TX, USA, 15–19 October 1995.
23. Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, USA, 2nd edition, 1990.
24. Frits W. Vaandrager. Verification of a distributed summation algorithm. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*, pages 190–203, Philadelphia, PA, USA, 21–24 August 1995. Springer.

A Hilbert's ϵ-operator and nondeterministic choice

Although Hilbert's ϵ operator [12] represents a fixed arbitrary choice, we can specify a sequence of nondeterministic choices with it. We assume a (possibly infinite) set of data items D. The universe $U = \mathbb{N} \times D$ is a set of 2-tuples of the form $\langle i, d \rangle$, i a natural number and d a data item. For any $u = \langle i, d \rangle \in U$, we define two projection operators:

$$\begin{aligned} \mathit{time}\ (\langle i, d \rangle) &= i \\ \mathit{val}\ (\langle i, d \rangle) &= d \end{aligned}$$

A *sequence* $[u_1, u_2, \ldots]$ is an element of U^ω, the set of finite and infinite strings over U. A *temporal sequence* is a sequence such that $\forall i, j, d : \langle i, d \rangle \in u_j :: i = j$. Let $S = [u_1, u_2, \ldots]$ be a temporal sequence. Then the *choice at step* i of S is $\mathit{val}\ (\epsilon\, \langle i, d \rangle\, u_i)$, written $\textsc{Nchoose}_i(S)$. Since we are including temporal values in the set operated on by ϵ, we can represent any sequence of choices out of the sets u_1, u_2, ...

Fair nondeterministic choice can also be specified. A *weakly fair sequence* is a temporal sequence that is finite or satisfies

$$\forall d \exists i :: (\forall j : i \leq j :: \langle j, d \rangle \in u_j) \Rightarrow (\forall k \exists l : i \leq k \leq l :: \textsc{Nchoose}_l = d)$$

. A *strongly fair sequence* is a temporal sequence that is finite or satisfies

$$\forall d :: (\forall i \exists j : i \leq j :: \langle j, d \rangle \in u_j) \Rightarrow (\forall k \exists l : k \leq l :: \textsc{Nchoose}_l = d)$$

.

An overview of compositional translations

Theo M.V. Janssen

Computer Science, University of Amsterdam Plantage Muidergracht 24,1018TV Amsterdam, The Netherlands
email: theo@fwi.uva.nl

Abstract. Translations from one language to another arise in many fields of science: in computer science (compilers, data base views), logic (embeddings), natural language (translation), and philosophy (Montague grammar). In all these fields one can find the same method: compositional translation, or in mathematical formulation, algebraic translation. In some fields it is a standard method, in other fields a rare approach. The aim of this paper is to give an overview of compositional translations. Special attention will be given to the notion 'correct translation' (which can be formalized by commutativity of a diagram). Furthermore, the first steps will be will be made towards a mathematical theory of translating.
keywords Translation, correctness, compiler, embedding, view update, natural language, semantics, commutative diagram.

1 Introduction

In philosophy of language the following principle is well known :

> The meaning of a compound expression is a function of the meanings of its parts and of the rule by which the parts are combined

It is called the compositionality principle or 'Frege's principle'. A survey of the role of the principle in philosophy of language is given in [14].

Computer scientists have been attracted by compositionality since the first steps towards a formal semantics. Two early quotations (from 1975) are by Milner : '... any abstract semantics should give a way of composing the meanings of the parts into the meaning of the whole ...' [21, p. 157], and Mazurkiewicz 'One of the most natural methods of assigning meanings to programs is to define the meaning of the whole program by the meaning of its constituents' [19, p. 75]. In these cases there is no awareness of the relation with the (older) principle from philosophy. The first publication that mentions the compositionality principle in connection with programming language seems to be one from Janssen & van Emde Boas at the conference celebrating 100 years Frege's Begriffschrift [7].

The principle characterizes the connection between a language and its semantic model. The mathematical formulation of this connection is that the language is organized as an algebra, the meanings form a similar algebra, and meaning assignment is a homomorphism. However, in most practical cases this situation

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 327-349, 1998.
Springer-Verlag Berlin Heidelberg 1998

does not arise so directly. Usually a logical language is used to describe the meanings. Since for logic an interpretation is defined, this indirectly defines a compositional meaning assignment. So compositional meaning assignment means in practice a compositional *translation* into a logic.

Meaning assignment is just one example of a translation. Translations occur in many fields of science, and between several kinds of languages, and for several purposes. One finds them in computer science when a computer program is compiled in a machine code, when a view update is translated in a data base update, but also when a natural language is translated in another one by the computer. They arise when a logic is embedded in another logic, or when meanings are given for the expressions of natural language by defining a translation into logic. In many of these fields independently the same idea arose: use an algebraic (compositional) translation.

Certainly not all translations are intuitively correct, and therefore in many disciplines formal correctness notions are given. Often this notion is based upon the commutativity of some diagram, and if that was not the case, it could be brought in that form. Surprisingly, in the field of algebraic compiler construction there is no consensus on the notion 'correctness'. We will discuss this issue extensively, and arguments in favor of one correctness notion will be given. It is not the correctness notion that is used in most articles on compilers, but it is the oldest one. The same correctness notion can also be found in several other fields where translations arise.

The aim of this article (preliminary version of [15]), is to compare the algebraic methods from these different fields and to discuss some of the fundamental issues, in particular the notion of correctness of translation. It turns out that the publications from the different fields of translation discuss the same issues and use related notions. Hence there seems to be a common basis for a general algebraic theory of translation. In this paper a first, small step will be made.

2 Translating from programming language to programming language

2.1 Compilers

A compiler can be conceived of as a translation from a source language SL (for instance a high level programming language) to a target language TL (for instance some assembler language). It has been proposed by several authors to deal with compiler design in an algebraic way. In diagram 1 the components are mentioned that will arise in the discussion: all corners are algebras, and all arrows are homomorphisms. Below we consider one proposal for compiler correctness, in the next sections four other proposals will be considered.

The intuitive ideal about a translation is that it formulates precisely the same information in another language. No information is added, nothing gets lost: the meaning of the target language is, if not identical, at least isomorphic with the meaning of the source language. This ideal is formulated by Polak [33] who

requires *Enc* (encode) to be the identity, and Mosses [27] who assumes *Enc* and *Dec* (decode) to be isomorphisms. Correctness is then defined as commutativity of diagram 1.

As we shall see in the next sections, when compiler correctness has to be proven by describing *Enc* or *Dec*, this is never done by proving one of them to be an isomorphism. The explanation is that the involved languages are very different and have different meanings. In the next sections examples will be given which illustrate this point. Therefore it is not surprising that in the final version of Mosses paper [27], that is [28], a different correctness notion is used (viz. the one from sect. 2.2). The ideal of identity or isomorphism is reached only if the situation is designed with that aim (see sect. 6), or if one takes an abstract point of view (see sect. 9), but these cases do not arise, as far as I know, in articles about compilers.

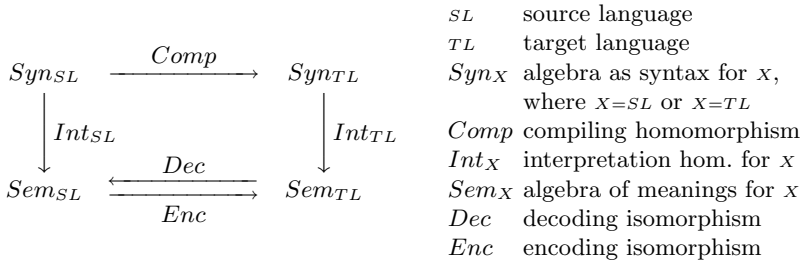
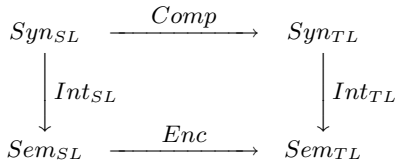


Diagram 1. Compiler correctness according to Polak [33, p. 17] and Mosses [27, p. 189]: there are isomorphisms *Dec* and *Enc* such that the diagram commutes in both directions.

2.2 The correctness notion of Thatcher et al.

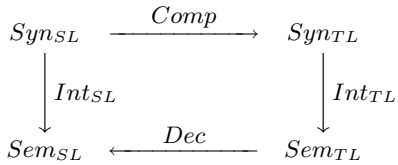
Certainly the most influential proposal for algebraic compiler construction is the one of Thatcher, Wagner and Wright [40]. It defines compiler correctness as commutativity of diagram 2. The proposal of Thatcher et al. is based upon the work of Morris [25] and aims at correcting, refining and completing that proposal. They do not present Morris' original version (diagram 3); instead they say that his advise was to use diagram 2. This is justified in a footnote where they say that 'Morris' diagram had $Dec\colon Sem_{TL} \to Sem_{SL}$, though in the text he uses $Enc\colon Sem_{SL} \to Sem_{TL}$'. Thus they suggest that by accident the wrong diagram was incorporated in Morris' article. We shall return to this point in sect. 2.3.

Let us now consider the example of a compiler given by Thatcher et al. The source language is a fragment of a programming language with 18 syntactic operations (forming for instance assignments, conditionals and the while-construction). Sem_{SL} is a kind of denotational semantics. Its primitive oper-



SL	source language
TL	target language
Syn_X	algebra as syntax for X, where $X{=}SL$ or $X{=}TL$
$Comp$	compiling homomorphism
Int_X	interpretation hom. for X
Sem_X	algebra of meanings for X
Enc	encoding homomorphism

Diagram 2. Compiler correctness according to Thatcher et al. [40]: there is a homomorphism *Enc* such that the diagram is commutative



SL	source language
TL	target language
Syn_X	algebra as syntax for X, where $X{=}SL$ or $X{=}TL$
$Comp$	compiling homomorphism
Int_X	interpretation hom. for X
Sem_X	algebra of meanings for X
Dec	decoding homomorphism

Diagram 3. Compiler correctness according to Morris [25]: there is a homomorphism *Dec* such that the diagram is commutative

ations are *assign* and *fetch*, together with general algebraic and arithmetical operations. The meanings of the syntactic operations are described by polynomials over the primitive operations. The target language consists of flow charts, and its meanings in Sem_{TL} are unfolded flow charts. As they say, the radical improvement in comparison with Morris lies in this part: making flowcharts algebraic. *Enc* is defined as a mapping from the carriers of Sem_{SL} into corresponding carriers in Sem_{TL}. For instance, the functions from *Environments* to *Environments* are mapped to the functions from $\langle Stacks \times Environments \rangle$ to $\langle Stacks \times Environments \rangle$ that leave the stack unchanged. Next it is proven that *Enc* is a homomorphism. The proof requires the checking of the 18 syntactic operations, and uses many properties of Sem_{SL} and Sem_{TL}. As a consequence both $Enc \circ Int_{SL}$ and $Int_{TL} \circ Comp$ are homomorphisms from Syn_{SL} to Sem_{TL}. Since Syn_{SL} is an initial algebra, there is a unique homomorphism from Syn_{SL} to Sem_{TL}, hence $Enc \circ Int_{SL} = Int_{TL} \circ Comp$, so the diagram commutes.

The definition of compiler correctness as commutativity of diagram 2 is, intuitively, not satisfactory. In the left hand side of the diagram some programming language is given, together with the intended meaning of this language. The right hand side should tell a machine how to perform the actions described by the programming language. Since a compiled program should do what it has to do according to the semantics of the programming language, going through a compiler should be a way to obtain the originally intended semantics. Hence

the meanings of the target algebra should be interpreted in the original semantic algebra of the programming language in order to see whether the compiler yields the intended results. So for a correct compilation there has to be a decoding mapping Dec: $Sem_{TL} \rightarrow Sem_{SL}$ such that diagram 3 commutes, i.e. the diagram of Morris gives the appropriate definition.

These considerations can be illustrated by two examples of compilers for which the definitions diverge: an intuitively incorrect compiler, and an intuitively correct one.

Example 1. Let SL be a programming language that has both positive and negative numbers, and that has multiplication as operation. Suppose that in the interpretation of TL all information concerning signs is thrown away: Sem_{TL} operates only with positive numbers. Of course, this is not what intuitively would be called a 'correct compiler'. According to the definition of Thatcher et al., this would be a correct compiler since there is a homomorphism Enc such that diagram 2 commutes (let the image of a number be its absolute value). Diagram 3 cannot be made commutative: there exists no decoding Dec that could achieve this, because a positive number in Sem_{TL} then should have two images in Sem_{SL}. So, according to the definition of Morris, the proposed compiler is incorrect, and this is in accordance with the intuition.
□

Example 2. Suppose SL has the syntactically distinct expressions -0 and $+0$, and both have semantic interpretation: the number zero. Suppose moreover that they correspond with two distinct expressions in TL (again -0 and $+0$), and that their interpretation in Sem_{TL} differs as well (say a different sign bit in their representation in the memory). Let the decoding homomorphism Dec map them to the same value Sem_{SL}: the number zero.

This compiler would intuitively be considered as correct. Indeed, diagram 3 commutes, and the compiler is correct according the definition of Morris. There is is no encoding homomorphism Enc that makes diagram 2 commutative, so according to the definition of Thatcher the compiler would be incorrect. Note moreover that there is there is no isomorphism between Sem_{SL} and Sem_{TL}.

Compilers resembling the one above were made in the seventies, an example is the CDC cyber. It had two representations for the number zero (positive and negative zero). Negation of a number (a string of bits) was very simple: replace each 1 by a 0, and each 0 by a 1. This number representation system was called 'one's complement'. Later computers (e.g. the IBM 360) used another system ('two's complement') which has only one representation for zero.
□

This discussion shows that the notion 'correct compiler' is not formalized by diagram 2, but by diagram 3, so there has to be a decoding homomorphism.

2.3 The correctness notion of Morris

In sect. 2.2 it appeared that the definition of Morris was the correct one. But what about the suggestion of Thatcher et. al. that the diagram in Morris' article was

not the intended one? Indeed, all the technical work in Morris' article is about the encoding function *Enc* from source semantics to target semantics. However, he gives explicitly his argument: 'It proves more convenient to define an "encoding" function $Enc: Sem_{SL} \rightarrow Sem_{TL}$ than one in the opposite direction; it will be necessary to prove as a final step in proving the correctness to show that *Enc* has a decoding inverse $Dec: Sem_{TL} \rightarrow Sem_{SL}$ $[\cdots]$' [25, p. 150, +15]. Such a statement is repeated after the definition of *Enc* [p. 150, -2]. These quotations show that the occurrence of the decoding *Dec* in the diagram was on purpose, and not some printing error. As a matter of fact, Morris could prove properties of *Enc* instead of properties *Dec* because a situation like the one from example 2 does not arise in his fragment.

Morris' correctness diagram can also be found in earlier publications by Burstal and Landin ([2] and by Milner and Weyrauch [22]). A variant is proposed by Chirica [3]: he does not use a decoding homomorphism, but family of homomorphisms. The great influence of Thatcher et al. [40] appears from the fact that their approach is followed without discussion in almost all later publications. Significant in this respect is the change from the definition by Mosses in [27] to the one in [28] (i.e. Thatcher's definition which just appeared). The publications which I found with a correctness diagram, have almost all the same diagram as Thatcher. They are Polak [33], Dybjer [6], Royer [35], Tofte [42], and Meijer [20]. The exception is T. Rus, who has his own diagram (see [36]), this is not discussed in this paper.

One might wonder why the original position was so easily abandoned. An explanation could be the influence of category theory. By category theory one is challenged to construct pullbacks, and the encoding homomorphism turns the diagram into such one. In any case, some authors tried (in personal communication) to explain Thatcher's compiler definition with this category-theoretic argument. An additional factor is probably, that the examples discussed in the articles have an injective encoding (this is not made explicit in the articles, though). In general, however, the encoding is no function at all, witness example 2.

3 Translating from view language to database language

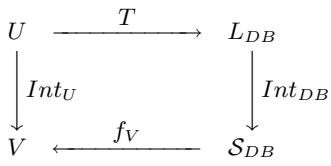
In this section we consider a translation problem that arises in connection with databases. Usually individual users of a data base are not allowed to see all the data, let it be the full structure of the data base. They have only access to their own view, which gives a restricted, and maybe modified, perspective. The view facility allows each user to see the database in its own way. The relation with the original data base is given in the view definition, which maps a state of the database into a view state. Instructions which the user performs on his view have to be translated into instructions of the database itself. For queries this goes without complications. For updates this raises problems because there can be several data base updates that correspond to a given view update. Furthermore,

the update has to be done in such a way that also after further updates the data base remains in correspondence with the view.

Bancilhon and Spyratos [1] study the problems mentioned above, and investigate which translations are allowed. Their first step is to formulate the requirements: updates can be undone, the composition of two updates is an update again, and the translation is a homomorphism. These properties are not formulated with commutative diagrams, but their proposal (their sect. 3) becomes more transparent if we do so.

Definition 1. Let U be an algebra of view updates, where U is closed under the operations composition (;) and right-inverse ($^{-1}$). Let $\mathbf{1}_{DB}$ be the identity on the data base. A *correct translation* is a homomorphism with the following two properties:

1. $T(u; u^{-1}) = \mathbf{1}_{DB}$
2. Diagram 4 commutes.



U	view language algebra, with operations ; and $^{-1}$
T	translation homomorphism
L_{DB}	database update language (restructured as image of U)
Int_U	interpretation of U in the view
Int_{DB}	interpretation of L_{DB} in the data base
V	the views with update operations
f_V	view definition
$\mathcal{S}_{DB}$	data base states with state transformations

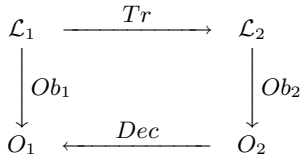
Diagram 4. T is a correct translation of view updates if the diagram commutes (algebraic reformulation of Bancilhon of Spyratos [1, sect. 3])

4 Translating from concurrent language to concurrent language

Shapiro [38] presents a general method to compare languages, and his particular aim is to compare concurrent programming languages. Such languages are difficult to compare because they use different notions of communication and synchronization and different models, and therefore their semantic models are often irreconcilable. The method Shapiro presents, is based on algebraic embeddings. In another paper [26] the method is applied to compare languages defined by machines (Turing machines and finite automata).

The central notion in his approach is 'observable behavior': which in the present context means 'state transitions'. The theme is the behavior of concurrent programming languages with respect to their parallel composition operator. The relation 'have the same observable behavior' defines an equivalence relation on the programming obtained, and this relation can be characterized as the kernel of interpretation function Ob . These behaviors can be compared, and thus give a comparison of the languages. His key definition is given below; he calls 'sound', what we called 'correct', otherwise the description is identical to his proposal.

Definition 2. A translation homomorphism Tr is *correct* if there is a Dec such that diagram 5 commutes.



$\mathcal{L}_1$	algebra defining concurrent language L_1
Tr	translation function
$\mathcal{L}_2$	algebra defining concurrent language L_1
Ob_1	mapping with observable behaviors as kernel
Ob_2	mapping with observable behaviors as kernel
O_1	set of observable behaviors
Dec	decoding function
O_2	set of observable behaviors

Diagram 5. Correct translation of concurrent languages according to Shapiro [38, p. 200]

A stronger notion is what he calls *faithful*: when Dec has an inverse. The meaning assignment Ob is *compositional* if that mapping defines a congruence relation. For these notions he proves some theorems, e.g. conditions when a correct translation is also faithful.

His main theorems state of two properties of concurrent languages that they are preserved under correct translations: 'interference freedom' and 'connection hiding'. For several programming languages it is proven either that they have such a property, or that they do not. Thus this method of comparison primarily yields many negative results: concurrent languages for which no embedding is possible. Positive results are more difficult to obtain because then details of the concurrent behavior have to be considered. That is done for some languages in [39]. Together with embeddings from the literature, it gives a catalogue in which 22 concurrent languages are compared.

5 Translating from logic to logic

There are many logical languages, and between several of them translations have been defined. The purpose of such translations is to investigate the relation between the logics, for instance their relative strength or their relative consistency.

If one considers the method behind such translations, it turns out that (almost) always the algebraic method is used. We shall consider a famous example: Gödel's translation (denoted Gt) of intuitionistic propositional logic into modal logic (see e.g. [4], [8]).

In intuitionistic logic connectives have a constructive interpretation. For instance $\phi \to \psi$ could be read as 'given a proof for ϕ, it can be transformed into a proof for ψ'. The disjunction $\phi \vee \psi$ is read as 'a proof for ϕ is available or a proof for ψ is available'. Since it may be the case that neither a proof for ϕ nor for $\neg\phi$ is available, it is explained why $\phi \vee \neg\phi$ is not a tautology in intuitionistic logic . This explanation has a modal flavor, made explicit in the translation Gt into modal logic S4. In the clauses of the translation the negation does not occur because in intuitionistic logic $\neg\phi$ is defined as an abbreviation for $\phi \to \bot$, where $\bot$ is 'absurdum'.

1. $\mathrm{Gt}(p) = \Box p$, for p an atom
2. $\mathrm{Gt}(\phi \vee \psi) = Gt(\phi) \vee Gt(\psi)$
3. $\mathrm{Gt}(\phi \wedge \psi) = Gt(\phi) \wedge Gt(\psi)$
4. $\mathrm{Gt}(\bot) = \bot$
5. $\mathrm{Gt}(\phi \to \psi) = \Box[Gt(\phi) \to Gt(\psi)]$

Note that the translation of $p \vee \neg p$ is $\Box p \vee \Box \neg \Box p$ (which is not a tautology in modal logic).

It is not difficult to formulate this translation in an explicit algebraic format. We introduce an algebra $\mathcal{IL}$ for the syntax of intuitionistic logic. This algebra has has for instance the operator $IL_\vee$ which puts $\vee$ between its two input arguments:

$IL_\vee(\phi, \psi) = \phi \vee \psi$, where ϕ and ψ are elements of IL.

The operators $IL_\wedge$ and $IL_\to$ are defined likewise. The generators of $\mathcal{IL}$ are $\bot, p, q \ldots$

For modal logic we do the same. Its syntactic algebra M has the operators $M_\to$, $M_\wedge$, $M_\vee$, and $M_\Box$, and the generators $\bot, p, q \ldots$. The operator $M_\Box$, for instance, is defined by

$ML_\Box(\phi) = \Box(\phi)$, where ϕ is an element of $\mathcal{M}$.

The translation Gt now becomes a a homomorhism from $\mathcal{IL}$ to an algebra $\mathcal{M}'$ derived from $\mathcal{M}$. The operators in this derived algebra are $M_\to$, $M_\wedge$ and $M_\vee$ from the original algebra $\mathcal{M}$, and a new one, viz. the polynomially defined operator $ML_\Box(ML_\to(X, Y))$. Its effect is by definition:

$ML_\Box(ML_\to(X, Y))(\phi, \psi) = \Box[\phi \to \psi]$.

Of course, the operator $IL_\vee$ corresponds in the translation with $M_\vee$ and $IL_\wedge$ with $M_\wedge$. Furthermore, $IL_\to$ correponds with $ML_\Box(ML_\to(X, Y))$. Finally, the generators $\bot, p, q \ldots$ of $\mathcal{IL}$ correspond with $\bot$, $\Box p$, $\Box q, \ldots$ respectively, which are the generators of $\mathcal{M}'$.

The method of translating exemplified by Gt, viz. the algebraic method, is the standard method in the field of logic: the definition of translation follows the clauses of the grammar of the source language logic, and for each clause the translation is given by a (possible compound) expression in the target logic. A large number of translations between logics is collected by Epstein [8, Chapter

10: 'Translations between Logic'. pp. 289-314]. Almost all of them are homomorphisms (there they are called 'grammatical translations'), and the few that are not, are also in other respects deviant [p. 313]. It would be interesting to investigate the semantic (model-theoretic) counterparts of such non homomorphic translations.

6 Translating from natural language to natural language

Many methods have been proposed for translating from one natural language to another. The Rosetta project of the Philips Research laboratories (Eindhoven, the Netherlands) has used one that is in that field very special: the compositional (algebraic) method. The syntax of the source language is organized as an algebra, the syntax of the target algebra is a similar algebra, and translating is an *iso*morphism. Their approach is illustrated by the following simplified example.

Consider sentence (1), which has (2) as translation in Dutch.

(1) Peter does not sing.
(2) Peter zingt niet.

The sentences have different syntactic structures: in English there is an auxiliary verb (*do*) that has no counterpart in the Dutch sentence. If one would design for each language separately context free rules producing the respective sentences, then the grammars would not be isomorphic. Nevertheless, in Rosetta the sentences are generated by isomorphic algebras, and below it will be explained how this is done.

The generators of an algebra E for this fragment of English are *Peter* and *to sing*. For Dutch the corresponding generators are *Peter* and *zingen*. E has an operator $R_{E,1}$ that produces from the two generators sentence (3), and $R_{D,1}$ produces likewise (4).

(3) Peter sings.
(4) Peter zingt.

Furthermore there is an operator $R_{E,2}$ that takes as input a sentence and yields its negation. This is not a straightforward rule because the rule has to find the finite verb, move it to another position, and insert *does* and *not*. The Dutch rule $R_{D,2}$ is simpler. So we have:

(5) $R_{E,2}(R_{E,1}(\text{Peter, to sing})) = \text{Peter does not sing.}$
(6) $R_{D,2}(R_{D,1}(\text{Peter, zingen})) = \text{Peter zingt niet.}$

The left hand sides of (5) and (6) describe how the sentence is formed. In (5) it says that $R_{E,1}$ is applied to two generators (*Peter, to sing*), and next $R_{E,2}$ is applied to the result. In algebra the left hand side of (5) is called a 'term', so a term represents a derivation of an expression. The terms corresponding with an algebra A form an algebra themselves, called the term algebra, denoted as T_A.

As one sees, the terms (derivations) in (5) and (6) are isomorphic. This is also the case for the (large) fragments described in the Rosetta system. The isomorphism became possible by adopting the following points of view:

- The algebras are designed, and not discovered as (innate) properties of the mind. The latter is the opinion of a prominent tradition in linguistics.
- The design of the algebras is guided by semantic insights: a syntactic operator corresponds with a meaning operation. This aspect constitutes a difference with many grammatical models in the linguistic tradition or in computational linguistics. In sect. 7 more information will be given about meanings for natural languages.
- Operators are powerful, they do more than just concatenation. They do not necessarily correspond with context free rules.
- Algebras for different languages are tuned. The algebra for a language is not necessarily the algebra that would be designed for the language when considered in isolation: sometimes a decision for one language is influenced by phenomena in the other language.

There are two properties of natural languages that cause differences in algebraic respects with the framework described in other sections.

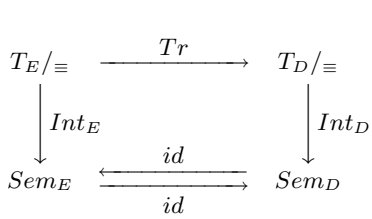
The first is that natural languages are ambiguous. The algebra for a language is designed in such a way that an expression which has two different meanings, can be formed in different ways. It may be formed from different generators, or using different operators (or both). So, in algebraic terminology, two different terms may represent the same expression of the language. Hence the algebra for the source language is not an initial algebra. Since differences in the way of formation of an expression may correspond with differences in translation, the translation homomorphism is not based on the algebra for the source language, but on the term algebra that corresponds with that syntactic algebra. This is by its nature an initial algebra. For the target language the same argumentation applies, so the range of the translation is the term algebra which corresponds with the algebra for the target language.

A second point is that natural languages have synonymous expressions; so one expression may have several equivalent translations. Rosetta aims at obtaining all possible translations and (distinctly from most translation systems) does not select, by some criterion, one of those. This situation does not only arise for words, but also for operators: one construction in the source language can sometimes be translated by several constructions in the target language. Furthermore, different expressions may have the same translation. So there is a many-many correspondence between source and target language. For these reasons, the translation is defined between sets of expressions. In algebraic terminology the situation is as follows. The relation 'are translation equivalent' is in the system a congruence relation on (sub)expressions. This congruence induces a quotient algebra for each of the term algebras, and the translation is defined between these quotient algebras. Due to this quotient construction, the translation homomorphism becomes an isomorphism. That the translation relation is a congruence relation is partially due to translation properties of natural language, but also due to the design of the algebra.

6.1 Correctness

A translation from one natural language into another should to be correct. And by correctness is of course understood that the original and the translation have the same meaning. Since natural languages have an infinite number of sentences, correctness of translation is a property on an infinite set. The algebraic method reduces this to a finite property: if the translations of generators and of operators are correct, the correctness for all sentences follows. Of the Rosetta system it is assumed, based on intuitions about translations, that the generators and operators are translated correctly. From that, the correctness of the whole system follows. This guarantee is something that other translation systems do not have, and is one of the advantages of the algebraic method. In proving compiler correctness the same situation arises, and the same guarantee can be given.

The algebraic structure of Rosetta is described extensively in chapter 19 of [34]. It is summarized in diagram (6).



T_X	term algebra for X, where X=E(nglish) or X=D(utch)
Tr	translation isomorphism
$\equiv$	congruence relation 'translation equivalent'
Int_X	interpretation for X
Sem_X	meanings for X ; so $Sem_E = Sem_D$

Diagram 6. The algebraic structure of Rosetta for translating from English to Dutch. The translation is correct if the diagram commutes in both directions (see [34, ch. 19])

7 Translating from natural language to logic

7.1 Montague grammar

Semantics of natural language is traditionally studied in the field of philosophy of language. Often meanings of natural language expressions are represented in some logic. For long, say until 1975, in all articles it was more or less stipulated which formula was the correct meaning representation of a given sentence (its 'logical form'). This situation has been characterized as follows: it seemed that a 'bilingual logician', who knew logic and who knew natural language, had provided the formula. An opinion often heard (the 'misleading form thesis') was that there exists a great difference between the sentence and its logical representation. Therefore it was proposed to design for certain purposes a 'purified natural' language. So natural language and logical languages were two worlds, with only loose connections.

A radical change in this situation was brought by Richard Montague, a mathematical logician. He developed a method to relate natural language and logic

in a systematic way, presented a semantical interesting fragment of English and provided it with a model-theoretic interpretation through a systematically translation into logic (see [24]). It became, for the first time in history, possible to calculate which meaning is associated with a given sentence, and to make predictions concerning meanings of sentences.

His method was presented in [23], and it is the same algebraic method followed in the other sections of this paper. The syntax of the natural language is a many sorted algebra, and meaning assignment is a homomorphic translation into a logical language. The domain of this homomorphism is not the syntactic algebra itself, but the corresponding term algebra (the algebra of derivations).. Different readings correspond with different ways of production, so with different terms.

The method of Montague grammar is illustrated by the simplified treatment of sentence (1).

(1) John and Mary walk

The syntactic algebra has three generators: *John* and *Mary* of sort *PN* (Proper Name), and *walk* of sort *V* (verb). Other sorts are *NP* (Noun Phrase) and *S* (Sentence). The operators are (for R_1 and R_2, see sect.n 6):

1. R_1: $NP \times V \rightarrow S$, where $R_1(\alpha, \beta) = \alpha\ \beta$
2. R_3: $PN \times PN \rightarrow NP$, where $R_3(\alpha, \beta) = \alpha\ and\ \beta$

So the production of (1) is described by the term:

(2) $R_1(R_3(John,\ Mary),\ walk)$

In Montague's original paper, sentences are translated into intensional logic. That is a higher order modal logic with lambda abstraction. For simplicity, we translate here into extensional predicate logic, enriched with lambda abstraction. The logic has one predicate: *WALK* , and two constants: j and m. The proper names translate into the corresponding constants, and the verb in the corresponding predicate. So the translation Tr of the generators is:

$$Tr(John) = j,\ \ Tr(Mary) = m,\ \ \text{and } Tr(walk) = WALK$$

The operators corresponding with R_1 and R_3 are respectively:

1. T_1: $Bool^{Pred} \times Pred \rightarrow Bool$, where $T_1(\gamma, \delta) = \gamma(\delta)$
2. T_3: $Indiv \times Indiv \rightarrow Bool^{Pred}$, where $T_3(\alpha, \beta) = \lambda P[P(\alpha) \wedge P(\beta)]$

So the translation of *John and Mary* is:

(3) $\lambda P\,[P(j) \wedge P(m)]$

And of the sentence *John and Mary walk*:

(4) $\lambda P\,[P(j) \wedge P(m)](WALK)$

This can be reduced (by lambda conversion) to:

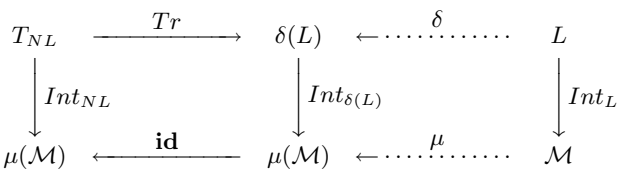
(5) $WALK\ (j) \wedge WALK\ (m)$

A significant point in the above example is the translation of *PN* -conjunction (R_3): it is an operator (T_3) that is defined by means of a polynomial expression. In larger fragments that situation would arise frequently: logic has few operators and constants, whereas natural language needs a lot. So the algebra NL for natural language is not similar with the algebra L for logic. New operators and constants are defined by means of polynomial expressions, and thus within L a reconstruction L' is made of T_{NL}. So T_{NL} is translated onto a polynomially derived algebra L'. Then the interpretation of L determines a unique interpretation of L'. This expressed in:

Theorem 1. Montague[23, p. 225] Let L be an algebra (for logic), $\mathcal{I}$ a homomorphism from L to some algebra $\mathcal{M}$, and let L' be an algebra obtained from L by replacing its operations by polynomially defined operations. Then there is a unique algebra $\mathcal{M}'$ such that there is a homomorphism $\mathcal{I}'$ from L' to $\mathcal{M}'$, where $\mathcal{I}'(a) = \mathcal{I}(a)$ whenever $\mathcal{I}'(a)$ is defined.

This theorem is the background of the following definition of a Montague grammar. The algebraic structure of a Montague grammar is presented in diagram (7).

Definition 3. A Montague grammar consists of a syntactic algebra NL, a logical algebra L, a polynomial derivor δ and a homomorphism from T_{NL} to $\delta(L)$.



NL	algebra for Natural Language	Int_X	interpretation of X
T_{NL}	terms over the algebra NL	μ	model-theoretic counterpart of δ
Tr	translation homomorphism	$\mathcal{M}$	model for L
L	algebra of logic	$\mu(\mathcal{M})$	induced model for $\delta(L)$
δ	derivor which restructures L	**id**	identity mapping
$\delta(L)$	derived logical algebra		

Diagram 7. The algebraic structure of Montague grammar (see [23] sect. 5, and [13])

7.2 Correctness

Of course, the meaning assignment is not an arbitrarily chosen one: it has to yield the 'correct' meaning. One might expect that this means that a meaning assignment has to capture our intuitions concerning meanings of phrases. Indeed, this was the case for the meaning assigned to *John and Mary walk*. For certain

types of expressions, and for simple sentences, one might base formal meanings directly on intuitions, but in many cases it becomes problematic. The intuition may point in the wrong direction, or there might be no intuition at all, especially for subexpressions of sentences. For instance, the meaning of *only* cannot be something like 'there is precisely one', because it does not only occur in phrases like *only John* but also in *only John and Mary* and *only man*.

The solution is to require meanings to formalize intuitions about entailment relations between sentences. A classical case from Montague best known article [24] is: sentence (6) entails (8), whereas (7) does not entail (8).

(6) John finds a unicorn
(7) John seeks a unicorn
(8) There exists a unicorn

A newer, intricate example is from Groenendijk en Stokhof [11]: from sentences (9) and (10) it follows that (11).

(9) John knows whether Mary comes.
(10) Mary does not come.
(11) John knows that Mary does not come.

This example illustrates again that intuitions concerning meanings of natural language expressions are not always available: what would be the intuition about *whether Mary comes*, or about *that Mary comes*? Based upon intuitions concerning *meaning entailments*, model-theoretic interpretations have to be defined that can account for them.

Montague grammar traditionally is a form of possible world semantics. Sentences are interpreted as sets of possible worlds (with the moments of time as parameter), and entailment between two sentences corresponds with set inclusion (for the same parameter value). For instance, for every moment of time, the set of worlds in which (6) is true, forms a subset of the set for which (8) is true. This is not the case for (7) and (8). The examples with *know* require a formalization of the entailment relation that is more complex.

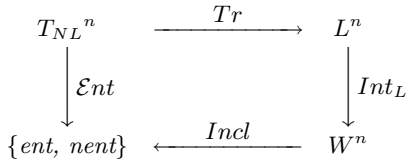
The principle behind this heuristics is expressed in a famous quotation by Lewis [18]:

> In order to say what a meaning is, we may first ask what a meaning does, and then find something that does that.

The entailment perspective on correctness is known in Montague grammar, but its formalization, as the commutative diagram (8) is new.

8 From programming language to logic

Translations from programming language to logic could be used to assign meanings to the programs, but usually the intention is to prove something about a program, for instance its correctness or termination behavior. A well known



$T_{NL}{}^n$	n-tuples of terms for natural language	Int_L	interpretation of L
Tr	translation homomorphism	W^n	n-tuples of meanings: functions from time points to sets of possible worlds
L^n	n-tuples of formulas from logic		
$\mathcal{E}nt$	intuitions concerning entailment (*ent*) and non-entailment (*nent*)	$Incl$	model-theoretic formalization of entailment (for $n = 2$ this is $\subseteq$)

Diagram 8. Tr gives a correct translation of natural language sentences into logic if the diagram commutes.

method to make such a link between programs and logic is the Floyd-Hoare approach. Since the Floyd-Hoare approach inductively follows the structure of the text, one might be tempted to consider it as describing a compositional translation into logic. As will be explained , this is not the case.

Consider, as an example, Hoare's assignment rule.

(1) $[t/x]P\{x{:=}\,t\}P$

Here P is some statement about the values of identifiers. By $[t/x]P$ is understood the statement obtained from P by substituting t for all occurrences of x. The rule says the following. Suppose P has to hold after the assignment, then the statement$[t/x]P$ has to hold before the assignment. An example. Suppose $t \equiv y + 1$, and $x > 9$ has to hold after the assignment, then $y + 1 > 9$ should hold before the assignment, i.e. $y > 8$.

The first point is that the displayed expression is not a formula from logic, but a proof rule. So it is something of a different nature. If the logic is extended with λ operators, we might use

(2) $\lambda P[[t/x]P]$

as denoting a predicate transformer representing the meaning of the assignment. However then a second problem arises. The substitution operator does not belong to the logic itself, but to the meta language used to talk about formulas and manipulations on them. In order to let (2) denote a meaning, the substitution has to be given a semantic interpretation. It changes the state with respect to which the subformula in its scope is interpreted, viz. the state which differs from the current one in that the value assigned to x equals t. Such an interpretation is given by Janssen and van EmdeBoas [17, 16].

It is interesting to note that the same point arises in a different context, viz. the algebraic verification of compilers. By Müller-Olm [29, pp. 30-32] a semantic interpretation of substitution is given, which is completely comparable.

A related problem arises with the rules for procedures with parameters. There are several mechanisms for parameter passing; e.g. call by reference, call by value, and call by name. The last one is defined by means of syntactic substitution! In a compositional approach one would like to obtain the meaning of the entire construction (procedure call with given parameter) by combining the meaning of the procedure name with the meaning of the parameter. A remark expressing this, is from Tennent in a discussion [30] 'Your first two semantics are not 'denotational' ... because the meaning of the procedure call constructs is not defined in terms of the meanings of its components'. Such a compositional analysis is given by Hung and Zucker [12]. They present a treatment in which all those mechanisms naturally find their place, each parameter mechanism corresponds with a different meanings for parameter.

Only some of the Floyd-Hoare rules can be seen (by suitable extension of the logic with λ-operators etc.) as describing meanings. Most can by their nature be considered only as proof rules. Hence the method of Floyd-Hoare is not a translation into logic.

9 Towards a general theory of translation

In previous sections it was shown that translations arise between several kinds of languages. We have seen that in most fields the algebraic method was put forward, often independently of the proposals in other fields. The publications in the different fields discuss the same issues and use related notions. So, there seems to be a common basis for a general theory of translation. Below a first, small step will be made.

We may start with a principle for translating that can be seen as the philosophical background for the algebraic approach:

> **The principle of compositionality of translation**
> The translation of a compound expression is a function of the translations of its parts and of the rule by which the parts are combined.

The first formulation of this principle was in a publication concerning the machine translation project Eurotra, but the idea behind the principle can be found in older publications. A stronger (symmetric) form of the principle is the leading principle of the machine translation project Rosetta [34]. The principle is inspired by Frege's well known principle of compositionality of meaning. The formulation given above mirrors the formulation of Frege's principle in [32, p. 318].

The principle of compositionality of translation can be formalized by requiring that source and target language are algebras SL and TL respectively, and that translating is a homomorphism between the term algebras T_{SL} to T_{TL}. However, for a practical reason, the definition below has more components. When two languages are given, they usually have their own internal structure and differ that much, that they do not have a similar syntax. Therefore the range of the translation function has to be an algebra that is constructed by means of

polynomial operators available in the target algebra. This leads to the following definition.

Definition 4. Let the source language be defined by the algebra SL , and the target language by an algebra TL. Let δ be a polynomial derivor that transforms TL in an algebra similar to SL. Then a *compositional translation* from source language to target language is a homomorphism from the term algebra T_{SL} to the term algebra $T_{\delta(TL)}$. The translation is *correct* if there is a mapping $Decode$: $Sem_{TL} \rightarrow Sem_{SL}$ such that the restriction Dec of $Decode$ to $Sem_{\delta(TL)}$ is a homomorphism that makes the leftmost square in diagram (9) commute. □

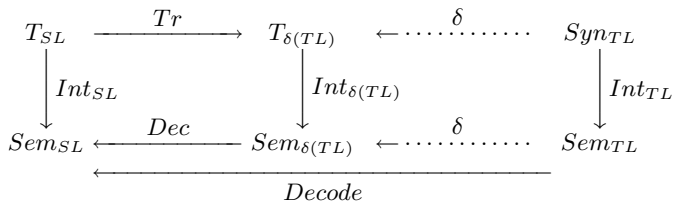


Diagram 9. A general framework for compositional translation

Below we consider the components of this diagram and their relation with the articles we have discussed before.

- T_{SL}
  The source language is an algebra SL. If the expressions of the source language are ambiguous, SL is not suitable as domain of the translation homomorphism. Therefore the term algebra T_{SL} is used as domain for the translation homomorphism. Such ambiguities arise for natural languages (see sect. 6 and 7). If the (sub)expressions of a fragment are not ambiguous, then the term algebra is isomorphic with the original algebra, and the original algebra can be used. This situation arises for logic, and for the examples used in articles about compiler construction. Therefore most authors use the original algebra as domain of the translation, and not the term algebra.
- $T_{\delta(TL)}$ and $Sem_{\delta(TL)}$
  The image of the translation homomorphism has to be an algebra similar to the source language algebra (otherwise the translation cannot be a homomorphism). Therefore a reconstruction of T_{SL} has to be made. The term 'embedding' from logic reflects this aspect, and the term 'reconstruction' is used frequently in [37]. Related diagrams with derivors can be found in [42]. Other authors on algebraic compiler construction do not mention this aspect explicitly, although they proceed in the same way.
- δ
  The new operators needed to form a reconstruction of TL are obtained by polynomials. In the field of natural language this idea is introduced by [24].

The role of polynomials is mostly not explicit in the field of compiler construction, but polynomials are frequently used there. In translations of logics, polynomial translations are standard, but also non-polynomial translations are used sometimes, see sect. 5. In Rosetta (sect.6) there is, due to the design of the algebras, a direct correspondence of operators; but if one investigates the details of the operators, it is possible to view them as polynomials as well.

– *Dec* and *Decode*
If *Dec* exists, then it is unique, because T_{SL} is an initial algebra and the diagram has to commute. Then the image of $x \in Sem_{\delta(TL)}$ can be defined as $X = Int_{SL}(Tr^{-1}({Int_{TL}}^{-1}(x)))$; it only has to be checked that X is a singleton. However, in general, it is difficult to say what X is, because $Sem_{\delta(TL)}$ is not independently given, but defined indirectly (viz. by means of the translation and interpretation). Therefore, there is no information whether for $x \in Sem_{TL}$ also $x \in Sem_{\delta(TL)}$. This explains why usually not *Dec* is defined, but *Decode*; a function with the original meanings as domain; *Dec* is then its restriction to $Sem_{\delta(TL)}$. The ideal that there is an isomorphism between source meanings and target meanings (see sect. 2) can be reached by switching to a quotient algebra: $Sem_{\delta(TL)}/_{Ker(Dec)}$. Without this abstraction *Dec* will be an isomorphism only the exceptional case where the algebras are designed with this purpose (Rosetta, sect. 6).

Definition (4), the structure in diagram (9) and the comments given above, are just the first steps toward a general framework for translating. The following points require further research, and probably other issues as well.

1. Other translations
Although this study brings together a lot of algebraic translations, there certainly are more. The work on 'institutions' (connections between specification formalisms) deals with a related subject [9]. A further comparison may give rise to other questions and answers concerning algebraic translation.
2. Algebra
In all publications concerning programming languages many sorted algebras are used. For Rosetta a one sorted algebra is used, and this also is the case in Montague grammar (see [13] for a many sorted version). However, for applications to natural language order sorted algebras seem most appropriate, and maybe this is also the case for programming languages (cf. [10]).
3. Homomorphism
Most publications follow the standard definition of a homomorphism for many sorted algebras. [36] argues for generalized homomorphisms: mappings which may change the signature (the sort structure). [34, p. 393] gives another generalization: homomorphisms which have not only elements in their range, but also operators. These homomorphisms may not only map two elements to one image, but also two operators.
4. Tools
In projects which deal with larger fragments of language, tools are needed in order to perform all the tasks in an algebraic way. Some of the publications

mentioned in this article describe such projects which have developed tools: [34, 37, 42], and [29].

5. Properties
   Some of the papers discussed here, are of a theoretical nature, and prove properties about the framework, e.g. [38] and [34, ch. 19]. These results might find their place in a coherent framework.

10 Conclusion

In this article we have seen many examples of translations, and sketched a common, algebraic, framework. Even the notion 'correct translation' turned out to be closely related in all fields. Inspired by this unity, I conclude with a quotation from 'Universal Grammar' ([23, p. 313], reprinted in [41, p. 222]), in which I made two adaptations (indicated in italics):

> There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians *and computer scientists*; indeed I consider it possible to comprehend *all these* kinds of languages within a single natural and precise mathematical theory.

Acknowledgments

I thank the following persons for their useful remarks or assistance during the preparation of this article: Mark van de Brand, Peter van Emde Boas, Dick Grune, Lex Hendrix, Dick de Jongh, Karen Kwast, Anton Nijholt, Teodor Rus, Guiseppe Scollo, Martin Steffens, and Albert Visser. I especially thank Willem Paul de Roever for the opportunity to give a contribution to the workshop 'compositionality'.

References

1. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on data bases*, 6:557–575, 1981.
2. R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, chapter 2, pages 17–43. American Elsevier, New York, 1969.
3. L.M. Chirica. *Contributions to compiler correctness.* PhD thesis, Dept. of Computer Science, University of California, Los Angeles, 1976. report UCLA-ENG-7697.
4. D. van Dalen. Intuitionistic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of philosphical logic. Vol III. Alternatives to classical logic*, number 166 in Synthese Library, chapter 4, pages 225–339. Reidel. Dodrecht, 1986.
5. D. Davidson and G. Harman, editors. *Semantics of natural language.* Number 40 in Synthese library. Reidel, Dordrecht, 1972.

6. P. Dybjer. Using domain algebras to prove the correctness of a compiler. In K. Mehlhorn, editor, *STACS 85, 2nd annual symposium on theoretical aspects of computer science, Saarbrücken, 1985*, number 182 in Lecture notes in computer science, pages 98–108, Berlin, 1985. Springer.
7. P. van Emde Boas and T.M.V. Janssen. The impact of Frege's principle of compositionality for the semantics of programming and natural language. In *"Begriffsschrift". Jenaer Frege-Conferenz 1979*, pages 110–129, Jena, 1979. Friedrich-Schiller Universität.
8. R.L. Epstein. *The semantic foundation of logic. vol 1. Propositional logic.* Number 35 in Nijhoff international philosophy series. Nijhoff/Kluwer, Dordrecht, 1990. Second edition published by Oxford University Press, Oxford.
9. J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
10. J.A. Goguen and G. Malcolm. *Algebraic semantics of imperative programs.* Foundations of computing. The MIT Press, Cambridge, Mass., 1996.
11. J. Groenendijk and M. Stokhof. Semantic analysis of wh-complements. *Linguistics and Philosophy*, 5:175–233, 1982.
12. H.-K. Hung and J. I. Zucker. Semantics of pointers, referencing and dereferencing with intensional logic. In *Proc. 6th annual IEEE symposium on Logic in Computer Science*, pages 127–136, Los Almolitos, California, 1991. IEEE Computer Society Press.
13. T.M.V. Janssen. *Foundations and Applications of Montague Grammar: part 1, Philosophy, Framework, Computer Science.* Number 19 in CWI tract. Centre for Mathematics and Computer Science, Amsterdam, 1986.
14. T.M.V. Janssen. Compositionality (with an appendix by B. Partee). In J. van Benthem and A. ter Meulen, editors, *Handbook of logic and language*, chapter 7, pages 417–473. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., 1997.
15. T.M.V. Janssen. Algebraic translations, correctness and algebraic compiler construction. *Journal of theoretical computer science*, 1998. to appear.
16. T.M.V. Janssen and P. van Emde Boas. The expressive power of intensional logic in the semantics of programming languages. In J. Gruska, editor, *Mathematical foundations of computer science 1977 (Proc. 6th. symp. Tatranska Lomnica)*, number 53 in Lecture notes in computer science, pages 303–311, Berlin, 1977. Springer.
17. T.M.V. Janssen and P. van Emde Boas. On the proper treatment of referencing, dereferencing and assignment. In A. Salomaa and M. Steinby, editors, *Automata, languages and programming (Proc. 4th. coll. Turku)*, number 52 in Lecture notes in computer science, pages 282–300, Berlin, 1977. Springer.
18. D. Lewis. General semantics. *Synthese*, 22:18–67, 1970. Reprinted in [5, pp. 169-248] and in [31, pp. 1-50].
19. A. Mazurkiewicz. Parallel recursive program schemes. In J. Becvar, editor, *Mathematical foundations of computer science (4 th. coll., Marianske Lazne)*, number 32 in Lecture notes in computer science, pages 75–87, Berlin, 1975. Springer.
20. E. Meijer. *Calculating compilers.* PhD thesis, Katholieke University Nijmegen, Nijmegen, 1992. ISBN 90-9004673-9.
21. H. J. Milner. Processes: a mathematical model of computing agents. In Rose H.E and J.C. Shepherdson (eds.), editors, *Logic colloquium '73 (Bristol)*, number 80 in Studies in logic and the foundations of mathematics, pages 157–173, Amsterdam, 1975. North Holland.

22. R. Milner and R. Weyrauch. Proving compiler correctness in a mechanized logic. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, chapter 3, pages 51–70. American Elsevier, New York, 1972.
23. R. Montague. Universal grammar. *Theoria*, 36:373–398, 1970. Reprinted in [41, pp. 222-246].
24. R. Montague. The proper treatment of quantification in ordinary English. In K.J.J. Hintikka, J.M.E. Moravcsik, and P. Suppes, editors, *Approaches to natural language, Synthese Library 49*, pages 221–242. Reidel, Dordrecht, 1973. Reprinted in [41, pp. 247-270].
25. F.L. Morris. Advice on structuring compilers and proving them correct. In *Proceedings ACM Symposium on principles of programming languages, Boston, 1973*, pages 144–152. Association for Computing Machinery, 1973.
26. Y. Moscowitz and E. Shapiro. On the structural simplicity of machines and languages. Technical Report CS93-04, Weizman institute, dept. of appl. math. and comp. sc., Rehovot, Israel, 1993. to appear in 'Annals of mathematics and artificial intelligence'.
27. P. Mosses. A constructive approach to compiler correctness. In N.D. Jones, editor, *Semantics-directed compiler generation (Proc. Workshop, Aarhus)*, number 94 in Lecture notes in computer science, pages 189–210. Springer, Berlin, 1980.
28. P. Mosses. A constructive approach to compiler correctness. In J. de Bakker and J. van Leeuwen, editors, *Automata, languages and programming (seventh colloquium, Noordwijkerhout)*, number 85 in Lecture notes in computer science, pages 449–469. Springer, Berlin, 1980.
29. M. Müller-Olm. *Modular Compiler verification*. Number 1283 in Lecture notes in computer science. Springer, Berlin, 1997.
30. E.J. Neuholt, editor. *Formal description of programming language concepts, Proc, IFIP working conference on formal description of programming concepts St. Andrews, Canada 1977*, Amsterdam, 1978.
31. B. Partee, editor. *Montague grammar*. Academic Press, New York, 1976.
32. B. Partee, A. ter Meulen, and R.E. Wall. *Mathematical Methods in Linguistics*. Number 30 in Studies in Linguistics and Philosophy. Kluwer, Dordrecht, 1990.
33. W. Polak. *Compiler specification and verification*. Number 124 in Lecture notes in computer science. Springer, Berlin, 1981.
34. M.T. Rosetta. *Compositional Translation*. The Kluwer International Series in Engineering and Computer Science 230. Kluwer, Dordrecht, 1994. (M.T. Rosetta = {L. Appelo, T. Janssen, F. de Jong, J. Landsbergen (eds.)}).
35. V. Royer. Transformations of denotational semantics in semantics directed compiler generation. In *Proceedings of the SIGPLAN '86 symposium on compiler construction*, SIGPLAN notices Vol. 21, Nr. 7, pages 68–73. Association for Computing Machinery, 1986.
36. T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.
37. T. Rus. Algebraic processing of programming languages. In A. Nijholt, G. Scollo, and R. Steetskamp, editors, *Algebraic methods in language processing AMILP'95*, number 10 in Twente workshop In language technology, pages 1–41, Enschede, 1995. Universiteit Twente.
38. E. Shapiro. Separating concurrent languages with categories of language embeddings. In *Proceedings of the 23 annual symposium on theory of computing (STOC'91, New Orleans)*, pages 198–208. ACM, 1991.

39. E. Shapiro. Embeddings among concurrent programming languages. In W. R Cleaveland, editor, *Proc. third international conference on concurrency theory, Stony Brook 92*, Springer lecture notes in computer science, pages 486–503, Berlin, 1992. Springer.
40. J.W. Thatcher, E.G. Wagner, and J.B. Wright. More on advice on structuring compilers and proving them correct. In H.A. Maurer, editor, *Automata, languages and programming. (Proc. 6th. coll. Graz)*, number 71 in Lecture notes in computer science, Berlin, 1979. Springer.
41. R.H. Thomason. *Formal Philosophy. Selected Papers of Richard Montague*. Yale University Press, New Haven, 1974.
42. M. Tofte. *Compiler generators. What they can do, what they might do, and what they will probably never do.* Number 19 in EATCS monographs on theoretical computer science. Springer, Berlin, 1990.

Compositional Verification of Multi-Agent Systems: a Formal Analysis of Pro-activeness and Reactiveness

Catholijn M. Jonker, Jan Treur

Vrije Universiteit Amsterdam
Department of Mathematics and Computer Science, Artificial Intelligence Group
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
URL: http://www.cs.vu.nl, Email: {jonker,treur}@cs.vu.nl

Abstract
A compositional method is presented for the verification of multi-agent systems. The advantages of the method are the well-structuredness of the proofs and the reusability of parts of these proofs in relation to reuse of components. The method is illustrated for an example multi-agent system, consisting of co-operative information gathering agents. This application of the verification method results in a formal analysis of pro-activeness and reactiveness of agents.

1 Introduction

When designing multi-agent systems, it is often hard to guarantee that the specification of a system that has been designed actually fulfils the needs, i.e., whether it satisfies the design requirements. Especially for critical applications, for example in real-time domains, there is a need to prove that the designed system will have certain properties under certain conditions (assumptions). While developing a proof of such properties, the assumptions that define the bounds within which the system will function properly are generated. For nontrivial examples, verification can be a very complex process, both in the conceptual and computational sense. For these reasons, it is a recent trend in the literature on verification in general to study the use of compositionality and abstraction to structure the process of verification; for example, see (Abadi and Lamport, 1993; Hooman, 1994; Dams, Gerth and Kelb, 1996).

The development of structured modelling frameworks and principled design methods tuned to the specific area of multi-agent systems is currently underway; e.g., (Brazier, Dunin-Keplicz, Jennings and Treur, 1995; Fisher and Wooldridge, 1997; Kinny, Georgeff and Rao, 1996). As part of any mature multi-agent system design method, a verification approach is required. For example, in (Fisher and Wooldridge, 1997) verification is addressed within a temporal belief logic. This verification method does not exploit compositionality within the agents. In the current paper, in Section 3, a compositional verification method for multi-agent systems is introduced. Roughly spoken, the requirements of the whole system are formally verified by deriving them from assumptions that themselves are properties of agents, which in their turn may be derived from assumptions on sub-components of agents, and so on.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 350-380, 1998.
© Springer-Verlag Berlin Heidelberg 1998

The compositional verification method introduced here is illustrated for an example multi-agent system, consisting of two cooperative information gathering agents and the world. For this example multi-agent system, requirements are formulated (both the required *static* and*dynamic* properties), including variants of*pro-activeness* and *reactiveness*. These requirements are formalised in terms of temporal semantics. It is shown how they can be derived from properties of agents and how these agent properties in turn can be derived from properties of the agent components. A compositional system specification is introduced in Section 4. The system specification defines how the system is composed of the two agents and the world and how each agent is composed of four agent components: for own process control, world interaction management, agent interaction management, and an agent specific task (which in this case is classification of objects in the world). The compositional specification itself is expressed in the modelling framework DESIRE, shortly introduced in Section 2. The application of the compositional verification method to the example multi-agent system is presented in Section 5 for the top level of the composition. More details on the lower levels can be found in Sections 6 and 7.

2 Compositional Modelling of Multi-Agent Systems

The example task model described in this paper is specified within the compositional modelling framework DESIRE for multi-agent systems (framework for DEsign and Specification of Interacting REasoning components; cf. (Langevelde, Philipsen and Treur, 1992; Brazier, Dunin-Keplicz, Jennings, Treur, 1995)). In DESIRE, a design consist of knowledge of the following three types:

- process composition,
- knowledge composition,
- the relation between process composition and knowledge composition.

These three types of knowledge are discussed in more detail below.

2.1 Process Composition

Process composition identifies the relevant processes at different levels of (process) abstraction, and describes how a process can be defined in terms of lower level processes.

2.1.1 Processes at Different Levels of Abstraction

Processes can be described at different levels of abstraction; for example, the process of the multi-agent system as a whole, processes defined by individual agents and the external world, and processes defined by task-related components of individual agents.

Specification of a Process

The identified processes are modelled as *components*. For each process the *types of information* required as input and resulting as output are identified as well. This is modelled as *input and output interfaces* of the components.

Specification of Process Abstraction Levels

The identified levels of process abstraction are modelled as *abstraction/specialisation relations* between components at adjacent levels of abstraction: components may be

composed of other components or they may be *primitive*. Primitive components may be either reasoning components (for example based on a knowledge base), or, alternatively, components capable of performing tasks such as calculation, information retrieval, optimisation, et cetera.

The identification of processes at different abstraction levels results in specification of components that can be used as building blocks, and of a specification of the sub-component relation, defining which components are a sub-component of a which other component. The distinction of different process abstraction levels results in process hiding.

2.1.2 Composition of Processes

The way in which processes at one level of abstraction are composed of processes at the adjacent lower abstraction level is called *composition*. This composition of processes is described by the possibilities for *information exchange* between processes (*static view* on the composition), and *task control knowledge* used to control processes and information exchange (*dynamic view* on the composition).

Information Exchange

Knowledge of information exchange defines which types of information can be transferred between components and the *information links* by which this can be achieved. Two types of information links are distinguished: *private* information links and *mediating* information links. For a given parent component, a *private information link* relates output of one of its components to input of another, by specifying which truth value of a specific output atom is linked with which truth value of a specific input atom. Atoms can be renamed: each component can be specified in its own language, independent of other components. In a similar manner *mediating information links* transfer information from the input interface of the parent component to the input interface of one of its components, or from the output interface of one of its components to the output interface of the parent component itself. Mediating links specify the relation between the information at two adjacent abstraction levels in the process composition.

Task Control Knowledge

Components may be activated sequentially or they may be continually capable of processing new input as soon as it arrives (*awake*). The same holds for information links: information links may be explicitly activated or they may be awake. *Task control knowledge* specifies under which conditions which components and information links are active (or made awake). Evaluation criteria, expressed in terms of the evaluation of the results (success or failure), provide a means to guide further processing.

2.2 Knowledge Composition

Knowledge composition identifies the knowledge structures at different levels of (knowledge) abstraction, and describes how a knowledge structure can be defined in terms of lower level knowledge structures. The knowledge abstraction levels may correspond to the process abstraction levels, but this is often not the case.

2.2.1 Knowledge Structures at Different Abstraction Levels

The two main structures used as building blocks to model knowledge are: *information types* and *knowledge bases*. Knowledge structures can be identified and described at different levels of abstraction. The resulting levels of knowledge abstraction can be distinguished for both information types and knowledge bases.

Information Types

An information type defines an ontology (lexicon, vocabulary) to describe objects or terms, their sorts, and the relations or functions that can be defined on these objects. Information types can logically be represented as signatures in *order-sorted predicate logic*.

Knowledge Bases

A knowledge base defines a part of the knowledge that is used in one or more of the processes. Knowledge is represented logically by rules in order-sorted predicate logic.

Knowledge bases use ontologies defined in information types. Which information types are used in a knowledge base defines a relation between information types and knowledge bases.

2.2.2 Composition of Knowledge Structures

Information types can be composed of more specific information types, following the principle of compositionality discussed above. Similarly, knowledge bases can be composed of more specific knowledge bases. The compositional structure is based on the different levels of knowledge abstraction that are distinguished, and results in information and knowledge hiding.

2.3 Relation Between Process and Knowledge Composition

Each process in a process composition uses knowledge structures. Which knowledge structures are used for which processes is defined by the relation between process composition and knowledge composition.

The semantics of the modelling language are based on temporal logic (cf., Brazier, Treur, Wijngaards and Willems, 1996). Design is supported by graphical tools within the DESIRE software environment. Translation into an operational system is straightforward; the software environment includes implementation generators with which specifications can be translated into executable code. DESIRE has been successfully applied to design both single agent and multi-agent systems.

3 Compositional Verification

The purpose of verification is to prove that, under a certain set of assumptions, a system will adhere to a certain set of properties, for example the design requirements. In our approach, this is done by a mathematical proof (i.e., a proof in the form mathematicians are accustomed to do) that the specification of the system together with the assumptions implies the properties that it needs to fulfil. In this sense verification leads to a formal analysis of relations between properties and assumptions.

3.1 The Compositional Verification Method

A compositional multi-agent system can be viewed at different levels of abstraction. Viewed from the top level, denoted by L_0, the complete system is one component S, with interfaces, whereas internal information and processes are hidden (information and process hiding). At the next lower level of abstraction, the system component S can be viewed as a composition of agents and the world, information links between them, and task control. Each agent A is composed of its sub-components, and so on. The compositional verification method takes this compositional structure into account.

For composed components two types of properties are recognised: behavioural and environmental properties. A behavioural property is a property on the output of the component. Behavioural properties can be conditional, or unconditional. A behavioural property is conditional if the statements about the output of the component hold under the assumption that some specific conditions hold for its input. For example, a conditional behavioural property of a diagnostic agent could be *conditional conclusion correctness of an agent* (i.e., if the observation information needed for diagnosis, which is input of the agent, is correct, then all diagnostic output of the agent is correct), whereas the corresponding unconditional property would be *conclusion correctness of an agent* (i.e., all diagnostic output of the agent is correct). An environmental property is a property on the input of the component (possibly referring to certain conditions on the output).

The primitive components can be verified using more traditional verification methods such as described in (Treur and Willems, 1994; Leemans, Treur and Willems, 1995). Verification of a composed component is done using properties of the sub-components it embeds and the task control knowledge, and environmental properties of the component (depending on the rest of the system, including the world). This introduces a form of compositionality in the verification process: given a set of environmental properties the proof that a certain component adheres to a set of behavioural properties depends on the (assumed) properties of its sub-components, properties of the interactions between those sub-components, and the manner in which they are controlled. The assumptions under which the component functions properly, are the properties to be proven for its sub-components. This implies that properties at different levels of abstraction are involved in the verification process.

Often these properties are not given at the start of the verification process. Actually, the process of verification has two main aims:

- to find the properties
- given the properties, to prove the properties

The verification proofs that connect one abstraction level with the other are compositional in the following manner: any proof relating level i to level i+1 can be combined with any proof relating level i-1 to level i, as long as the same properties at level i are involved. This means, for example, that the whole compositional structure beneath level i can be replaced by a completely different design as long as the same properties at level i are achieved. After such a modification the proof from level i to level i+1 can be reused; only the proof from level i-1 to level i has to be adapted. In this sense the verification method supports reuse of verification proofs.

The compositional verification method can be formulated in more detail as follows:

A. Verifying one Abstraction Level Against the Other

For each abstraction level the following procedure for verification is followed:

1. Determine which properties are of interest (for the higher level).
2. Determine which assumptions (at the lower level) are needed to guarantee these properties, and which environment properties.
3. Prove the properties on the basis of these assumptions, and the environment properties.

B. Verifying a Primitive Component

For primitive knowledge-based components a number of techniques exist in literature, see for example (Treur, Willems 1994; Leemans, Treur, Willems 1995). For primitive non-knowledge-based components, such as databases, or neural networks, or optimisation algorithms, verification techniques can be used that are especially tuned for that type of component.

C. The Overall Verification Process

To verify the complete system

1. Determine the properties that are desired for the whole system.
2. Apply the above procedure **A** iteratively until primitive components are reached. In the iteration the desired properties of abstraction level L_i are either:
   - ¥ those determined in step **A1**, if i = 0, or
   - ¥ the assumptions made for the higher level L_{i-1}, if i > 0
3. Verify the primitive components according to **B**.

The results of verification are:

- Properties and assumptions at the different abstraction levels.
- The logical relations between the properties of different abstraction levels.

Notes:

- both static and dynamic properties and connections between them are covered.
- reuse of verification results is supported (refining an existed verified compositional model by further decomposition, leads to a verification of the refined system in which the verification structure of the original system can be reused).
- process and information hiding limits the complexity of the verification per abstraction level.
- a requirement to apply the compositional verification method described above is the availability of an explicit specification of how the system description at an abstraction level L_i is composed from the descriptions at the lower abstraction level L_{i+1}; the compositional modelling framework DESIRE is an instance of a modelling framework that fulfils this requirement.
- in principle alternative (e.g., bottom-up or mixed) procedures can be formulated as well.

3.2 Semantics Behind the Compositional Verification Method

In principle, verification is always relative to semantics of the system descriptions that are verified. For the compositional verification method, these semantics are based on compositional information states which evolve over time. In this subsection a brief overview of these assumed semantics is given.

An *information state* M of a component D is an assignment of truth values {true, false, unknown} to the set of ground atoms that play a role within D. The compositional structure of D is reflected in the structure of the information state. A formal definition can be found in (Brazier, Treur, Wijngaards and Willems, 1996; Brazier, Eck and Treur, 1996). The set of all possible information states of D is denoted by IS(D).

A *trace* $\mathcal{M}$ of a component D is a sequence of information states $(M^t)_{t \in \mathbf{N}}$ in IS(D). The set of all traces is denoted by $IS(D)^{\mathbf{N}}$, or Traces(D). Given a trace $\mathcal{M}$ of component D, the information state of the input interface of component C at time point t of the component D is denoted by $state_D(\mathcal{M}, t, input(C))$, where C is either D or a sub-component of D. Analogously, $state_D(\mathcal{M}, t, output(C))$, denotes the information state of the output interface of component C at time point t of the component D. Given a trace $\mathcal{M}$ of component D, the task control information state of component C at time point t of the component D is denoted by $state_C(\mathcal{M}, t, tc(C))$, where C is either D or a sub-component of D.

To connect neighbouring levels of abstraction in a verification proof for a DESIRE specification, the following elements can be used:

- the assumptions of the sub-components specified within component D
- the interactions between the sub-components of D and/or the interfaces of D
- the input / output information states of the sub-components of D
- the task control information states of the sub-components of D
- the information states of component D
- the task control information states of component D

4 The Example Multi-Agent Model

The example multi-agent model is composed of three components: two agents A and B and a component W representing the external world, see Figure 1. Each of the agents is able to acquire partial information about the external world (by observation). Each agent s own observations are insufficient to draw conclusions of a desired type, but the combined information of both agents is sufficient. Therefore communication is required to be able to draw conclusions. The agents can communicate their own observation results and requests for observation information of the other agent. This by itself not unrealistic situation is simplified to the following materialised form. The world situation consists of an object that has to be classified. One agent can only observe the bottom view of the object, the other agent the side view. By exchanging and combining observation information they are able to classify the object.

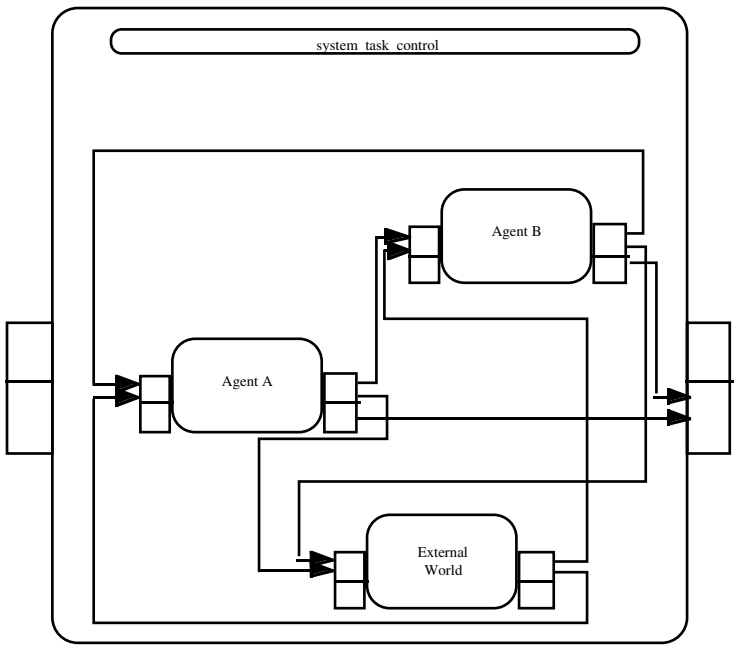


Fig. 1. The example Multi-Agent System

Communication from the agent A to B takes place in the following manner:

- the agent A generates at its output interface a statement of the form:
  to_be_communicated_to(<*type*>, <*atom*>, <*sign*>, B)
- the information is transferred to B; thereby it translated into
  communicated_by(<*type*>, <*atom*>, <*sign*>, A)

In the example <*type*> can be filled with a label request or world_info, <*atom*> is an atom expressing information on the world, and <*sign*>, is one of pos or neg, to indicate truth or falsity.

Interaction between an agent A and the world takes place as follows:

- the agent A generates at its output interface a statement of the form:
  to_be_observed(<*atom*>)
- the information is transferred to W; thereby it is translated into
  to_be_observed_by(<*atom*>, A)
- the world W generates at its output interface a statement of the form:
  observation_result_for(<*atom*>, <*sign*>, A)
- the information is transferred to A; thereby it is translated into
  observation_result(<*atom*>, <*sign*>)

Part of the output of an agent are conclusions about the classification of the object of the form object_type(s); these are transferred to the output of the system.

To be able to perform its tasks, each agent is composed of four components, see Figure 2: three for generic agent tasks (world interaction management, agent interaction management, own proces control), and one for an agent specific task (object classification).

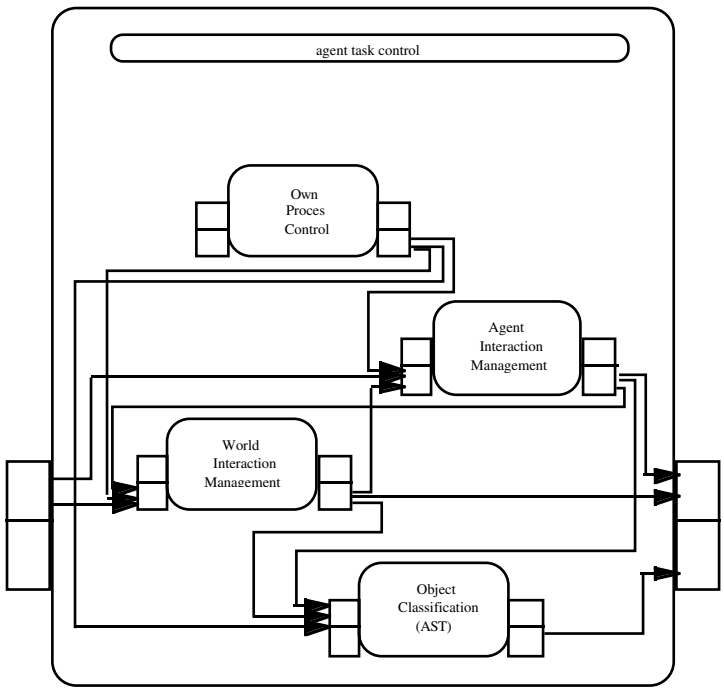


Fig. 2. Composition of an agent

object classification (agent specific task)

This component is able to draw a conclusion if it has input on the two views on the object. The component can reason on the basis of the world knowledge represented in the table depicted in Table 1:

	○	△	□
○	sphere	cone	cylinder
△	cone	tetrahedron	pyramid
□	cylinder	pyramid	cube

Table 1. World knowledge

world interaction management

This component reasons about the manner in which the agent interacts with the world. Here it is decided under which conditions which observations are to be performed in the world.

agent interaction management

This component reasons about the agent s communication with other agents. In this component it is determined when to request which information from the other agent. Another task is to determine when to provide which observation information to the other agent and what to do with the world information received from the other agent.

own process control

This component defines the agent s own characteristics or attitudes. Information on these attitudes can be transferred to other components, to influence the reasoning that takes place there. The agents can differ in their attitudes towards observation and communication: an agent may or may not be *pro-active*, in the sense that it takes the initiative with respect to one or more of:

- performing observations
- communicate its own observation results to the other agent
- ask the other agent for its observation results
- draw conclusions about the classification of the object

Moreover, it may be *reactive* to the other agent in the sense that it responds to a request for observation information:

- by communicating its observation result as soon as they are available
- by starting to observe for the other agent

These agent attitudes are represented explicitly as (meta-)facts in the agent s component own process control. By varying these attitude facts, different variants of agents can be defined. The impact of these explicitly specified characteristics has been specified in the model. For example, if an agent has the attitude that it will always take the initiative to communicate its observation results as soon as they are acquired, then the agent s behaviour should show this, but if the characteristic is not there, then this behaviour should not be present. This requires an adequate interplay between the component own process control and the component agent interaction management within the agent, and adequate knowledge within agent interaction management.

The successfulness of the system depends on the attitudes of the agents. For example, if both agents are pro-active and reactive in all respects, then they can easily come to a conclusion. However, it is also possible that one of the agents is only reactive, and still the other agent comes to a conclusion. Or, an agent that is only pro-active in reasoning and reactive in information acquisition may come to a conclusion due to pro-activeness of the other agent. So, successfulness can be achieved in many ways and depends on subtle interactions between pro-activeness and reactiveness attitudes of both agents. The formal analysis of the example in the following sections provides a detailed picture of these possibilities.

5 Formal Analysis of the Example System: Top Level

In this section the properties of the system as a whole (defined in Section 5.1) are related to properties of the agents and the world (defined in Section 5.2 and 5.3), and their interaction.

5.1 Properties for the Top Level of the System

First, it is determined which properties the system as a whole should satisfy. Considering that the system S is a classification system, it is expected that S produces output of the form object_type(s) for some s. A first requirement is that output generated by the system is correct, i.e., if the system derives object_type(s) for some s, it is true in the world situation. Let the world state (which is assumed static) be denoted by M. In Figure 4 the successfulness property of S is related to other properties of S and assumed properties of the next level. In Figure 3 the correctness property of S is similarly related to other properties. The following property relates the output of the system to the current world state.

Correctness of S

The system S is called *correct* if:

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall t\ \forall s$$
$$[\ \text{state}_S(\mathcal{M}, t, \text{output(S)}) \models \text{object_type(s)} \Rightarrow M \models \text{object_type(s)}\ ]$$
$$\wedge\ [\ \text{state}_S(\mathcal{M}, t, \text{output(S)}) \models \neg\text{object_type(s)} \Rightarrow M \models \neg\text{object_type(s)}\ ]$$

Output information of S is only provided by agents

As can be seen in Figure 1 the only information links connected to the output of S are the information link from agent A to S and the information link from agent B to S. Furthermore, the output interface of S cannot spontaneously change its contents. Therefore, the output information of the system is only provided by agents.

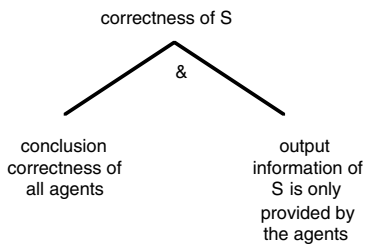


Fig. 3. Correctness of S

Next, the system is required to be successful in generating conclusions: during the process, for each s, at some time point it should either have derived positive output, or negative output:

Successfulness of S

The system S is called *successful* if:

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall s\ \exists t\ \text{state}_S(\mathcal{M}, t, \text{output(S)}) \models \text{object_type(s)})$$
$$\vee\ \exists t\ \text{state}_S(\mathcal{M}, t, \text{output(S)}) \models \neg\text{object_type(s)})$$

To guarantee the adequate information exchange between the different components, the following properties are needed:

Interaction effectiveness

The interaction from agent X to the output interface of system S is called *effective* if, for ϕ either object_type(s) or $\neg$object_type(s):

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall t\ \forall s\ [\text{state}_S(\mathcal{M}, t, \text{output(X)}) \models \phi \Rightarrow \exists t' > t\ \text{state}_S(\mathcal{M}, t', \text{output(S)}) \models \phi]$$

The interaction from the external world W to agent X is called *effective* if:

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall t\ \forall r, \text{sign}\ [\text{state}_S(\mathcal{M}, t, \text{output(W)}) \models \text{observation_result_for(view(X,r), sign, X)} \Rightarrow \exists t' > t\ \text{state}_S(\mathcal{M}, t', \text{input(X)}) \models \text{observation_result(view(X,r),sign)}]$$

The interaction from the agent X to the external world W is called *effective* if:

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall t\ \forall r, \text{sign}\ [\text{state}_S(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_observed (view(X,r))} \Rightarrow \exists t' > t\ \text{state}_S(\mathcal{M}, t', \text{input(W)}) \models \text{to_be_observed_by (view(X,r), X)}]$$

Interaction effectiveness can be proven from the detailed specification of the information links involved and the timely functioning of those information links as specified in the task control of the component containing the information link given in the detailed specification. This property is needed to prove several environment properties of the agents and also to prove successfulness of S. Sometimes it will not be stated explicitly.

Agent provides output information of S

An agent X provides output information of system S if X is conclusion successful and the interaction from X to S is effective.

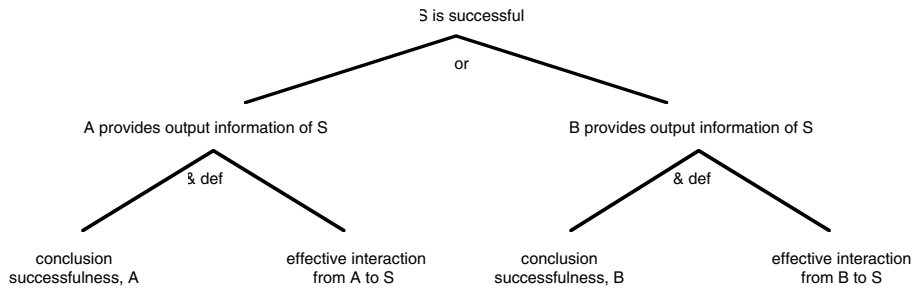


Fig. 4. Successfulness of S

It is undesirable (for a static world situation) that the system changes its mind during the process. Therefore the requirement is chosen that once a conclusion has been derived, this is never revised:

Conservativity of s

The system S is called *conservative* if:

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall t\ \forall s$$
$$[\ \text{state}_S(\mathcal{M}, t, \text{output(S)}) \models \text{object_type(s)} \Rightarrow$$
$$\forall t' > t\ \ \text{state}_S(\mathcal{M}, t', \text{output(S)}) \models \text{object_type(s)}\ ]$$
$$\wedge\ [\ \text{state}_S(\mathcal{M}, t, \text{output(S)}) \models \neg\, \text{object_type(s)} \Rightarrow$$
$$\forall t' > t\ \ \text{state}_S(\mathcal{M}, t', \text{output(S)}) \models \neg\, \text{object_type(s)}\ ]\ )$$

The property of conservativity of S can be proven in a similar way as the other properties of S. The proof has been omitted from this paper.

Communication effectiveness

The communication from agent X to agent Y is called *effective* if:

$$\forall \mathcal{M} \in \text{Traces(S)}\ \forall t\ \forall \text{sign}\ \forall\ \phi$$
$$[\text{state}_S(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_communicated_to(q, } \phi\text{, sign, Y)}$$
$$\Rightarrow \exists t' > t\ \ \text{state}_S(\mathcal{M}, t', \text{input(Y)}) \models \text{communicated_from(q, } \phi\text{, sign, X)}\ ]$$

Communication effectiveness can be proven in the same way as interaction effectiveness. This property is needed to prove environment properties of the agents.

5.2 Properties of the Agents

The required properties of the system have been proven from assumed properties of the components at one level lower. During this proof process these assumptions have been discovered. A number of assumptions are quite straightforward. For example, correctness inherits upward from the agents to the system:

Conclusion correctness of an agent

An agent X is called *conclusion correct* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall s$$
$$[\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \text{object_type(s)} \Rightarrow \text{M} \models \text{object_type(s)}\ ]$$
$$\wedge\ [\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \neg\text{object_type(s)} \Rightarrow \text{M} \models \neg\text{object_type(s)}\ ]$$

This property logically depends on other properties of the agent, input correctness and conditional conclusion correctness, as can be seen for agent A in Figure 5.

Input correctness of an agent

a) An agent X is called *observation input correct* if

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall r$$
$$[\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{observation_result(view(X,r),pos)} \Rightarrow \text{M} \models \text{view(X,r)}\ ]$$
$$\wedge\ \forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall r$$
$$[\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{observation_result(view(X,r),neg)} \Rightarrow \text{M} \models \neg\text{view(X,r)}\ ]$$

b) An agent X is called *communication input correct* if

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall r$$
$$[\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{communicated_from(world_info,view(Y,r), pos,Y)} \Rightarrow \text{M} \models \text{view(Y,r)}\ ]$$
$$\wedge\ \forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall r$$
$$[\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{communicated_from(world_info,view(Y,r), neg,Y)} \Rightarrow \text{M} \models \neg\text{view(Y,r)}\ ]$$

c) An agent X is called *input correct* if X is observation input correct and communication input correct.

Conditional conclusion correctness of an agent
An agent X is called *conditionally conclusion correct* if the following holds: if input correctness of X then conclusion correctness of X.

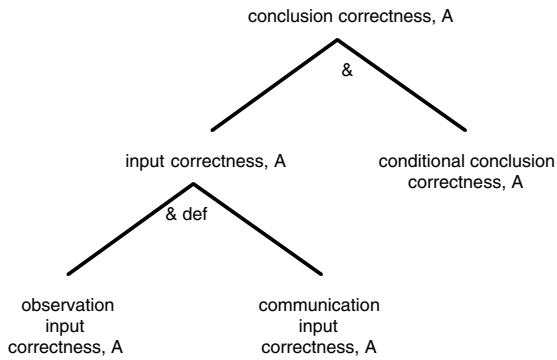


Fig. 5. Conclusion correctness of A

The following property is needed to prove conservativity of S.

Conclusion conservativity of an agent
The agent X is called *conclusion conservative* if:

$$\forall \mathfrak{M} \in \text{Traces(X)}\ \forall t\ \forall s$$
$$[\ \text{state}_X(\mathfrak{M}, t, \text{output(X)}) \models \text{object_type(s)} \Longrightarrow$$
$$\forall t' > t\ \ \text{state}_X(\mathfrak{M}, t', \text{output(X)}) \models \text{object_type(s)}\ ]$$
$$\wedge\ [\ \text{state}_X(\mathfrak{M}, t, \text{output(X)}) \models \neg\, \text{object_type(s)} \Longrightarrow$$
$$\forall t' > t\ \ \text{state}_X(\mathfrak{M}, t', \text{output(X)}) \models \neg\, \text{object_type(s)}\ ]\ )$$

Again, the proof has been omitted from this paper. Successfulness of the system (see Figure 4) depends on successfulness of at least one of the agents.

Conclusion successfulness of an agent
The agent X is called *conclusion successful* if:

$$\forall \mathfrak{M} \in \text{Traces(X)}\ \exists t\ \ \text{state}_X(\mathfrak{M}, t, \text{output(X)}) \models \text{object_type(s)})$$
$$\vee\ \exists t\ \ \text{state}_X(\mathfrak{M}, t, \text{output(X)}) \models \neg \text{object_type(s)})$$

This property can be proven in a number of ways. However, in all proofs the properties information saturation of the agent and conclusion pro-activeness is required, see Figure 6 for the logical relations between properties of agent A that are needed to prove conclusion successfulness of agent A.

Information saturation of an agent

a) The agent X is called *observation info saturating* if:

$\forall \mathcal{M} \in$ Traces(X) $\exists t\ \forall r\ \exists$ sign

state$_X(\mathcal{M}, t, \text{input(X)}) \models$ observation_result(view(X,r), sign)

b) The agent X is called *communicated info saturating* if:

$\forall \mathcal{M} \in$ Traces(X) $\exists t\ \forall r\ \exists$ sign

state$_X(\mathcal{M}, t, \text{input(X)}) \models$ communicated_from(world_info, view(Y,r), sign, Y)

c) The agent X is called *request saturating* if for all agents Y different from X:

$\forall \mathcal{M} \in$ Traces(X) $\exists t\ \forall r$

state$_X(\mathcal{M}, t, \text{input(X)}) \models$ communicated_from(request, view(X,r), pos, Y)

d) The agent X is called *information saturating* if X is observation info saturating and communicated info saturating.

Conclusion pro-activeness

The agent X is called *conclusion pro-active* if for all agents Y different from X:

$\forall \mathcal{M} \in$ Traces(X) $\forall t,t$

$[\forall r,r\ \exists$ sign,sign

state$_X(\mathcal{M}, t, \text{input(X)}) \models$ observation_result(view(X,r), sign) $\wedge$

state$_X(\mathcal{M}, t, \text{input(X)}) \models$ communicated_from(world_info, view(Y,r), sign , Y) $]]$

$\Rightarrow\ \exists$ t >t, t >t $\forall$s [state$_X(\mathcal{M}, t, \text{output(X)}) \models$ object_type(s) $\vee$

state$_X(\mathcal{M}, t, \text{output(X)}) \models \neg$object_type(s)]

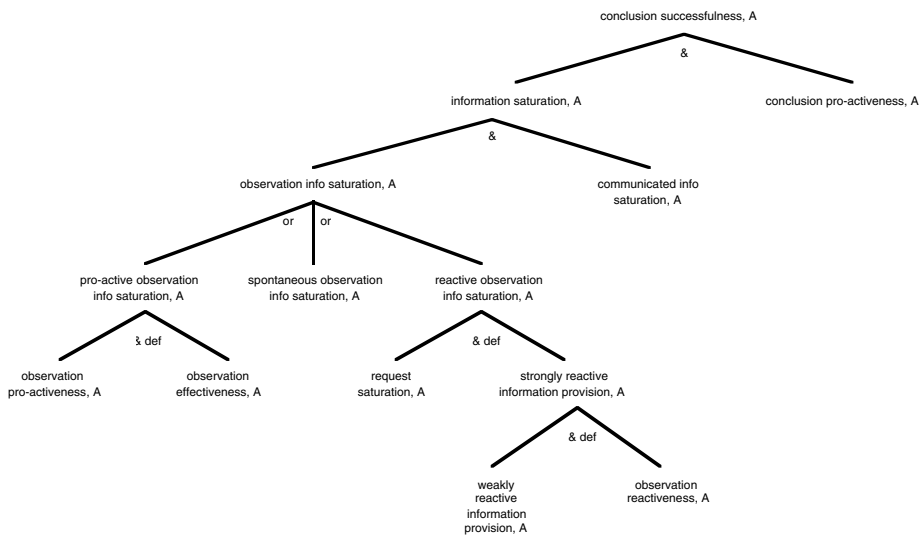


Fig. 6. Conclusion successfulness of A

All properties occurring in Figure 6 are also properties of agent A. The leaves in the tree are either environmental properties of A or behavioural properties of A. To prove environmental properties of a component behavioural properties of other components of the same level (in this case other agents and/or the world) are needed as are properties about interactions between these components (in this case interactions

between agents and between the world and agents). For example, the property communicated information saturation of agent A (see Figure 6) can be proved from interaction effectiveness from agent B to agent A and the property successful information provision of agent B, see Figure 7.

Information provision successfulness

a) The agent X is called *successful information providing* if:

$\forall \mathcal{M} \in$ Traces(X) $\forall r, \exists$ sign $\exists t$

$\text{state}_X(\mathcal{M}, t, \text{output(X)}) \models$ to_be_communicated_to(world_info, view(X,r), sign, Y)

b) The agent X is called *successful information providing pro-active* if
X is observation effective and strongly information providing pro-active.

c) The agent X is called *successful information providing reactive* if
X is request saturating and reactive observation effective.

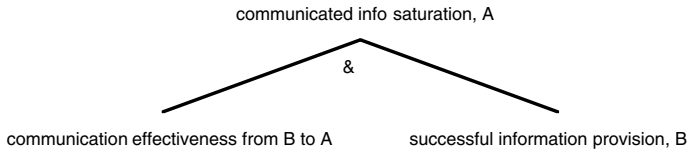


Fig. 7. Communicated info saturation of A

Similarly, the properties observation input correctness and communication input correctness of an agent depend on correct information coming from the other agent or from the world (see Section 5.3 for definitions of properties concerning the world), see Figure 8.

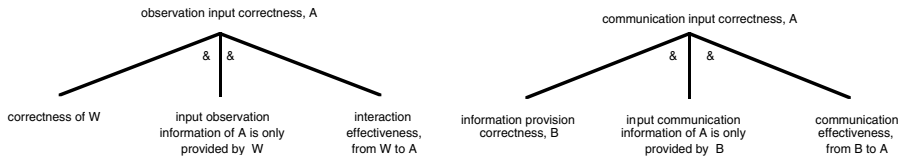


Fig. 8. Information correctness of A

The property **spontaneous observation info saturation of an agent** as used in Figure 6 is defined by the property pro-activeness of the world (see Section 5.3) and interaction effectiveness from the world to that agent, see Figure 9.

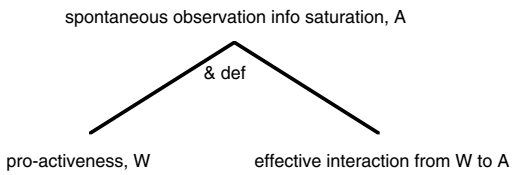


Fig. 9. Spontaneous observation info saturation of A

The property successful information provision of agent B, used in Figure 7, can be proven in several ways. The first division made in Figure 10 is between pro-active and reactive information provision. Also the notions strong and weak are used.

Information provision correctness

The agent X is called *information providing correct* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall r\ \forall t$$
$$[\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_communicated_to(world_info, view(X,r), pos, Y)}$$
$$\Rightarrow\ M \models \text{view(X,r)}\ ]$$
$$\wedge\ \forall \mathcal{M} \in \text{Traces(X)}\ \forall r\ \forall t$$
$$[\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_communicated_to(world_info, view(X,r), neg, Y)}$$
$$\Rightarrow\ M \models \neg\text{view(X,r)}\ ]$$

Information providing pro-activeness

a) The agent X is called *weakly information providing pro-active* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t, r, \text{sign}$$
$$[\ \text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{observation_result(view(X,r), sign)}$$
$$\Rightarrow \exists t\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_communicated_to(world_info, view(X,r), sign, Y)}\ ]$$

b) The agent X is called *strongly information providing pro-active* if
X is weakly information providing pro-active and observation pro-active.

Information providing reactiveness

a) The agent X is called *weakly information providing reactive* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t, t, r, \text{sign}$$
$$[\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{observation_result(view(X,r), sign)}\ \wedge$$
$$\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{communicated_from(request,view(X,r), pos,Y)}\ ]$$
$$\Rightarrow\ \exists t > t,\ t > t\ \ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models$$
$$\text{to_be_communicated_to(world_info, view(X,r), sign, Y)}$$

b) The agent X is called *strongly information providing reactive* if
X is weakly information providing reactive and observation reactive.

c) The agent X is called *reactive observation info saturating* if
X is request saturating and strongly information providing reactive.

The tree in Figure 10 consists of logical relations between properties of the agent B. Some of them have been defined above, the others are defined as follows.

Information acquisition pro-activeness of an agent

a) The agent X is called *observation pro-active* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall r\ \exists t\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_observed (view(X,r))}\ ]$$

b) The agent X is called *request pro-active* if for all agents Y different from X:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall r\ \exists t\ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models$$
$$\text{to_be_communicated_to(request, view(Y,r), pos, Y)]}$$

c) The agent X is called *information acquisition pro-active* if X is observation pro-active and request pro-active.

Observation reactiveness of an agent

The agent X is called *observation reactive* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall r\ [\text{state}_X(\mathcal{M}, t, \text{input(X)}) \models \text{communicated_from(request, view(X,r), pos, Y)}$$
$$\Rightarrow \exists t\ \ \text{state}_X(\mathcal{M}, t, \text{output(X)}) \models \text{to_be_observed (view(X,r))}\ ]$$

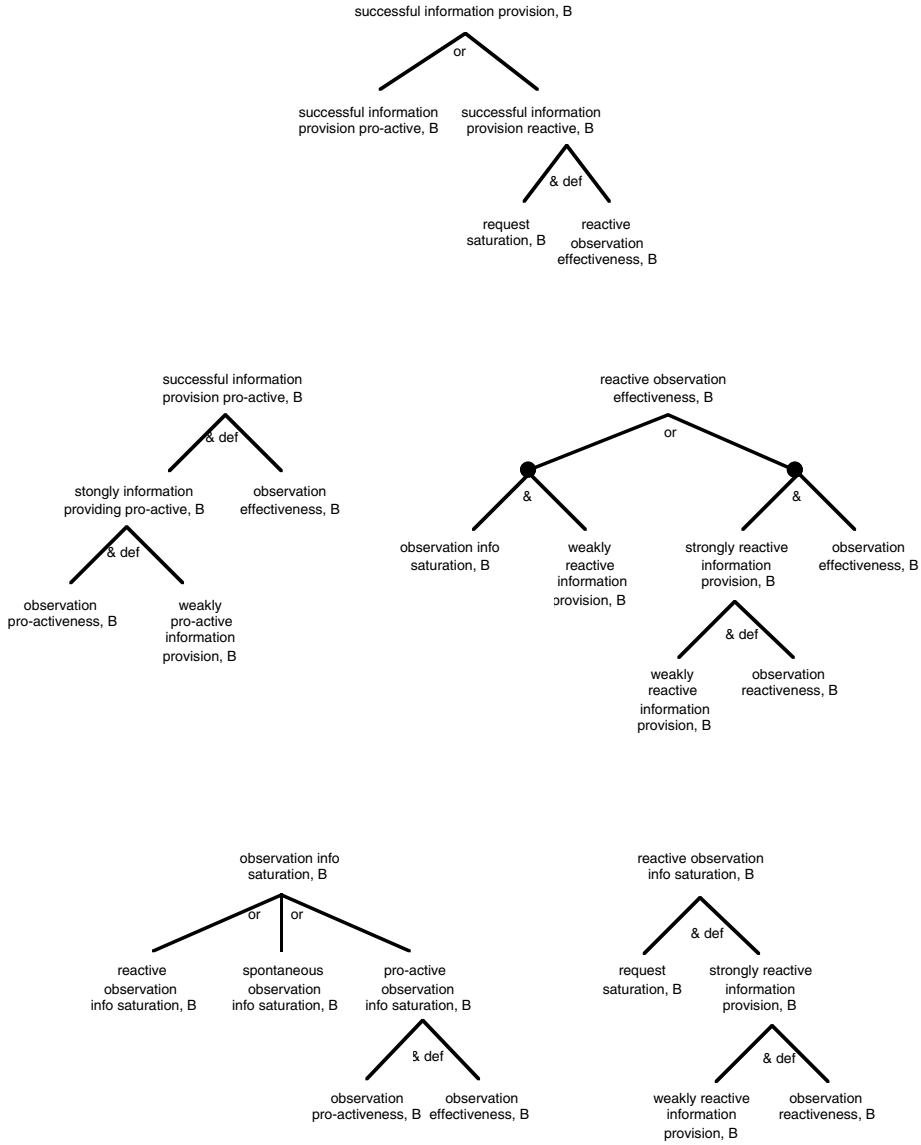


Fig. 10. Successful information provision of B

Information acquisition effectiveness of an agent

a) The agent X is called *observation effective* if:

$\forall \mathcal{M} \in$ Traces(X) $\forall t\ \forall r$ [state$_X(\mathcal{M}, t,$ output(X)) $\models$ to_be_observed (view(X,r))
$\Rightarrow \exists t' > t\ \exists$ sign state$_X(\mathcal{M}, t',$ input(X)) $\models$ observation_result(view(X,r), sign)]

b) The agent X is called *request effective* if:

$\forall \mathcal{M} \in$ Traces(X) $\forall t\ \forall r$
[state$_X(\mathcal{M}, t,$ output(X)) $\models$ to_be_communicated_to(request,view(Y,r),pos,Y)
$\Rightarrow \exists t' > t,$ sign state$_X(\mathcal{M}, t',$ input(X)) $\models$ communicated_from(world_info, view(Y,r), sign, Y)]

c) The agent X is called *information acquisition effective* if
X is observation effective and request effective.

d) The agent X is called *reactive observation effective* if:

$\forall \mathcal{M} \in$ Traces(X) $\forall t\ \forall r$
[state$_X(\mathcal{M}, t,$ input(X)) $\models$ communicated_from(request, view(Y,r), pos, Y)
$\Rightarrow \exists t' > t,$ sign state$_X(\mathcal{M}, t',$ input(X)) $\models$ observation_result(view(X,r), sign)]

e) The agent X is called *pro-active observation info saturating* if
X is observation pro-active and observation effective.

The relations between the environmental properties request saturation of agent A and observation effectiveness of agent A, and properties of the world, of agent B, and of interactions between agents and world can be found in Figure 11.

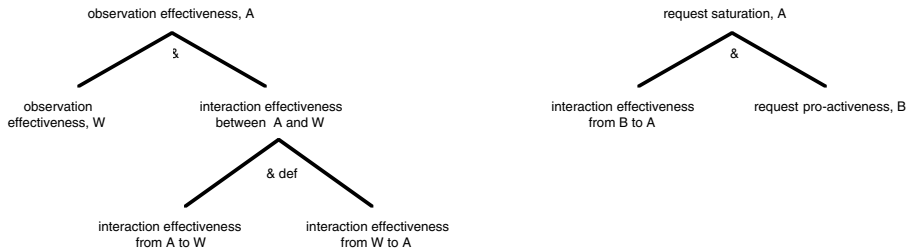


Fig. 11. Observation effectiveness and request saturation of A

5.3 Properties of the World

For the component World assumptions on correctness and conservativity are made.

Correctness of the world

The world W is called *correct* if:

$\forall \mathcal{M} \in$ Traces(W) $\forall t\ \forall v, X$
[state$_W(\mathcal{M}, t,$ output(W)) $\models$ observation_result_for(view(X, r), pos, X) $\Rightarrow$ M $\models$ view(X, r)]
$\wedge\ \forall \mathcal{M} \in$ Traces(W) $\forall t\ \forall v, X$
[state$_W(\mathcal{M}, t,$ output(W)) $\models$ observation_result_for(view(X, r), neg, X) $\Rightarrow$ M $\models \neg$ view(X, r)]

Conservativity of the world

The world W is called *conservative* if:

$\forall \mathcal{M} \in$ Traces(W) $\forall t\ \forall r,$ sign, X
[state$_W(\mathcal{M}, t,$ output(W)) $\models$ observation_result_for(view(X, r), sign, X)
$\Rightarrow\ \forall t' > t$ state$_W(\mathcal{M}, t',$ output(W)) $\models$ observation_result_for(view(X, r), sign, X)]

Moreover, the world should be effective in providing observation results for observations initiated by the agents.

Observation effectiveness of the world
The component W is called *observation effective* if:

$\forall \mathcal{M} \in$ Traces(W) $\forall t \; \forall r, X$

$[\; \text{state}_W(\mathcal{M}, t, \text{input(W)}) \models \text{to_be_observed_by(view(X, r), X)} \;]$

$\Rightarrow [\; \exists t' > t, \text{sign} \quad \text{state}_W(\mathcal{M}, t', \text{output(W)}) \models \text{observation_result_for(view(X, r), sign, X)} \;]$

It is also possible that the world provides observation information without an initiative from the agent (e.g., by automated sensors). In this case the world shows pro-activeness:

Observation pro-activeness of the world
The component W is called *observation pro-active* if:

$\forall \mathcal{M} \in$ Traces(W) $\forall r, X \;\; \exists t, \text{sign}$

$\text{state}_W(\mathcal{M}, t, \text{output(W)}) \models \text{observation_result_for(view(X, r), sign, X)}$

6 Properties of Agent Components

The properties of the agents needed to prove the properties of the top level of the system were discussed in Section 5.2. The assumed properties of the sub-components of an agent, needed to prove the behavioural properties of that agent, and the logical structure of those proofs in the form of trees are discussed in this section. Properties of the component own process control play a role in the proof of each behavioural agent property.

6.1 Properties of Own Process Control

In the component Own Process Control the agent s attitudes are explicitly represented. The attitudes are represented in the following manner:

attitude	*representation within* OPC
pro-active observation	observation_attitude(pro-active)
weakly pro-active information provision	info_provision_attitude(weakly_pro-active)
strongly pro-active information provision	info_provision_attitude(strongly_pro-active)
pro-active requesting	requesting_attitude(pro-active)
pro-active reasoning	reasoning_attitude(weakly_pro-active)
reactive observation	observation_attitude(reactive)
weakly reactive information provision	info_provision_attitude(weakly_reactive)
strongly reactive information provision	info_provision_attitude(strongly_reactive)

Attitude determination successfulness of OPC
The component OPC is called *attitude determination successful* for the attitude *pro-activeness of observation* if:

$\forall \mathcal{M} \in$ Traces(OPC) $\exists t \; \forall t' \geq t$

$\text{state}_{OPC}(\mathcal{M}, t', \text{output(OPC)}) \models \text{observation_attitude(pro-active)}$

In a similar manner attitude determination successfulness for the other attitudes is defined.

In the example the attitudes are assumed to be defined in a static manner, as general facts in OPC. However, it is not difficult to define them dynamically, (i.e., that an agent may change its attitude on the basis of experiences) by specifying a knowledge base that takes into account (dynamic) input for OPC.

Attitude conservativity of OPC

The component OPC is called *attitude conservative* for the attitude *pro-activeness of observation* if:

$\forall \mathcal{M} \in$ Traces(OPC) $\forall t$
$\text{state}_{OPC}(\mathcal{M}, t, \text{output(OPC)}) \models$ observation_attitude(pro-active)
$\Rightarrow \forall t > t \ \text{state}_{OPC}(\mathcal{M}, t, \text{output(OPC)}) \models$ observation_attitude(pro-active)

In a similar manner attitude conservativity for the other attitudes is defined.

In order to prove observation pro-activeness of agent A not only properties of OPC are needed, but also of the component WIM. The logical relations between these properties can be found in Figure 12.

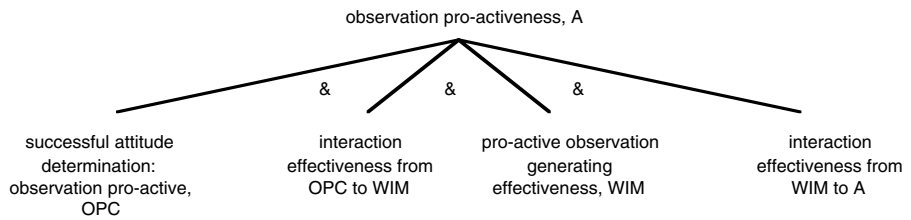


Fig. 12. Observation pro-activeness of A

The properties concerning interaction effectiveness used on this level correspond to the same properties on the top level. Explicit definitions have been omitted in this paper.

6.2 Properties of World Interaction Management

If the agent is pro-active for observation, see Figure 12, then the agent makes sure that every observation is performed at least once. The component world interaction management initiates these observations.

Pro-active observation generation effectiveness of WIM

The component WIM of agent X is called *pro-actively observation generation effective* if:

$\forall \mathcal{M} \in$ Traces(WIM) $\forall t \ \forall r$
[$\text{state}_{WIM}(\mathcal{M}, t, \text{input(WIM)}) \models$ observation_attitude(pro-active)
$\Rightarrow \exists t \ \text{state}_{WIM}(\mathcal{M}, t, \text{output(WIM)}) \models$ to_be_observed (view(X,r))]

Note that no temporal restrictions are put on t: either the observation has been generated in the past (in which case no new observation has to be initiated), or it has to be done now or in the future.
In the reactive case, also the presence of a request is of importance:

Reactive observation generation effectiveness of WIM
The component WIM of agent X is called *reactively observation generation effective* if:

$\forall \mathcal{M} \in$ Traces(WIM) $\forall t \; \forall r$
[[$state_{WIM}(\mathcal{M}, t, input(WIM)) \models$ observation_attitude(reactive) $\wedge$
$state_{WIM}(\mathcal{M}, t, input(WIM)) \models$ requested(view(X,r))]
$\Rightarrow \exists t \;\; state_{WIM}(\mathcal{M}, t, output(WIM)) \models$ to_be_observed (view(X,r))]

Given this property and the ability of AIM to pass on request information to WIM it is possible to prove observation reativeness of agent A, see Figure 13.

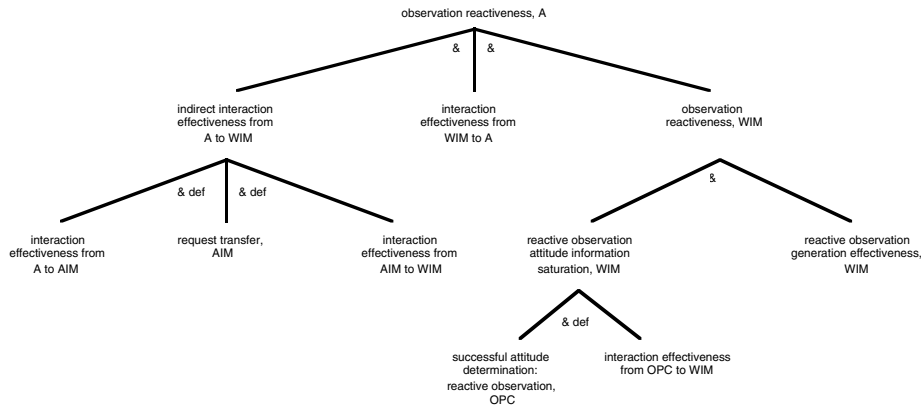


Fig. 13. Observation reactiveness of A

Observation result transfer of WIM
The component WIM of agent X is called *observation result transferring* if:

$\forall \mathcal{M} \in$ Traces(WIM) $\forall t \; \forall r$, sign
[$state_{WIM}(\mathcal{M}, t, input(WIM)) \models$ observation_result (view(X,r), sign)
$\Rightarrow \exists t' \geq t \; state_{WIM}(\mathcal{M}, t', output(WIM)) \models$ observation_result (view(X,r), sign)]

This property is used in Figure 14, 15, and 17.

In order to prove weakly pro-active or weakly reactive information provision of A, the properties of the component agent interaction management are of importance.

6.3 Properties of Agent Interaction Management

Pro-active information provision effectiveness of AIM

The component AIM of agent X is called *weakly pro-actively information provision effective* if for every agent Y different from X:

$\forall \mathcal{M} \in$ Traces(AIM) $\forall t \; \forall r$, sign

[$\text{state}_{AIM}(\mathcal{M}, t, \text{input(AIM)}) \models$ info_provision_attitude(weakly_pro-active) $\wedge$

$\text{state}_{AIM}(\mathcal{M}, t, \text{input(AIM)}) \models$ observation_result (view(X,r), sign)

$\Rightarrow \exists t$ $\text{state}_{AIM}(\mathcal{M}, t, \text{output(AIM)}) \models$ to_be_communicated_to(world_info, view(X,r), pos, Y)]

Figure 14 shows how the agent property weakly pro-active information provision depends on other properties of AIM. The component AIM needs observation information, therefore, the observation result transfer property of WIM, and effective interaction from the input of the agent to WIM, and from WIM to AIM should hold. The correct necessary attitude information is provided by OPC.

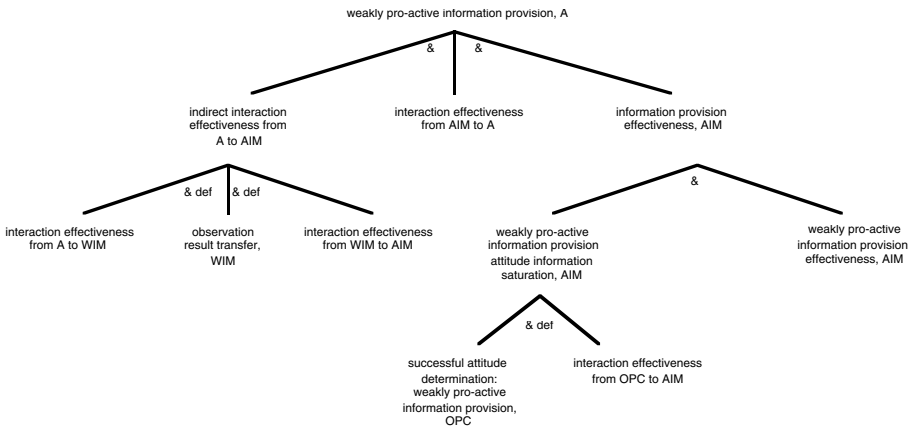


Fig. 14. Weakly pro-active information provision of A

Reactive information provision effectiveness of AIM

The component AIM of agent X is called *weakly reactively information provision effective* if for every agent Y different from X:

$\forall \mathcal{M} \in$ Traces(AIM) $\forall t,t \; \forall r$, sign

[$\text{state}_{AIM}(\mathcal{M}, t, \text{input(AIM)}) \models$ info_provision_attitude(weakly_reactive) $\wedge$

$\text{state}_{AIM}(\mathcal{M}, t, \text{input(AIM)}) \models$ observation_result (view(X,r), sign) $\wedge$

$\text{state}_{AIM}(\mathcal{M}, t, \text{input(AIM)}) \models$ requested(view(X,r))

$\Rightarrow \exists t \geq t$ $\text{state}_{AIM}(\mathcal{M}, t, \text{output(AIM)}) \models$

to_be_communicated_to(world_info, view(X,r), pos, Y)]

Similarly, the reactive information provision effectiveness property of AIM is needed to prove the agent property weakly reactive information provision, see Figure 15.

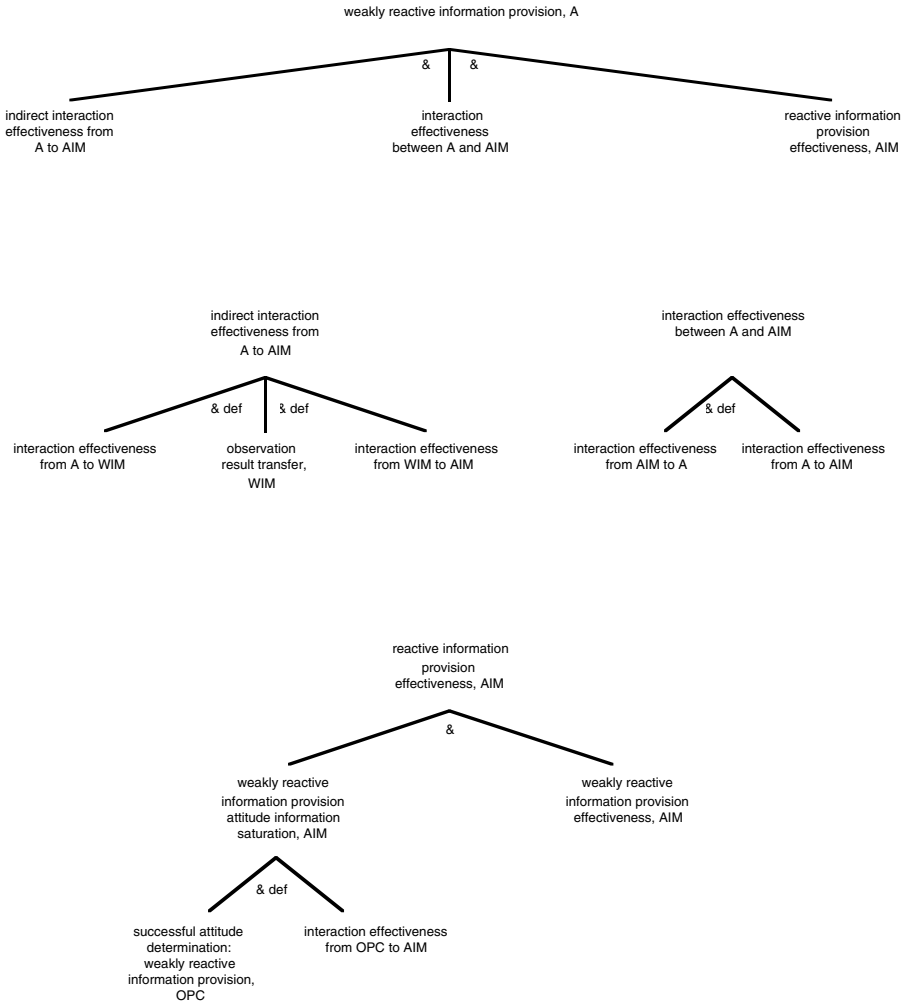


Fig. 15. Weakly reactive information provision of A

Request transfer of AIM

The component AIM of agent X is called *request transferring* if for every agent Y different from X:

$\forall \mathcal{M} \in$ Traces(AIM) $\forall t \; \forall r$

[$\text{state}_{\text{AIM}}(\mathcal{M}, t, \text{input(AIM)}) \models$ communicated_by (request, view(X,r), pos, Y)

$\Rightarrow \exists t' \geq t \; \text{state}_{\text{AIM}}(\mathcal{M}, t', \text{output(AIM)}) \models$ requested(view(X,r))]

This property is used in Figure 13 and in Figure 17. The following property can be used to prove request pro-activeness of agent A, see Figure 16.

Pro-active request generation effectiveness of AIM

The component AIM of agent X is called *pro-actively request generation effective* if for every agent Y different from X:

$\forall \mathcal{M} \in$ Traces(AIM) $\forall t \forall r$
[$state_{AIM}(\mathcal{M}, t, input(AIM)) \models$ requesting_attitude(pro-active)
$\Rightarrow \exists t$ $state_{AIM}(\mathcal{M}, t, output(AIM)) \models$ to_be_communicated_to(request, view(Y,r), pos, Y)]

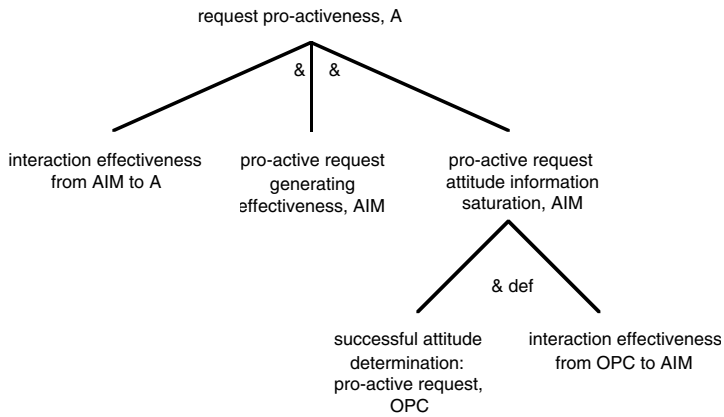


Fig. 16. Request pro-activeness of A

Communicated information transfer of AIM

The component AIM of agent X is called *communicated information transferring* if for every agent Y different from X:

$\forall \mathcal{M} \in$ Traces(AIM) $\forall t \forall r$, sign [$state_{AIM}(\mathcal{M}, t, input(AIM)) \models$
communicated_by (world_info, view(Y,r), sign, Y)
$\Rightarrow \exists t \geq t$ $state_{AIM}(\mathcal{M}, t, output(AIM)) \models$ received_world_info(view(Y,r), sign)]

6.4 Properties of the Agent Specific Task: OC

The required properties of the component OC are the following. Conclusiveness defines that the component is able to draw decisive conclusions if sufficient input is provided.

Conclusiveness of OC

The component OC is called *conclusive* if, under the condition that all required input information has been acquired, for every output atom a conclusion is derived.

$\forall \mathcal{M} \in$ Traces(OC) [$\forall Y \exists r,t$ $state_{OC}(\mathcal{M}, t, input(OC)) \models$ view(Y,r)]
$\Rightarrow \forall s$ [$\exists t$ $state_{OC}(\mathcal{M}, t , output(OC)) \models$ object_type(s) $\vee$
$\exists t$ $state_{OC}(\mathcal{M}, t , output(OC)) \models \neg$object_type(s)]

To allow that OPC controls the reasoning on the basis of its reasoning attitude, the following conditional variant of conclusiveness is needed. This means that only conclusions are drawn if OC has been input (transferred from OPC) the right targets. Conditional conclusiveness is used to prove conclusion pro-activeness of agent A in Figure 17.

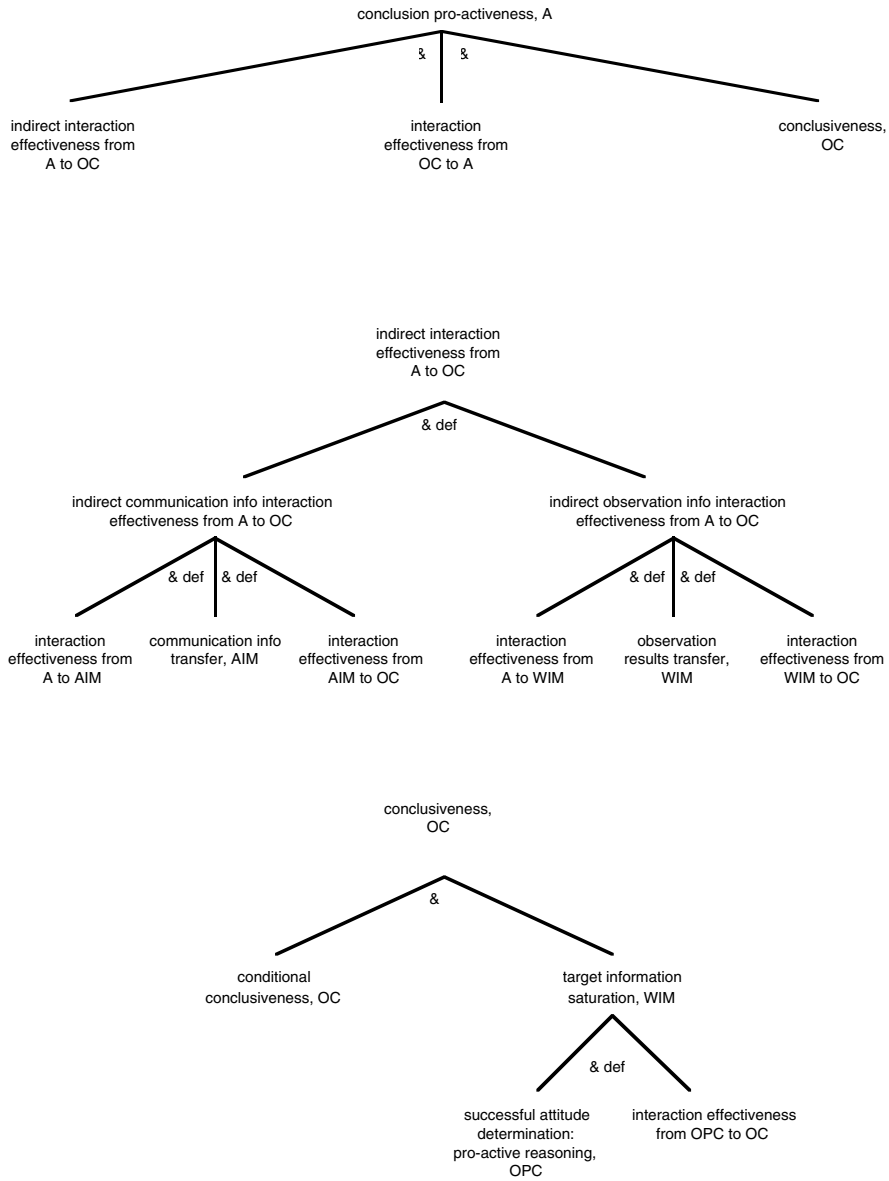


Fig. 17. Conclusion pro-activeness of A

Conditional conclusiveness of oc
The component oc is called *conditionally conclusive* if, under the condition that all required input information has been acquired, for every output atom which is associated to its focus (as a target), a conclusion is derived:

$$\forall \mathcal{M} \in \text{Traces(OC)}\ \forall s$$
$$[\forall Y\ \exists r,t\ \text{state}_{OC}(\mathcal{M}, t, \text{input(OC)}) \models \text{view(Y,r)}\ ] \wedge$$
$$[\text{state}_{OC}(\mathcal{M}, t, \text{input(OC)}) \models \text{target(OC_focus, object_type(s), determine)}$$
$$\Rightarrow [\exists t\ \text{state}_{OC}(\mathcal{M}, t, \text{output(OC)}) \models \text{object_type(s)} \vee$$
$$\exists t\ \text{state}_{OC}(\mathcal{M}, t, \text{output(OC)}) \models \neg \text{object_type(s)}]$$

Conclusion correctness means: if a conclusion is derived, then this conclusion corresponds to the world situation.

Conclusion correctness of oc
The component OC is called *conclusion correct* if:

$$\forall \mathcal{M} \in \text{Traces(OC)}\ \forall t\ \forall s$$
$$[\ \text{state}_{OC}(\mathcal{M}, t, \text{output(OC)}) \models \text{object_type(s)} \Rightarrow M \models \text{object_type(s)}\ ]$$
$$\wedge\ [\text{state}_{OC}(\mathcal{M}, t, \text{output(OC)}) \models \neg\ \text{object_type(s)} \Rightarrow M \models \neg \text{object_type(s)}\ ]$$

Conservation can be defined by:

Conclusion conservativity of oc
The component OC is called *conclusion conservative* if:

$$\forall \mathcal{M} \in \text{Traces(X)}\ \forall t\ \forall s$$
$$[\ \text{state}_{OC}(\mathcal{M}, t, \text{output(OC)}) \models \text{object_type(s)} \Rightarrow$$
$$\forall t' > t\ \ \text{state}_{X}(\mathcal{M}, t', \text{output(OC)}) \models \text{object_type(s)}\ ]$$
$$\wedge\ [\ \text{state}_{OC}(\mathcal{M}, t, \text{output(OC)}) \models \neg\ \text{object_type(s)} \Rightarrow$$
$$\forall t' > t\ \ \text{state}_{OC}(\mathcal{M}, t', \text{output(OC)}) \models \neg\ \text{object_type(s)}\ ]$$

6.5 Domain Assumptions

The properties also need assumptions on the domain knowledge to be used in the model.

Static world
The world state is static during the processing of the system s.

Empirically foundedness
The possible conclusions can be uniquely characterised by means of observations; in other words: if two world situations satisfy exactly the same observations, then they also satisfy exactly the same conclusions (see Treur and Willems, 1994).

7 Verification of Primitive Components

In Sections 5 and 6 verification of the multi-agent model was described, based on assumed properties of the primitive components. The primitive components can be verified making use of the more standard methods introduced in (Treur and Willems, 1994; Leemans, Treur and Willems, 1993). For example, the component Object Classification should satisfy conclusion correctness and conditional conclusiveness.

Actually, these two properties reduce to static properties described in (Treur and Willems, 1994). In fact all properties required for primitive components reduce to static properties that define a constraint on the combined input-output states of the component. Such properties can be verified by the (static) methods described in the references mentioned.

8 Conclusions

The modelling approach DESIRE is based on compositionality of processes and knowledge at different levels of abstraction. The compositional verification method described in this paper fits well to DESIRE, but can also be useful to any other compositional modelling approach. Two main advantages of a compositional approach to modelling are the transparent structure of the design and support for reuse of components and generic models. The compositional verification method extends these main advantages to (1) a well-structured verification process, and (2) the reusability of proofs for properties of components that are reused.

The first advantage entails that both conceptually and computationally the complexity of the verification process can be handled by compositionality at different levels of abstraction. Apart from the work reported here, a generic model for diagnosis has been verified (Cornelissen, Jonker and Treur, 1997) and a multi-agent system with agents negotiating about load-balancing of electricity use. The second advantage entails: if a modified component satisfies the same properties as the previous one, the proof of the properties at the higher levels of abstraction can be reused to show that the new system has the same properties as the original. This has high value for a library of reusable generic models and components. The verification of generic models forces one to find the assumptions under which for the considered domain the generic model is applicable, as is also discussed in (Fensel, 1995; Fensel and Benjamins, 1996). A library of reusable components and task models may consist of both specifications of the components and models, and their design rationale. As part of the design rationale, at least the properties of the components and their logical relations can be documented.

Also due to the compositional nature of the verification method, a distributed approach to verification is facilitated. This implies that several persons can work on the verification of the same system at the same time, once the properties to be verified have been determined. Since the proof of properties of a composed component depends on the properties of its sub-components, it is only necessary to know or to agree on the properties of these sub-components.

The formal analysis of variants of reactiveness and pro-activeness properties deepened our insight in these notions and their logical relationships and interactions. Semantical formalisation of different variants of reactiveness and pro-activeness have been found in the form of conditional temporal statements. The notion of information and process hiding, in DESIRE modelled in terms of components at different abstraction levels, made it possible to distinguish in a natural manner between observable and non-observable variants of pro-activeness and reactiveness: the variants of behaviour that can be observed from outside the agent (at its interface), and the variants of internal behaviour (in its sub-components and interactions between them) that cannot be observed from outside. This formal analysis could be a starting point

for a more general mathematical or logical theory on pro-activeness and reactiveness, and their interaction. Actually, the logical relations, in this paper depicted in the form of AND/OR graphs in the figures, can be viewed as lemmas and theorems in such a theory.

A main difference in comparison to (Fisher and Wooldridge, 1997) is that our approach exploits compositionality. An advantage of their approach is that they can make use of a temporal belief logic. It would be a challenge to extend the approach as referred to a compositional variant of temporal belief logic. A first step in this direction can be found in (Engelfriet, Jonker and Treur, 1997). Also a main difference of the current paper in comparison to the work in (Fensel, 1995, Fensel and Benjamins, 1996; Fensel et al, 1996) is that in our approach compositionality of the verification is addressed; in the work as referred only domain assumptions are taken into account, and no hierarchical relations between properties are defined.

A future continuation of this work will consider the development of tools for verification. At the moment only tools exist for the verification of primitive components; no tools for the verification of composed components exist yet. To support the handwork of verification it would be useful to have tools to assist in the creation of the proof. This could be done by formalising the proofs of a verification process using a first order logic in which time and states are represented explicitly, and an interactive theorem prover to support the proofs. Another option that will be explored is to extend Fisher and Wooldridge s approach to the compositional case. Yet another option to be explored is whether the tool KIV (based on dynamic logic) can be used. Some first, positive experiences with KIV for verification of an example model of a knowledge-based system are reported in (Fensel et al, 1996).

Acknowledgements

Wieke de Vries has read an earlier version of this paper, which has led to a number of improvements in the text.

References

Abadi, M. and L. Lamport (1993). Composing Specifications, ACM Transactions on Programming Languages and Systems, Vol. 15, No. 1, p. 73-132.

Benjamins, R., Fensel, D., Straatman, R. (1996). Assumptions of problem-solving methods and their role in knowledge engineering. In: W. Wahlster (ed.), Proceedings of the 12th European Conference on AI, ECAI'96, John Wiley and Sons, pp. 408-412.

Brazier, F.M.T. , Dunin-Keplicz, B., Jennings, N.R. and Treur, J. (1995). Formal specification of Multi-Agent Systems: a real-world case. In: V. Lesser (ed.), Proceedings of the First International Conference on Multi-Agent Systems, ICMAS-95, MIT Press, Cambridge, MA, pp. 25-32. Extended version in: International Journal of Cooperative Information Systems, M. Huhns, M. Singh, (Eds.), special issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, vol. 6, 1997, pp. 67-94.

Brazier, F.M.T., Treur, J., Wijngaards, N.J.E. and Willems, M. (1996). Temporal semantics of complex reasoning tasks. In: B.R. Gaines, M.A. Musen (eds.), Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-based Systems workshop, KAW'96, Calgary: SRDG Publications, Department of Computer Science, University of Calgary, pp. 15/1-15/17. Extended version to appear in Data and Knowledge Engineering, 1998

Cornelissen, F., Jonker, C.M., Treur, J. (1997). Compositional verification of knowledge-based systems: a case study in diagnostic reasoning. In: E.Plaza, R. Benjamins (eds.), Knowledge Acquisition, Modelling and Management, Proc. of the 10th EKAW, Lecture Notes in AI, vol. 1319, Springer Verlag, pp. 65-80. Extended abstract in: Proceedings of the Fourth European Symposium on the Validation and Verification of Knowledge-based Systems, EUROVAV'97.

Dams, D., Gerth, R., Kelb, P. (1996). Practical Symbolic Model Checking of the full μ-calculus using Compositional Abstractions. Report, Eindhoven University of Technology, Department of Mathematics and Computer Science.

Engelfriet, J., Jonker, C.M., Treur, J., (1997). Compositional Verification of Knowledge-based Systems in Temporal Epistemic Logic. In: A. Bossi (ed.), Proceedings of the ILPS'97 Workshop on Verification.

Fensel, D. (1995). Assumptions and limitations of a problem solving method: a case study. In: B.R. Gaines, M.A. Musen (eds.), Proceedings of the 9th Banff Knowledge Acquisition for Knowledge-based Systems workshop, KAW'95, Calgary: SRDG Publications, Department of Computer Science, University of Calgary.

Fensel, D., Benjamins, R. (1996). Assumptions in model-based diagnosis. In: B.R. Gaines, M.A. Musen (eds.), Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-based Systems workshop, KAW'96, Calgary: SRDG Publications, Department of Computer Science, University of Calgary, pp. 5/1-5/18.

Fensel, D., Schonegge, A., Groenboom, R., Wielinga, B. (1996). Specification and verification of knowledge-based systems. In: B.R. Gaines, M.A. Musen (eds.), Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-based Systems workshop, KAW'96, Calgary: SRDG Publications, Department of Computer Science, University of Calgary, pp. 4/1-4/20.

Fisher, M., Wooldridge, M. (1997) On the Formal Specification and Verification of Multi-Agent Systems. International Journal of Cooperative Information Systems, M. Huhns, M. Singh, (eds.), special issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems, vol. 6, pp. 67-94.

Harmelen, F. van and Fensel, D. (1995). Formal Methods in Knowledge Engineering. Knowledge Engineering Review, Volume 10, Number 4.

Hooman, J. (1994). Compositional Verification of a Distributed Real-Time Arbitration Protocol. Real-Time Systems, vol. 6, pp. 173-206.

Kinny, D., Georgeff, M.P., Rao, A.S. (1996). A Methodology and Technique for Systems of BDI Agents. In: W. van der Velde, J.W. Perram (eds.), Agents Breaking Away, Proceedings 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96, Lecture Notes in AI, vol. 1038, Springer Verlag, pp. 56-71.

Langevelde, I.A. van, A. Philipsen, and J. Treur (1992). Formal Specification of Compositional Architectures. In B. Neumann (ed.), Proceedings of the 10th European Conference on AI, ECAI 92, Wiley and Sons, pp. 272-276.

Leemans, P., J. Treur, and M. Willems (1993). On the verification of knowledge-based reasoning modules, Report IR-346, Department of Mathematics & Computer Science, AI Group, Vrije Universiteit Amsterdam.

Rao, A.S. and Georgeff, M.P. (1991). Modeling rational agents within a BDI architecture. In: R. Fikes and E. Sandewall (eds.), Proceedings of the Second Conference on Knowledge Representation and Reasoning, Morgan Kaufman, pp. 473-484.

Treur, J., and M. Willems (1994). A logical foundation for verification. In: Proceedings of the 11th European Conference on AI, ECAI 94, A. Cohn (ed.), John Wiley & Sons, Ltd., pp. 745-749.

Wooldridge, M., N.R. Jennings (eds.) (1995), Intelligent Agents, Proceedings of the First International Workshop on Agent Theories, Architectures and Languages, Lecture Notes in AI, vol. 890, Springer Verlag.

Modular Model Checking *

Orna Kupferman[1]** and Moshe Y. Vardi[2]***

[1] EECS Department, UC Berkeley, Berkeley CA 94720-1770, U.S.A.
Email: orna@eecs.berkeley.edu URL: http://www.eecs.berkeley.edu/~orna
[2] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Abstract. In *modular verification* the specification of a module consists of two parts. One part describes the guaranteed behavior of the module. The other part describes the assumed behavior of the system in which the module is interacting. This is called the *assume-guarantee* paradigm. In this paper we consider assume-guarantee specifications in which the guarantee is specified by branching temporal formulas. We distinguish between two approaches. In the first approach, the assumption is specified by branching temporal formulas. In the second approach, the assumption is specified by linear temporal logic. We consider guarantees in $\forall$CTL and $\forall$CTL*, the universal fragments of CTL and CTL*, and assumptions in LTL, $\forall$CTL, and $\forall$CTL*. We describe a reduction of modular model checking to standard model checking. Using the reduction, we show that modular model checking is PSPACE-complete for $\forall$CTL and is EXPSPACE-complete for $\forall$CTL*. We then show that the case of LTL assumption is a special case of the case of $\forall$CTL* assumption, but that the EXPSPACE-hardness result apply already to assumptions in LTL.

1 Introduction

Temporal logics, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying concurrent programs [Pnu77, Pnu81]. One of the most significant developments in this area is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* programs [CES86, LP85, QS81]. This

* This paper is based on "On the complexity of modular model checking", by M.Y. Vardi, *Proc. 10th IEEE Symp. on Logic in Computer Science* (LICS'95), June 1995, pp. 101–111, and "On the complexity of branching modular model checking", by O. Kupferman and M.Y. Vardi, *Proc. 6th International Conf. on Concurrency Theory* (CONCUR'95), August 1995, Springer-Verlag, Lecture Notes in Computer Science 962, pp. 408–422.

** Supported in part by ONR YIP award N00014-95-1-0520, by NSF CAREER award CCR-9501708, by NSF grant CCR-9504469, by AFOSR contract F49620-93-1-0056, by ARO MURI grant DAAH-04-96-1-0341, by ARPA grant NAG2-892, and by SRC contract 95-DC-324.036.

*** Supported in part by NSF grants CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 381-401, 1998.
Springer-Verlag Berlin Heidelberg 1998

derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state programs, as well as from the great ease of use of fully algorithmic methods. Finite-state programs can be modeled by transition systems where each state has a bounded description, and hence can be characterized by a fixed number of boolean atomic propositions. This means that a finite-state program can be viewed as a finite *propositional Kripke structure* and its properties can be specified using *propositional* temporal logic. Thus, to verify the correctness of the program with respect to a desired behavior, one only has to check that the program, modeled as a finite Kripke structure, satisfies (is a model of) the propositional temporal logic formula that specifies that behavior. Hence the name *model checking* for the verification methods derived from this viewpoint. Surveys can be found in [CG87, Wol89, CGL93].

We distinguish between two types of temporal logics: linear and branching [Lam80]. In linear temporal logics, each moment in time has a unique possible future, while in branching temporal logics, each moment in time may split into several possible futures. The complexity of model checking for both linear and branching temporal logics is well understood: suppose we are given a program of size n and a temporal logic formula of size m. For a branching temporal logic such as CTL, model-checking algorithms run in time $O(nm)$ [CES86], while, for linear temporal logic such as LTL, model-checking algorithms run in time $n2^{O(m)}$ [LP85]. Since model checking with respect to a linear temporal logic formula is PSPACE-complete [SC85], the latter bound probably cannot be improved. The difference in the complexity of linear and branching model checking has been viewed as an argument in favor of the branching paradigm.

Model checking suffers, however, from the so-called *state-explosion* problem. In a concurrent setting, the program under consideration is typically the parallel composition of many modules. As a result, the size of the state space of the program is the product of the sizes of the state spaces of the participating modules. This gives rise to state spaces of exceedingly large sizes, which makes even linear-time algorithms impractical. This issue is one of the most important one in the area of computer-aided verification and is the subject of active research (cf. [BCM+90]).

Modular verification is one possible way to address the state-explosion problem, cf. [CLM89, ASSS94]. In modular verification, one uses proof rules of the following form:

$$\left.\begin{array}{l} M_1 \models \psi_1 \\ M_2 \models \psi_2 \\ C(\psi_1, \psi_2, \psi) \end{array}\right\} M_1 \| M_2 \models \psi$$

Here, $M \models \theta$ means that the module M satisfies the formula θ, the symbol "$\|$" denotes parallel composition, and $C(\psi_1, \psi_2, \psi)$ is some logical condition relating ψ_1, ψ_2, and ψ. Using modular proof rules enables one to apply model checking only to the underlying modules, which have much smaller state spaces.

The state-explosion problem is only one motivation for pursuing modular verification. Modular verification is advocated also for other methodological reasons; a robust verification methodology should provide rules for deducing properties

of programs from the properties of their constituent modules. Indeed, efforts to develop modular verification frameworks were undertaken in the mid 1980s [Pnu85b].

A key observation, see [Lam83, Jon83, MC81], is that in modular verification the specification should include two parts. One part describes the desired behavior of the module. The other part describes the assumed behavior of the system within which the module is interacting. This is called the *assume-guarantee* paradigm, as the specification describes what behavior the module is *guaranteed* to exhibit, *assuming* that the system behaves in the promised way.

For the linear temporal paradigm, an assume-guarantee specification is a pair $\langle \varphi, \psi \rangle$, where both φ and ψ are linear temporal logic formulas. The meaning of such a pair is that all the computations of the module are guaranteed to satisfy ψ, assuming that all the computations of the environment satisfy φ. As observed in [Pnu85b], in this case the assume-guarantee pair $\langle \varphi, \psi \rangle$ can be combined to a single linear temporal logic formula $\varphi \rightarrow \psi$ (see also [JT95]). Thus, model checking a module with respect to assume-guarantee specifications in which both the assumed and the guaranteed behaviors are linear temporal logic formulas is essentially the same as model checking the module with respect to linear temporal logic formulas.

The situation is different for the branching temporal paradigm. Here the guarantee is a branching temporal formula, which describes the computation tree of the module. There are two approaches, however, to the assumptions in assume-guarantee pairs. The first approach, implicit in [CES86, EL85b, EL87] and made explicit in [Jos87a, Jos87b, Jos89, DDGJ89], is that the assumption in the assume-guarantee pair concerns the interaction of the module with its environment along each computation, and is therefore more naturally expressed in linear temporal logic. Thus, in this approach, an assume-guarantee pair should consist of a *linear* temporal assumption φ and a *branching* temporal guarantee ψ. The meaning of such a pair is that ψ holds in the computation tree that consists of all computations of the program that satisfy φ. The problem of verifying that a given module M satisfies such a pair $\langle \varphi, \psi \rangle$, which we call the *linear-branching modular model-checking problem*, is more general than either linear or branching model checking.

A second approach was considered in [GL94], where assumptions are taken to apply to the computation tree of the system within which the module is interacting. Accordingly, assumptions in [GL94] are also expressed in branching temporal logic. There, a module M satisfies an assume-guarantee pair $\langle \varphi, \psi \rangle$ iff whenever M is part of a system satisfying φ, the system satisfies ψ too. We call this *branching modular model checking*. Furthermore, it is argued there, as well as in [DDGJ89, Jos89, GL91, DGG93], that in the context of modular verification it is advantageous to use only *universal* branching temporal logic, i.e., branching temporal logic without existential path quantifiers. That is, in a universal branching temporal logic one can state properties of all computations of a program, but one cannot state that certain computations exist. Consequently, universal branching temporal logic formulas have the helpful property that once

they are satisfied in a module, they are satisfied also in a system that contains this module. The focus in [GL94] is on using ∀CTL, the universal fragment of CTL, for both the assumption and the guarantee.

In this paper, we focus on the branching modular model-checking problem, which we show to be a proper extension of the linear-branching modular model-checking problem. We consider assumptions and guarantees in both ∀CTL and in the more expressive ∀CTL*. Our key result is that modular model checking can be reduced to standard model checking. At the same time, we show that there is a significant penalty in computational complexity. The fundamental technique used here is the *maximal-model* technique introduced in [GL94]. It is shown there that with every ∀CTL formula φ one can associate a *maximal model* M_φ (called the *tableau* of φ in [GL94]) such that a module M satisfies φ precisely when M *simulates* M_φ (we define simulation later on). We use here automata-theoretic techniques for CTL* [VS85, EJ88] to construct maximal models for ∀CTL* formulas. While maximal models for ∀CTL involve an exponential blow-up, maximal models for ∀CTL* involve a doubly exponential blow-up. The maximal-model technique yield optimal algorithms for modular model checking. We prove that the problem is PSPACE-complete for ∀CTL and is EXPSPACE-complete for ∀CTL*. We then show that the linear-branching model-checking problem is a special case of the branching modular model-checking problem, but that the EXPSPACE-hardness result apply already to assumptions in linear temporal logic. We also show that the increase in complexity is solely to the assumption part of the specification. This suggests that modular model checking in the branching temporal framework can be practical only for very small assumptions.

2 Preliminaries

2.1 The Temporal Logics LTL, CTL*, and CTL

The logic *LTL* is a linear temporal logic. Formulas of LTL are built from a set AP of atomic proposition using the usual Boolean operators and the temporal operators X ("next time"), U ("until"), and $\tilde{U}$ ("duality of until"). We present here a positive normal form in which negation may be applied only to atomic propositions. Given a set AP, an LTL formula is defined as follows:

- **true**, **false**, p, or $\neg p$, for $p \in AP$.
- $\psi \vee \varphi$, $\psi \wedge \varphi$, $X\psi$, $\psi U \varphi$, or $\psi \tilde{U} \varphi$, where ψ and φ are LTL formulas.

We define the semantics of LTL with respect to a *computation* $\pi = \sigma_0, \sigma_1, \ldots$, where for every $j \geq 0$, we have that σ_j is a subset of AP, denoting the set of atomic propositions that hold in the j's position of π. We denote the suffix $\sigma_j, \sigma_{j+1}, \ldots$ of π by π^j. We use $\pi \models \psi$ to indicate that an LTL formula ψ holds in the path π. The relation $\models$ is inductively defined as follows:

- For all π, we have that $\pi \models$ **true** and $\pi \not\models$ **false**.
- For an atomic proposition $p \in AP$, we have $\pi \models p$ iff $p \in \sigma_0$ and $\pi \models \neg p$ iff $p \notin \sigma_0$.

- $\pi \models \psi \vee \varphi$ iff $\pi \models \psi$ or $\pi \models \varphi$.
- $\pi \models \psi \wedge \varphi$ iff $\pi \models \psi$ and $\pi \models \varphi$.
- $\pi \models X\psi$ iff $\pi^1 \models \psi$.
- $\pi \models \psi U \varphi$ iff there exists $k \geq 0$ such that $\pi^k \models \varphi$ and $\pi^i \models \psi$ for all $0 \leq i < k$.
- $\pi \models \psi \tilde{U} \varphi$ iff for every $k \geq 0$ for which $\pi^k \not\models \varphi$, there exists $0 \leq i < k$ such that $\pi^i \models \psi$.

We denote the size of a formula φ by $|\varphi|$ and we use the following abbreviations in writing formulas:

- $\rightarrow$ and $\leftrightarrow$, interpreted in the usual way.
- $F\psi = \mathbf{true} U \psi$ ("eventually").
- $G\psi = \neg F \neg \psi$ ("always").

The logic $CTL^\star$ is a branching temporal logic. A path quantifier, E ("for some path") or A ("for all paths"), can prefix an assertion composed of an arbitrary combination of linear time operators. There are two types of formulas in CTL*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific path. Formally, let AP be a set of atomic proposition names. A CTL* state formula (again, in a positive normal form) is either:

- **true**, **false**, p or $\neg p$, for $p \in AP$.
- $\psi \vee \varphi$ or $\psi \wedge \varphi$ where ψ and φ are CTL* state formulas.
- $E\psi$ or $A\psi$, where ψ is a CTL* path formula.

A CTL* path formula is either:

- A CTL* state formula.
- $\psi \vee \varphi$, $\psi \wedge \varphi$, $X\psi$, $\psi U \varphi$, or $\psi \tilde{U} \varphi$, where ψ and φ are CTL* path formulas.

The logic CTL* consists of the set of state formulas generated by the above rules. The logic CTL is a restricted subset of CTL*. In CTL, the temporal operators X, U, and $\tilde{U}$ must be immediately preceded by a path quantifier. Formally, it is the subset of CTL* obtained by restricting the path formulas to be $X\psi$, $\psi U \varphi$, or $\psi \tilde{U} \varphi$, where ψ and φ are CTL state formulas.

The logic $\forall CTL^\star$ is a restricted subset of CTL* that allows only the universal path quantifier A. Note that since negation in CTL* can be applied only to atomic propositions, assertions of the form $\neg A\psi$, which is equivalent to $E\neg\psi$, are not possible. Thus, the logic $\forall$CTL* is not closed under negation. The logic $\forall$CTL is defined similarly, as the restricted subset of CTL that allows the universal path quantifier only. The logics $\exists$CTL* and $\exists$CTL are defined analogously, as the existential fragments of CTL* and CTL, respectively. Note that negating a $\forall$CTL* formula results in an $\exists$CTL* formula. For example, $\neg ApU(AXq)$ is equivalent to $E(\neg p)\tilde{U}(EX\neg q)$. Conversely, negating a $\exists$CTL* formula results in an $\forall$CTL* formula.

The *closure* $cl(\psi)$ of a CTL* formula ψ is the set of all state subformulas of ψ (including ψ but excluding **true** and **false**). For example, $cl(E(pU(AXq))) =$

$\{E(pU(AXq)), p, AXq, q\}$. It is easy to see that the size of $cl(\psi)$ is linear in the size of ψ. We say that a CTL* formula φ is an *U-formula* if it is of the form $A\varphi_1 U\varphi_2$ or $E\varphi_1 U\varphi_2$. The subformula φ_2 is then called the *eventuality* of φ. Similarly, φ is a *$\tilde{U}$-formula* if it is of the form $A\varphi_1 \tilde{U}\varphi_2$ or $E\varphi_1 \tilde{U}\varphi_2$. We denote by $AU(\psi)$ the set of formulas of the from $A\varphi_1 U\varphi_2$ in $cl(\psi)$. The sets $EU(\psi)$, $A\tilde{U}(\psi)$, and $E\tilde{U}(\psi)$ are defined similarly.

We define the semantics of CTL* (and its sublanguages) with respect to *fair Rabin modules* (*fair modules*, for short). A fair module $M = \langle AP, W, R, W_0, L, \alpha\rangle$ consists of a set AP of atomic propositions, a set W of states, a total transition relation $R \subseteq W \times W$, a set $W_0 \subseteq W$ of initial states, a labeling function $L : W \to 2^{AP}$, and a Rabin fairness condition α; that is, α defines a subset of W^ω (our choice of this type of fairness condition is technically motivated, as will be clarified in the sequel). For a state $w \in W$, we use $bd(w)$ to denote the branching degree of w; that is, the number of different R-successors that w has. A *computation* of a fair module is a sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0$, we have that $\langle w_i, w_{i+1}\rangle \in R$. We extend the labeling function L to computations and denote by $L(\pi)$ the word $L(w_0) \cdot L(w_1) \cdots$. For a computation π, let $inf(\pi)$ denote the set of states that repeat infinitely often in π. That is,

$$inf(\pi) = \{w : \text{ for infinitely many } i \geq 0, \text{ we have } w_i = w\}.$$

A computation of M is *fair* iff it satisfies the fairness condition α. Thus, if α is $\langle G_1, B_1\rangle, \ldots, \langle G_k, B_k\rangle\}$, then π is fair iff there exists $1 \leq i \leq k$ such that $inf(\pi) \cap G_i \neq \emptyset$ and $inf(\pi) \cap B_i = \emptyset$. In other words, iff π visits G_i infinitely often and visits B_i only finitely often. We say that a fair module is *nonempty* iff there exists a fair computation that starts at an initial state. A *module* is a fair module with no fairness condition. That is, all the computations of a module are considered fair. We denote a module by $M = \langle AP, W, R, W_0, L\rangle$.

We use $w \models \varphi$ to indicate that a state formula φ holds at state w (assuming an agreed fair module M). The relation $\models$ is inductively defined as follows (the relation $\pi \models \psi$ for a path formula ψ is the same as for ψ in LTL).

- For all w, we have that $w \models \mathbf{true}$ and $w \not\models \mathbf{false}$.
- For an atomic proposition $p \in AP$, we have $w \models p$ iff $p \in L(w)$ and $w \models \neg p$ iff $p \notin L(w)$.
- $w \models \psi \vee \varphi$ iff $w \models \psi$ or $w \models \varphi$.
- $w \models \psi \wedge \varphi$ iff $w \models \psi$ and $w \models \varphi$.
- $w \models E\psi$ iff there exists a fair computation $\pi = w_0, w_1, \ldots$ such that $w_0 = w$ and $\pi \models \psi$.
- $w \models A\psi$ iff for all fair computations $\pi = w_0, w_1, \ldots$ such that $w_0 = w$, we have $\pi \models \psi$.
- $\pi \models \varphi$ for a computation $\pi = w_0, w_1, \ldots$ and a state formula φ iff $w_0 \models \varphi$.

A fair module M satisfies a formula φ, denoted $M \models \varphi$, iff φ holds in *all* initial states of M. The problem of determining whether a given fair module M satisfies a formula φ is the *fair-model-checking* problem. The complexity of fair model checking is very well understood.

Theorem 1.

(1) [SC85, VW86] *The fair-model-checking problem for specification in LTL is PSPACE-complete. Determining whether* $M \models \varphi$ *for* φ *in LTL can be done in time* $k2^{O(l)}$ *and space* $O((\log k + l)^2)$*, where* k *is the size of* M*, and* l *is the length of* φ*.*

(2) [CES86, KV95] *The fair-model-checking problem for specification in CTL is PTIME-complete. Determining whether* $M \models \varphi$ *for* φ *in CTL can be done in time* $O(kl)$ *and space* $O(l \log^2 k)$*, where* k *is the size of* M*, and* l *is the length of* φ*.*

(3) [EL85a, KV95] *The fair-model-checking problem for specification in CTL** *is PSPACE-complete. Determining whether* $M \models \varphi$ *for* φ *in CTL** *can be done in time* $k2^{O(l)}$ *and space* $O(l(\log k + l)^2)$*, where* k *is the size of* M*, and* l *is the length of* φ*.*

Since modular model-checking with assumption φ and guarantee ψ in LTL reduce to model checking the formula $\varphi \rightarrow \psi$ [Pnu85a], it follows that determining whether M guarantees ψ under the assumption φ can be done in time $k2^{O(l+m)}$ and space $O((\log k + l + m)^2)$, where k is the size of M, l is the length of φ, and m is the length of ψ.

2.2 Simulation Relation and Composition of Modules

In the context of modular verification, it is helpful to define an order relation between fair modules [GL94]. Intuitively, the order captures what it means for a fair module M' to have "more behaviors" than a fair module M. Let $M = \langle AP, W, R, W_0, L, \alpha \rangle$ and $M' = \langle AP', W', R', W'_0, L', \alpha' \rangle$ be two fair modules for which $AP' \subseteq AP$, and let w and w' be states in W and W', respectively. A relation $H \subseteq W \times W'$ is a *simulation relation* from $\langle M, w \rangle$ to $\langle M', w' \rangle$ iff the following conditions hold:

(1) $H(w, w')$.

(2) For all s and s', we have that $H(s, s')$ implies the following:

(2.1) $L(s) \cap AP' = L(s')$.

(2.2) For every fair computation $\pi = s_0, s_1, \ldots$ in M, with $s_0 = s$, there exists a fair computation $\pi' = s'_0, s'_1, \ldots$ in M', with $s'_0 = s'$, such that for all $i \geq 0$, we have $H(s_i, s'_i)$.

A simulation relation H is a *simulation from* M *to* M' iff for every $w \in W_0$ there exists $w' \in W'_0$ such that $H(w, w')$. If there exists a simulation from M to M', we say that M *simulates* M' and we write $M \leq M'$. Intuitively, it means that the fair module M' has more behaviors than the fair module M. In fact, every possible behavior of M is also a possible behavior of M'. Note that our simulation is an extension of the classical simulation used by Milner [Mil71], where there are no fairness conditions. We sometimes relate module (with no fairness condition) with $\leq$. Then, we assume that all computations are fair, and the relation that follows coincides with the one in [Mil71].

Theorem 2. [GL94]

(1) *The simulation relation $\leq$ is a preorder (i.e., a reflexive and transitive order).*
(2) *For every M and M' such that $M \leq M'$, and for every universal branching temporal logic formula φ, $M' \models \varphi$ implies $M \models \varphi$.*

Let M and M' be two modules. The *composition* of M and M', denoted $M \| M'$, is a module that has exactly these behaviors that are joint to M and M'. Formally, if $M = \langle AP, W, R, W_0, L \rangle$ and $M' = \langle AP', W', R', W'_0, L' \rangle$, then $M \| M' = \langle AP'', W'', R'', W''_0, L'' \rangle$, where,

- $AP'' = AP \cup AP'$.
- $W'' = \{\langle w, w' \rangle : L(w) \cap AP' = L(w') \cap AP\}$.
- $R'' = \{\langle \langle w, w' \rangle, \langle s, s' \rangle \rangle : \langle w, s \rangle \in R \text{ and } \langle w', s' \rangle \in R'\}$.
- $W''_0 = (W_0 \times W'_0) \cap W''$.
- For every $\langle w, w' \rangle \in W''$, we have $L''(\langle w, w' \rangle) = L(w) \cup L'(w')$.

We also define the composition of a fair module M with a module M'. Here, $M \| M'$ is a fair module that has exactly these behaviors that are joint to M and M' and are fair in M. Formally, if $M = \langle AP, W, R, W_0, L, \alpha \rangle$ and $M' = \langle AP', W', R', W'_0, L' \rangle$, then $M \| M' = \langle AP'', W'', R'', W''_0, L'', \alpha'' \rangle$, where AP'', W'', R'', W''_0, and L'' are as in the composition of two modules, and

- $\alpha'' = \{\langle (G \times W') \cap W'', (B \times W') \cap W'' \rangle : \langle G, B \rangle \in \alpha\}$.

It is easy to see that if M and M' have n and n' states (that we assume to be disjoint), and M has m pairs in its fairness condition, then $M \| M'$ has nn' states and m pairs.

The following properties of compositions are proven in [GL94] for fair Streett modules (modules where the fairness condition is Streett), and we prove them here for modules and fair Rabin modules.

Theorem 3. *For every module M and fair Rabin modules M' and M'', the following hold.*

(1) *If $M' \leq M''$ then $M \| M' \leq M \| M''$.*
(2) *$M' \leq M' \| M'$.*

Proof: The proof is very similar to the proof for Streett modules given in [GL94]. We start with (1). Assume that $M' \leq M''$. Let H be a simulation from M' to M''. Let W be the states space of M. It is easy to see that the relation $H' = \{\langle \langle w, w' \rangle, \langle w, w'' \rangle \rangle : H(w', w'')\}$ is a simulation from $M \| M'$ to $M \| M''$. In order to prove (2), recall that the state space of $M' \| M'$ is $W' \times W'$, where W' is the state space of M'. Therefore, it is easy to see that the relation $H = \{\langle w', \langle w', w' \rangle \rangle\}$ is a simulation from M' to $M' \| M'$. □

A fair module M is a *maximal model* for an $\forall$CTL* formula φ if it allows all behaviors consistent with φ. Formally, M is a maximal model of φ if $M \models \varphi$ and for every fair module M' we have that $M' \leq M$ if $M' \models \varphi$. Note that by the preceding theorem, if $M' \leq M$, then $M' \models \varphi$. Thus, M_φ is a maximal model for φ if for every fair module M, we have that $M \leq M_\varphi$ iff $M \models \varphi$.

Theorem 4. [GL94] *For every $\forall$CTL formula φ, there exists a maximal model M_φ of size $2^{O(|\varphi|)}$.*

2.3 Büchi Word Automata

Given an alphabet Σ, an *infinite word over Σ* is an infinite sequence $w = w_1 \cdot w_2 \cdot w_3 \cdots$ of letters in Σ. A *Büchi automaton over infinite words* is $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is an acceptance condition (a condition that defines a subset of Q^ω). Intuitively, $\delta(q, \sigma)$ is the set of states that $\mathcal{A}$ can move into when it is in state q and it reads the letter σ. Since $\mathcal{A}$ may have several initial states and since the transition function may specify many possible transitions for each state and letter, $\mathcal{A}$ may be *nondeterministic*. If $|Q^0| = 1$ and δ is such that for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\delta(q, \sigma)| = 1$, then $\mathcal{A}$ is a *deterministic* automaton.

Given an input infinite word $w = c_0 \cdot c_1 \cdots \in \Sigma^\omega$, a *run* of $\mathcal{A}$ on w can be viewed as a function $r : \mathbb{N} \to Q$ where $r(0) \in Q^0$ (i.e., the run starts in one of the initial states) and for every $i \geq 0$, we have $r(i+1) \in \delta(r(i), c_i)$ (i.e., the run obeys the transition function). Each run r induces a set $inf(r)$ of states that r visits *infinitely often*. Formally,

$$inf(r) = \{q \in Q \ : \text{ for infinitely many } i \geq 0, \text{ we have } r(i) = q\}.$$

As Q is finite, it is guaranteed that $Inf(r) \neq \emptyset$. The run r *accepts* w iff $Inf(r) \cap F \neq \emptyset$. Note that a nondeterministic automaton can have many runs on w. In contrast, a deterministic automaton has a single run on w. An automaton $\mathcal{A}$ accepts an input word w iff there exists a run r of $\mathcal{A}$ on w such that r accepts w. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of infinite words that $\mathcal{A}$ accepts. Thus, each word automaton defines a subset of Σ^ω.

Computations of a fair module can be viewed as infinite words over the alphabet 2^{AP}. According to this view, each fair module corresponds to a language over the alphabet 2^{AP} and can be associated with an automaton. A similar connection has been established between LTL formulas and Büchi automata:

Theorem 5. [VW94] *Given an LTL formula ψ, there is a Büchi automaton $\mathcal{A}_\psi = \langle 2^{AP}, Q, \delta, Q_0, F \rangle$, with $2^{O(|\psi|)}$ states, such that $\mathcal{L}(\mathcal{A}_\psi)$ is exactly the set of computations satisfying ψ.*

2.4 Branching Modular Model-Checking for ∀CTL

In modular verification, one uses assertions of the form $\langle\varphi\rangle M\langle\psi\rangle$ to specify that whenever M is part of a system satisfying the universal branching temporal logic formula φ, the system satisfies the universal branching temporal logic formula ψ too. Formally, $\langle\varphi\rangle M\langle\psi\rangle$ holds if $M\|M' \models \psi$ for all M' such that $M\|M' \models \varphi$. Here φ is an assumption on the behavior of the system and ψ is the guarantee on the behavior of the fair module. Assume-guarantee assertions are used in modular proof rules of the following form:

$$\left.\begin{array}{l}\langle\varphi_1\rangle M_1\langle\psi_1\rangle\\ \langle\mathbf{true}\rangle M_1\langle\varphi_1\rangle\\ \langle\varphi_2\rangle M_2\langle\psi_2\rangle\\ \langle\mathbf{true}\rangle M_2\langle\varphi_2\rangle\end{array}\right\}\ \langle\mathbf{true}\rangle M_1\|M_2\langle\psi_1\wedge\psi_2\rangle$$

Thus, a key step in modular verification is checking that assume-guarantee assertions hold, which we call the *branching modular model-checking problem*. It was shown in [GL94] that the maximal-model technique yields a solution to the modular model-checking problem.

Theorem 6. [GL94] *For all ∀CTL formulas φ and ψ, and for every fair module M, we have that $\langle\varphi\rangle M\langle\psi\rangle$ iff $M\|M_\varphi \models \psi$.*

Thus, modular model checking for ∀CTL is reducible to standard model checking for ∀CTL. Combining this with Theorems 4 and 1, we get the following complexity results.

Theorem 7.

(1) *Determining whether $\langle\varphi\rangle M\langle\psi\rangle$, for φ and ψ in ∀CTL, can be done in time $km2^{O(l)}$ and space $O(m(\log k+l)^2)$, where k is the size of M, l is the length of φ, and m is the length of ψ.*

(2) *Determining whether $\langle\varphi\rangle M\langle\psi\rangle$, for φ in ∀CTL and ψ in ∀CTL*, can be done in time $k2^{O(l+m)}$ and space $O(m(\log k+l+m)^2)$, where k is the size of M, l is the length of φ, and m is the length of ψ.*

A comparison of Theorem 7 with Theorem 1 shows that the complexity of branching modular model checking with assumptions in ∀CTL is higher than the complexity of CTL model checking, but is comparable to the complexity of LTL model checking. How do LTL and ∀CTL compare from the expressiveness point of view? While ∀CTL and LTL have incomparable expressive power, in practice one often finds LTL to be more expressive, as the specifications that can be expressed in ∀CTL but not in LTL rarely arise in practical settings. Since the complexity of CTL model checking is lower than that of LTL model checking (Theorem 1), we are often willing to settle for the lower expressiveness of CTL; that is, we are willing to verify the design with respect to weaker specifications, with the hope that design errors will be discovered in the process. For example, a significant portion of verified properties are *safety* properties that can be expressed as $AG\varphi$, where φ is a propositional formula.

While we might be willing to settle for a weak guarantee, we cannot, however, settle for weak assumptions. In many cases one needs to adopt rather strong assumptions in order to verify even a very weak guarantee. Very often such assumptions are simply not expressible in ∀CTL. For example, the assumption $AFG\varphi$, where φ is a propositional formula, cannot be expressed in ∀CTL[EH86]. Thus, ∀CTL is simply not expressive enough as a specification language for modular model checking. In the next section we will consider using ∀CTL* and LTL as specification languages for assumptions in modular model checking.

3 Modular Model-Checking for ∀CTL* and LTL

3.1 Maximal Models

We now consider assumptions in ∀CTL*, and we wish to construct maximal models for such assumptions. Unfortunately, the tableau-based technique that was used in the proof of Theorem 4 does not seem to extend to ∀CTL*. Indeed, while the satisfiability problem for CTL can be solved using tableau-based technique [EH85], the satisfiability problem for CTL* requires sophisticated automata-theoretic techniques [EJ88]. We now show that these automata-theoretic techniques can be used to construct maximal models for ∀CTL* formulas.

Theorem 8. *For every ∀CTL* formula φ, there exists a maximal model M_φ of size $2^{2^{O(|\varphi|)}}$.*

Proof: For a ∀CTL* formula φ, let $sf(\varphi)$ denote the set of state subformulas of φ. Given φ, let $\forall(\varphi) \subseteq sf(\varphi)$ denote the set of all the state subformulas of φ of the form $A\xi$. Let $\mathcal{A}_{\forall(\varphi)}$ be a Büchi ω-automaton over $\Sigma = 2^{sf(\varphi)}$ such that $\mathcal{A}_{\forall(\varphi)}$ accepts an infinite word $\pi = w_0, w_1, \ldots$ iff there exists a suffix $w_i, w_{i+1}, \ldots$ of π and a formula $A\xi \in \forall(\varphi)$ such that $A\xi \in w_i$ and $w_i, w_{i+1}, \ldots$ does not satisfy ξ. Technically, $\mathcal{A}_{\forall(\varphi)}$ nondeterministically guesses a location i and a formula $A\xi$ and then follows the Büchi ω-automaton of $\neg\xi$. Consequently, if $w_i, w_{i+1}, \ldots$ does not satisfy ξ, the automaton $\mathcal{A}_{\forall(\varphi)}$ would accept π. By Theorem 5, such $\mathcal{A}_{\forall(\varphi)}$ of size $2^{O(|\varphi|)}$ exists. Note that though ξ is a path formula of a branching temporal logic, we interpret it here over linear sequences. Since these sequences are labeled with all the state subformulas of ξ, this causes no difficulty, as we can regard the state subformulas of ξ as atomic propositions and regard ξ as a linear temporal logic formula.

We now take $\mathcal{A}_{\forall(\varphi)}$ and co-determinize it. The resulted automaton, called $\overline{\mathcal{A}_{\forall(\varphi)}}$, is a deterministic Rabin automaton that accepts exactly all the words $\pi = w_0, w_1, \ldots$ for which if a state w_i is labeled with some $A\xi \in \forall(\varphi)$, then ξ is satisfied in the suffix $w_i, w_{i+1}, \ldots$ of π. By [Saf89], the automaton $\overline{\mathcal{A}_{\forall(\varphi)}}$ is of size $2^{2^{O(|\varphi|)}}$.

For a set $s \subseteq sf(\varphi)$, we say that s is *consistent* iff the following four conditions hold:

1. For every $p \in AP$, if $p \in s$, then $\neg p \notin s$.

2. For every $p \in AP$, if $\neg p \in s$, then $p \notin s$.
3. For every $\varphi_1 \wedge \varphi_2 \in s$, we have that $\varphi_1 \in s$ and $\varphi_2 \in s$.
4. For every $\varphi_1 \vee \varphi_2 \in s$, we have that $\varphi_1 \in s$ or $\varphi_2 \in s$.

Let $c(\varphi)$ denote the set of all consistent subsets of $sf(\varphi)$. Consider the module

$$M = \langle AP, c(\varphi), c(\varphi) \times c(\varphi), W_0, L \rangle,$$

where the initial set W_0 includes all states $w \in c(\varphi)$ for which $\varphi \in w$ (note that if φ is satisfiable, the set W_0 is not empty), and for every $w \in c(\varphi)$, we have that $L(w) = w \cap AP$. That is, M is more general than any model of φ, yet, it is not necessarily a model of φ. To make it a maximal model, we take the product of M with $\overline{\mathcal{A}_{\forall(\varphi)}}$ as follows. Let $\overline{\mathcal{A}_{\forall(\varphi)}} = \langle \Sigma, Q, \delta, q_0, \beta \rangle$, where $\beta = \{\langle G_1, B_1 \rangle, \ldots, \langle G_k, B_k \rangle\}$. Then, $M_\varphi = \langle AP, c(\varphi) \times Q, R, W_0 \times \{q_0\}, L', \beta' \rangle$, where R, L', and β' are defined as follows.

- $R = \{\langle \langle w, q \rangle, \langle w', q' \rangle \rangle : \delta(q, w) = q'\}$.
- For all $w \in c(\varphi)$ and $q \in Q$, we have $L'(\langle w, q \rangle) = L(w)$.
- $\beta' = \{\langle c(\varphi) \times G_1, c(\varphi) \times B_1 \rangle, \ldots, \langle c(\varphi) \times G_k, c(\varphi) \times B_k \rangle\}$.

We now prove the correctness of our construction. That is, we show that $M_\varphi \models \varphi$ and that for all fair modules M, we have $M \models \varphi$ only if $M \leq M_\varphi$ We first prove that $M_\varphi \models \varphi$. More precisely, we prove that for every reachable state $\langle w, q \rangle \in c(\varphi) \times Q$, and for every formula $\psi \in w$, we have that $\langle w, q \rangle \models \psi$. The proof proceeds easily by induction on the structure of ψ. In particular, satisfaction of formulas of the form $A\xi$ follows from the product with $\overline{\mathcal{A}_{\forall(\varphi)}}$. To see this, consider a state $\langle w, q \rangle$ and a formula $A\xi \in w$. Let $\langle w_1, q_1 \rangle, \langle w_2, q_2 \rangle, \ldots$ be a fair computation of M_φ that starts in $\langle w, q \rangle$; that it, $\langle w_1, q_1 \rangle = \langle w, q \rangle$. By the definition of R and β', the sequence $w_1, w_2, \ldots$ is a suffix of a word accepted by $\overline{\mathcal{A}_{\forall(\varphi)}}$. Hence, for all formulas of the form $A\xi' \in w$, the computation $w_1, w_2, \ldots$ satisfies ξ'. Thus, in particular, $w_1, w_2, \ldots$ satisfies ξ.

Consider now a fair module $M = \langle AP, W_M, R_M, W_M^0, L_M, \alpha_M \rangle$ and assume that $M \models \varphi$. We show a simulation H from M to M_φ. For every state $w \in W_M$, define $f(w)$ to be the set in $c(\varphi)$ of state formulas that are true in w. The simulation H is the smallest set that satisfies the following:

- For every $w \in W_M^0$, we have $H(w, \langle f(w), q_0 \rangle)$.
- For every w_1, w_2 in W_M and $\langle f(w_1), q_1 \rangle \in c(\varphi) \times Q$ such that $\langle w_1, w_2 \rangle \in R_M$ and $H(w_1, \langle f(w_1), q_1 \rangle)$, we have $H(w_2, \langle f(w_2), \delta(q_1, f(w_1)) \rangle)$.

We prove that H is indeed a simulation from M to M_φ. That is, we prove that for all $w \in W_M^0$, there exists $w' \in W_0 \times \{q_0\}$ such that H is a simulation relation from $\langle M, w \rangle$ to $\langle M_\varphi, w' \rangle$. Consider a state $w \in W_M^0$. Since $M \models \varphi$, then, by the definition of $f(w)$ and W_0, we have $\langle f(w), q_0 \rangle \in W_0 \times \{q_0\}$, and hence, by the definition of H, we have $H(w, \langle f(w), q_0 \rangle)$. Now, let $w \in W_M$ and $\langle f(w), q \rangle \in c(\varphi) \times Q$ be such that $H(w, \langle f(w), q \rangle)$. By the definition of H, all the pairs in H are of this form. By the definition of L', we have that $L'(\langle f(w), q \rangle) = L_M(w)$. So, the first requirement on pairs in a simulation holds. For the second requirement,

assume $H(w, \langle f(w), q\rangle)$ and let $\pi = w_0, w_1, \ldots$ be a fair computation in M with $w_0 = w$. Consider the computation $\pi' = \langle f(w), q\rangle, \langle f(w_1), q_1\rangle, \ldots$ where $q_1 = \delta(q, f(w))$ and for every $i \geq 1$, we have $q_{i+1} = \delta(q_i, f(w_i))$. By the definition of H, we have that for all $i \geq 1$ we have $H(w_i, \langle f(w_i), q_i\rangle)$. So, it remains to show that π' is fair in M_φ. Since $M \models \varphi$ and π is fair, then for each state w_i and formula $A\xi \in \forall(\varphi)$ such that $A\xi \in f(w_i)$, we have that ξ is satisfied in $w_i, w_{i+1}, \ldots$. Thus, $q_i, q_{i+1}, \ldots$ is an accepting run of $\overline{\mathcal{A}_{\forall(\varphi)}}$ with q_i as an initial state over $w_i, w_{i+1}, \ldots$. Therefore, by the definition of β', the computation π' is fair. □

We can now obtain an alternative proof of Theorem 4.

Theorem 9. *For every $\forall$CTL formula φ, there exists M_φ of size $2^{O(|\varphi|)}$.*

Proof: Exactly as for $\forall$CTL*. Here, however, $\mathcal{A}_{\forall(\varphi)}$ is of size $O(|\varphi|)$, and hence $\overline{\mathcal{A}_{\forall(\varphi)}}$ is of size $2^{O(|\varphi|)}$. □

3.2 The Branching Modular Model-Checking Problem

We now show the maximal-model technique enables us to reduce modular model checking to standard model checking.

Theorem 10. *For all $\forall$CTL* formulas φ and ψ, and for every fair module M, we have that $\langle\varphi\rangle M\langle\psi\rangle$ iff $M \| M_\varphi \models \psi$.*

Proof: Assume first that $\langle\varphi\rangle M\langle\psi\rangle$. Thus, whenever M is part of a system satisfying φ, the system satisfies ψ too. Since $M_\varphi \models \varphi$ and $M \| M_\varphi \leq M_\varphi$, we have, by Theorem 2 (2), that $M \| M_\varphi$ satisfies φ. Consequently, $M \| M_\varphi$ satisfies ψ.

Assume now that $M \| M_\varphi \models \psi$ and let $M \| M'$ be such that $M \| M' \models \varphi$. Then, $M \| M' \leq M_\varphi$, which implies, by Theorem 3 (1), that $M \| M \| M' \leq M \| M_\varphi$. Thus, by Theorem 3 (2), $M \| M' \leq M \| M_\varphi$ and therefore, by Theorem 2 (2), $M \| M' \models \psi$. Hence, $\langle\varphi\rangle M\langle\psi\rangle$. □

It follows that branching modular model checking can be reduced to fair model checking. In Theorem 11 below we apply the reduction to the logics $\forall$CTL and $\forall$CTL*. We also show that the upper bounds that follow are tight.

Theorem 11.

(1) *The branching modular model-checking problem for $\forall$CTL is PSPACE-complete.*
(2) *The branching modular model-checking problem for $\forall$CTL* is EXPSPACE-complete.*
(3) *Determining whether $\langle\varphi\rangle M\langle\psi\rangle$, for φ and ψ in $\forall$CTL*, can be done in time $k2^{O(m)+2^{O(l)}}$ in space $O(m(m + \log k + 2^{O(l)})^2)$, where l is the length of φ, k is the size of M, and m is the length of ψ.*

(4) *Determining whether* $\langle\varphi\rangle M\langle\psi\rangle$*, for* φ *in* $\forall CTL^*$ *and* ψ *in* $\forall CTL$*, can be done in time* $km2^{2^{O(l)}}$ *in space* $O(m(\log k + 2^{O(l)})^2)$*, where* l *is the length of* φ*,* k *is the size of* M*, and* m *is the length of* ψ*.*

Proof: We start with the upper bounds. By Theorem 10, the problem of determining whether $\langle\varphi\rangle M\langle\psi\rangle$ is reducible to model checking of ψ in $M\|M_\varphi$. The upper bounds bounds then follow from Theorems 1 and 8.

We now turn to the lower bounds. For both bounds, we do a reduction from the implication problem. The implication problem is defined as follows: given two formulas φ and ψ, does φ imply ψ (denoted $\varphi \rightarrow \psi$)? Namely, does ψ hold in every fair module in which φ holds? For a set AP of atomic propositions, let M_{AP} be the maximal model over AP. That is,

$$M_{AP} = \langle AP, 2^{AP}, 2^{AP} \times 2^{AP}, 2^{AP}, L, \{\langle 2^{AP}, \emptyset\rangle\}\rangle,$$

where for all $w \in 2^{AP}$ we have that $L(w) = w$. Let φ and ψ be $\forall$CTL* formulas over a set AP of atomic propositions. For every fair module M, the fair modules M and $M\|M_{AP}$ simulate each other. Hence, for every $\forall$CTL* formula φ over AP we have that $M\|M_{AP} \models \varphi$ iff $M \models \varphi$. Thus, the implication $\varphi \rightarrow \psi$ holds iff $\langle\varphi\rangle M_{AP}\langle\psi\rangle$. The complexity of the reduction depends on the size of M_{AP}. We will show that for both $\forall$CTL and $\forall$CTL*, the size of M_{AP} is fixed.

To prove the PSPACE lower bound for the implication problem for $\forall$CTL, we prove a PSPACE lower bound for its satisfiability problem. The result then follows since the formula φ is satisfiable if and only if φ does not imply $A\mathbf{false}$. We prove hardness in PSPACE for $\forall$CTL satisfiability by a reduction from LTL satisfiability, proved to be PSPACE-hard in [SC85]. Given an LTL formula ξ, let ξ_A be the $\forall$CTL formula obtained from ξ by preceding each temporal operator with the path quantifier A. For example, if $\xi = FXp$ then $\xi_A = AFAXp$. It is easy to see that ξ is satisfiable iff ξ_A is satisfiable. Indeed, a computation that satisfies ξ can be viewed as a fair module satisfying ξ_A. For the second direction, assume that ξ_A is satisfiable in some fair module M. Consider a fair computation π of M. We can view π as a fair module of branching degree 1. Clearly, π simulates M, and thus, it satisfies ξ_A as well. Also, since its branching degree is 1, the computation π also satisfies ξ. Thus, ξ is satisfiable. The satisfiability problem for LTL is PSPACE-hard already for formulas with a fixed number of atomic propositions. The PSPACE-hardness proof in [SC85] uses temporal formulas with an unbounded number of atomic propositions. Nevertheless, By using a Turing machine M that accepts a PSPACE-complete language, it is possible to bound the number of atomic propositions used to the size of the working alphabet of M. Since it is possible to encode the truth values of m atomic propositions in one state by the truth values of a single atomic proposition along $\log m$ states, it follows that satisfiability of temporal formulas with a single atomic proposition is also PSPACE-hard.It follows that the implication problem for $\forall$CTL is PSPACE-hard already for formulas with a fixed number of atomic propositions. Thus, the size of M_{AP} in our reduction is fixed.

To prove the EXPSPACE lower bound for the implication problem for $\forall$CTL*, we do a reduction from the problem whether an exponential-space deterministic

Turing machine T accepts an input word x. That is, given T and x, we construct two $\forall$CTL* formulas φ and ψ such that T accepts x iff φ does not imply ψ. In fact we prove a stronger lower bound. Given T and x, we construct an LTL formula ξ and an $\exists$CTL formula θ such that the length of ξ is polynomial in the size of T and the length of x, the length of θ is fixed, and T accepts an input word x iff the formula $A\xi \wedge \theta$ is satisfiable. Thus, taking $\varphi = A\xi$ and $\psi = \neg\theta$, we have that T accepts x iff the implication $\varphi \rightarrow \psi$ does not hold. Thus, the branching modular model-checking problem is EXPSPACE-complete even for an assumption of the form $A\xi$, where ξ is an LTL formula, a fixed fair module, and an $\forall$CTL guarantee. For details see [KV95]. □

Note that the crucial factor in the complexity of the branching modular model checking problem is the assumption part of the specification. Indeed, the lower bounds given in Theorem 11 remains true even is we fix the guarantee part of the specification. This suggests that modular model checking in the branching temporal framework can be practical only for very small assumptions. Indeed, in many examples the assumptions do tend to be of a very small size [Jos87b, Jos89, GL94], see also [AL93]. We will come back to this point in Section 4.

3.3 The Linear-Branching Modular Model-Checking Problem

The modular proof rule in the preceding section uses branching assumptions and guarantees. As mentioned in the introduction, there is another approach in which the assumption is a linear temporal formula, while the guarantee is a branching temporal formula. In this approach, the assumption in the assume-guarantee pair concerns the interaction of the module with its environment along each computation, and is therefore more naturally expressed in a linear temporal logic. We denote this kind of assertion by $[\varphi]M\langle\psi\rangle$. The meaning of such an assertion is that the branching temporal formula ψ holds in the computation tree that consists of all computations of the program that satisfy the linear temporal formula φ.

The idea is to use assume-guarantee assertions in modular proof rules of the following form [Jos87a, Jos87b, Jos89]:

$$\left.\begin{array}{l} [\varphi_2]M_1\langle\psi_1\rangle \\ [\mathbf{true}]M_1\langle br(\varphi_1)\rangle \\ [\varphi_1]M_2\langle\psi_2\rangle \\ [\mathbf{true}]M_1\langle br(\varphi_2)\rangle \end{array}\right\} [\mathbf{true}]M_1 \| M_2\langle\psi_1 \wedge \psi_2\rangle$$

where $br(\varphi)$ is a branching version (it is an $\forall$CTL formula) of the LTL formula φ; see above references for details. Verifying assertions of the form $[\varphi]M\langle\psi\rangle$ is called the *linear-branching modular model-checking problem.*

In order to define the linear-branching model checking problem formally, we define *extended modules.* An extended module $\langle M, P\rangle$ is a module $M = \langle AP, W, R, W_0, L\rangle$ extended by a language $P \subseteq (2^{AP})^\omega$. We regard P as a fairness condition: a computation π of M is fair iff $L(\pi) \in P$. Unlike the Rabin fairness

condition, fairness of a computation π with respect to P cannot be determined by the fairness of any of π's suffixes. Therefore, in order to define the semantics of CTL* formulas with respect to extended modules, we first associate with each module M a *tree module* M^t. Intuitively, M^t is obtained by unwinding M to an infinite tree. Let $M = \langle AP, W, R, W_0, L\rangle$. A *partial path* ζ in M is a finite prefix, $w_0, w_1, \ldots, w_k$, of a computation in M, where $w_0 \in W_0$. We denote the set of partial paths of M by $ppath(M)$. The tree module of M is $M^t = \langle AP, ppath(M), R^t, \{w_0\}, L^t\rangle$, where for every partial path $\zeta = w_0, \ldots, w_k \in ppath(M)$, we have

- $R^t(\zeta, \zeta')$ iff there exists $w_{k+1} \in W$ such that $\zeta' = w_0, \ldots, w_k, w_{k+1}$ and $R(w_k, w_{k+1})$. That is, the partial path ζ' extends the partial path ζ by a single transition in M.
- $L^t(\zeta) = L(w_k)$.

Note that M^t is indeed a tree; every state has a unique predecessor. A computation $\zeta_0, \zeta_1, \ldots$ in M^t is an *anchored path* iff ζ_0 is in W_0.

The semantics of CTL* with respect to tree modules extended by a fairness condition $P \subseteq (2^{AP})^\omega$ is defined as the usual semantics of CTL*, with path quantification ranging only over anchored paths that are labeled by a word in P. Thus, for example,

- $\zeta \models E\xi$ if there exists an anchored path $\pi = \zeta_0, \ldots, \zeta_i, \zeta_{i+1}, \ldots$ of M^t and $i \geq 0$ such that $L(\pi) \in P$, $\zeta_i = \zeta$, and $\pi^i \models \xi$.
- $\zeta \models A\xi$ if for all anchored paths $\pi = \zeta_0, \ldots, \zeta_i, \zeta_{i+1}, \ldots$ of M^t and $i \geq 0$ such that $L(\pi) \in P$ and $\zeta_i = \zeta$, we have $\pi^i \models \xi$.

Note that by defining the truth of formulas on the nodes of the computation tree M^t, we guarantee that only one path leads to each node. The extended tree module $\langle M^t, P_\varphi\rangle$ satisfies a formula ψ iff $\{w_0\} \models \psi$. We say that $\langle M, P\rangle \models \psi$ iff $\langle M^t, P\rangle \models \psi$. Now, $[\varphi]M\langle\psi\rangle$ holds iff $\langle M, P_\varphi\rangle \models \psi$, where P_φ is the set of all that computations that satisfy φ.[3].

We first show that when the language P is given by a deterministic Rabin automaton, we can translated the extended modules $\langle M, P\rangle$ to an equivalent fair module.

Lemma 12. *Let $\langle M, P\rangle$ be an extended module and let $\mathcal{A}_P$ be a deterministic Rabin automaton such that $\mathcal{L}(\mathcal{A}_P) = P$. We can construct a fair module M' such that for every CTL* formula ψ, we have that $\langle M, P\rangle \models \psi$ iff $M' \models \psi$. Moreover, $M' \leq M \| M'$.*

Proof: Let $M = \langle AP, W, R, W_0, L\rangle$ and $\mathcal{A}_P = \langle 2^{AP}, Q, \delta, q_0, F\rangle$. We define $M' = \langle AP, W \times Q, R', W_0 \times \{q_0\}, L', \alpha\rangle$, where

- $R'(\langle w, q\rangle, \langle w', q'\rangle)$ iff $R(w, w')$ and $\delta(q, L(w)) = q'$.

[3] We note that the formal definitions of $[\varphi]M\langle\psi\rangle$ in [Jos87a, Jos87b, Jos89] apply only to restricted linear temporal assumptions and involve a complicated syntactic construction.

- $L'(\langle w, q\rangle) = L(w)$.
- $\alpha = \{\langle W \times G, W \times B\rangle : \langle G, B\rangle \in F\}$.

We prove that $\langle M^t, P\rangle$ and M' satisfy the same CTL* formulas. For a state $\zeta = w_0, \ldots, w_k \in \mathit{ppath}(M)$, we denote by $\mathit{last}(\zeta)$ the state $w_k \in W$, and denote by $\sigma(\zeta)$ the finite word $L(w_0) \cdots L(w_k) \in (2^{AP})^*$. Also, for a finite word $\sigma \in (2^{AP})^*$, let $\delta(q_0, \sigma)$ be the state that $\mathcal{A}_P$ reaches after reading σ.

The fact that every state in M^t is associated with a single partial path of M enables us to relate the states of M^t with the states of M'. Formally, we claim the following. For every CTL* formula ψ and state ζ in $\langle M^t, P\rangle$, we have that $\zeta \models \psi$ in $\langle M^t, P\rangle$ iff $\langle \mathit{last}(\zeta), \delta(q_0, \sigma(\zeta))\rangle \models \psi$ in M'. In particular, $\{w_0\} \models \psi$ in $\langle M^t, P\rangle$ iff $\langle w_0, q_0\rangle \models \psi$ in M'. The proof proceeds by induction on the structure of ψ. The interesting case is $\psi = A\xi$ or $\psi = E\xi$, for a path formula ξ. Let $\psi = A\xi$. Assume first that $\zeta \models \psi$ in $\langle M^t, P\rangle$. Then, for every anchored path $\pi = \zeta_0, \ldots, \zeta_i, \zeta_{i+1}, \ldots$ of M^t such that $L(\pi) \in P$ and $\zeta_i = \zeta$, we have that $\pi^i \models \xi$ in $\langle M^t, P\rangle$. Consider a fair computation $\rho = \langle w_0, q_0\rangle, \ldots, \langle w_i, q_i\rangle, \langle w_{i+1}, q_{i+1}\rangle, \ldots$ in M' for which $\langle w_i, q_i\rangle = \langle \mathit{last}(\zeta), \delta(q_0, \sigma(\zeta))\rangle$. Let $\pi = \zeta_0, \ldots, \zeta_i, \zeta_i \cdot w_{i+1}, \zeta_i \cdot w_{i+1} \cdot w_{i+2}, \ldots$ be the anchored path in M^t that corresponds to ρ. Since ρ is fair, $L(\pi) \in P$. Hence, $\zeta_i, \zeta_i \cdot w_{i+1}, \zeta_i \cdot w_{i+1} \cdot w_{i+2}, \ldots$ satisfies ξ. Then, by the induction hypothesis, $\langle w_i, q_i\rangle, \langle w_{i+1}, q_{i+1}\rangle, \ldots$ satisfies ξ as well and we are done. The proof for $\psi = E\xi$ is similar.

It is left to see that $M' \leq M \| M'$. Recall that the state space of $M \| M'$ is $W \times W \times Q$. Intuitively, since M' is a restriction of M, composing M' with M does not restrict it further. Formally, it is easy to see that the relation

$$H = \{\langle \langle w, q\rangle, \langle w, w, q\rangle\rangle : \langle w, q\rangle \in W \times Q\}$$

is a fair simulation from M' to $M \| M$.

□

To solve the the linear-branching model-checking problem, we show that the branching modular framework is more general than the linear-branching modular framework. Thus, the algorithms discussed in Section 3.2 are applicable also here.

Theorem 13. *For every LTL formula φ, fair module M, and a $\forall$CTL* formula ψ, we have that $\langle A\varphi\rangle M\langle\psi\rangle$ iff $[\varphi]M\langle\psi\rangle$.*

Proof: Given φ, M, and ψ, assume first that $[\varphi]M\langle\psi\rangle$ holds. Let P_φ be the set of computations satisfying φ. Thus, the extended module $\langle M, P_\varphi\rangle$ satisfies ψ. Consider the composition $M \| M'$ of M with some module M'. Recall that for M and M' with state spaces W and W', respectively, the state space W'' of $M \| M'$ consists of all the pairs $\langle w, w'\rangle$ for which w and w' agree on the labels of the atomic propositions joint to M and M'. Then, the relation

$$H = \{\langle \langle w, w'\rangle, w\rangle : \langle w, w'\rangle \in W''\}$$

is a simulation relation from $M \| M'$ to M. It is easy to see that H is also a simulation relation from $\langle M \| M', P_\varphi\rangle$ to $\langle M, P_\varphi\rangle$. Hence, $\langle M \| M', P_\varphi\rangle$ satisfies

ψ as well. Let M' be such that $M\|M' \models A\varphi$. That is, all the computations in $M\|M'$ satisfy φ. Hence, the identity relation is a simulation relation from $M\|M'$ to $\langle M\|M', P_\varphi\rangle$. Therefore, as $\langle M\|M', P_\varphi\rangle$ satisfies ψ, so does $M\|M'$, and we are done.

Assume now that $\langle A\varphi\rangle M\langle\psi\rangle$ holds. Let $M_{A\varphi}$ be the maximal model of $A\varphi$. Since $M_{A\varphi} \models A\varphi$ and $M\|M_{A\varphi} \leq M_{A\varphi}$, we have that $M\|M_{A\varphi} \models A\varphi$ and therefore, by the assumption, $M\|M_{A\varphi} \models \psi$. Let M' be the fair module equivalent to $\langle M, P_\varphi\rangle$, as defined in Lemma 12. That is, $\langle M, P_\varphi\rangle$ and M' satisfy the same CTL* formulas. Since $\langle M, P_\varphi\rangle \models A\varphi$, we have that $M' \models A\varphi$. Hence, $M' \leq M_{A\varphi}$ and therefore, by Theorem 2 (2), $M\|M' \leq M\|M_{A\varphi}$. Hence, as $M\|M_{A\varphi} \models \psi$, we have that $M\|M' \models \psi$. Since, by Lemma 12, $M' \leq M'\|M$, it follows that M' satisfies ψ as well. Hence, $\langle M, P_\varphi\rangle$ also satisfies ψ and we are done. □

It is known that the ∀CTL formula $AFAGp$ is not equivalent to any formula of the form $A\xi$, where ξ is an LTL formula. Thus, the branching modular framework is strictly more expressive than the linear-branching modular framework, with no increase in worst-case computational complexity (we have seen, in the proof of Theorem 11, that the EXPSPACE lower bound holds already for assumptions of the form $A\xi$ for an LTL formula ξ).

4 Concluding Remarks

The results of the paper indicate that modular model-checking for general universal or linear assumptions is rather intractable. Our results provide an *a posteriori* justification for Josko's restriction on the linear temporal assumption [Jos87a, Jos87b, Jos89]. Essentially, for a restricted linear temporal assumption φ, one can get a more economical automata-theoretic construction of the maximal model associated with the CTL* formula $A\varphi$ (exponential rather than doubly exponential). We note that it is argued in [LP85] that an exponential time complexity in the size of the specification might be tolerable in practical applications.

There is, however, a fundamental difference between the impact that the guarantee and the assumption have on the complexity of model checking. Both assumption and guarantee are often given as a conjunction of formulas. That is, we are often trying to verify assume-guarantee assertions of the form

$$\langle\varphi_1 \wedge \ldots \wedge \varphi_l\rangle M\langle\psi_1 \wedge \ldots \wedge \psi_m\rangle.$$

Each conjunct expresses a certain property about a module or its environment. Typically, each conjunct is of a rather small size. While it is possible to decompose the guarantee and reduce the problem to verifying assertions of the form $\langle\varphi_1 \wedge \ldots \wedge \varphi_l\rangle M\langle\psi\rangle$, where ψ is of small size, it is not possible in general to decompose the assumption in a similar fashion. Thus, it may seem that in trying to employ modular verification in order to overcome the state-explosion problem, we are merely replacing it with the *assumption-explosion problem*.

This observation provides a justification to the approach taken in [CLM89] to avoid the assume-guarantee paradigm. Instead of describing the interaction of the module by an LTL formula, it is proposed there to model the environment by *interface* processes. As is shown there, these processes are typically much simpler than the full environment of the module. By composing a module with its interface processes and then verifying properties of the composition, it can be guaranteed that these properties will be preserved at the global level.

Acknowledgment: We thank Martin Abadi, Orna Grumberg, and Pierre Wolper for helpful suggestions and discussions.

References

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[ASSS94] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. In *Proc. 6th Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 324–337, Stanford, CA, June 1994. Springer-Verlag.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CG87] E.M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. In *Annual Review of Computer Science*, volume 2, pages 269–290, 1987.

[CGL93] E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer-Verlag, 1993.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In R. Parikh, editor, *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1989.

[DDGJ89] W. Damm, G. Döhmen, V. Gerstner, and B. Josko. Modular verification of Petri nets: the temporal logic approach. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proceedings of REX Workshop)*, volume 430 of *Lecture Notes in Computer Science*, pages 180–207, Mook, The Netherlands, May/June 1989. Springer-Verlag.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of reduced models for checking fragments of CTL. In *Proc. 5th Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag, June 1993.

[EH85] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30:1–24, 1985.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 368–377, White Plains, October 1988.

[EL85a] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, January 1985.

[EL85b] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, North Holywood, 1985. Western Periodicals Company.

[EL87] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[GL91] O. Grumberg and D.E. Long. Model checking and modular verification. In *Proc. 2nd Conferance on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, 1991.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[Jon83] C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *Information Processing 83: Proc. IFIP 9th World Congress*, pages 321–332. IFIP, North-Holland, 1983.

[Jos87a] B. Josko. MCTL – an extension of CTL for modular verification of concurrent systems. In *Temporal Logic in Specification, Proceedings*, volume 398 of *Lecture Notes in Computer Science*, pages 165–187, Altrincham,UK, April 1987. Springer-Verlag.

[Jos87b] B. Josko. Model checking of CTL formulae under liveness assumptions. In *Proc. 14th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 267 of *Lecture Notes in Computer Science*, pages 280–289. Springer-Verlag, July 1987.

[Jos89] B. Josko. Verifying the correctness of AADL modules using model chekcing. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (Proceedings of REX Workshop)*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400, Mook, The Netherlands, May/June 1989. Springer-Verlag.

[JT95] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 262–276, Aarhus, Denmark, May 1995. Springer-Verlag.

[KV95] O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conferance on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 408–422, Philadelphia, August 1995. Springer-Verlag.

[Lam80] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systenms*, 5:190–222, 1983.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[MC81] B. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. on Software Engineering*, 7:417–426, 1981.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, September 1971.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu85a] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Proc. Advanced School on Current Trends in Concurrency*, pages 510–584, Berlin, 1985. Volume 224, LNCS, Springer-Verlag.

[Pnu85b] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 123–144. Springer-Verlag, 1985.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[Saf89] S. Safra. *Complexity of automata on infinite objects*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1989.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.

[VS85] M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol89] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, volume 398, pages 75–123. Lecture Notes in Computer Science, Springer-Verlag, 1989.

Composition: A Way to Make Proofs Harder

Leslie Lamport

Systems Research Center, Digital Equipment Corporation

Abstract. Compositional reasoning about a system means writing its specification as the parallel composition of components and reasoning separately about each component. When distracting language issues are removed and the underlying mathematics is revealed, compositional reasoning is seen to be of little use.

1 Introduction

When an engineer designs a bridge, she makes a mathematical model of it and reasons mathematically about her model. She might talk about *calculating* rather than *reasoning*, but calculating $\sqrt{2}$ to three decimal places is just a way of proving $|\sqrt{2} - 1.414| < 10^{-3}$. The engineer reasons compositionally, using laws of mathematics to decompose her calculations into small steps. She would probably be mystified by the concept of compositional reasoning about bridges, finding it hard to imagine any form of reasoning that was not compositional.

Because computer systems can be built with software rather than girders and rivets, many computer scientists believe these systems should not be modeled with the ordinary mathematics used by engineers and scientists, but with something that looks vaguely like a programming language. We call such a language a *pseudo-programming languages* (PPL). Some PPLs, such as CSP, use constructs of ordinary programming languages. Others, like CCS, use more abstract notation. But, they have two defining properties: they are specially designed to model computer systems, and they are not meant to implement useful, real-world programs.

When using a pseudo-programming language, compositional reasoning means writing a model as the composition of smaller pseudo-programs, and reasoning separately about those smaller pseudo-programs. If one believes in using PPLs to model computer systems, then it is natural to believe that decomposition should be done in terms of the PPL, so compositionality must be a Good Thing.

We adopt the radical approach of modeling computer systems the way engineers model bridges—using mathematics. Compositionality is then a trivial consequence of the compositionality of ordinary mathematics. We will see that the compositional approaches based on pseudo-programming languages are analogous to performing calculations about a bridge design by decomposing it into smaller bridge designs. While this technique may occasionally be useful, it is hardly a good general approach to bridge design.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 402-423, 1998.
Springer-Verlag Berlin Heidelberg 1998

2 The Mathematical Laws of Composition

Mathematical reasoning is embodied in statements (also called theorems) and their proofs. The reasoning is hierarchical—the proof of a statement consists of a sequence of statements, each with its proof. The decomposition stops at a level at which the proof is sufficiently obvious that it can be written as a short, simple paragraph. How rigorous the proof is depends on what "obvious" means. In the most rigorous proofs, it means simple enough so that even a computer can verify it. Less rigorous proofs assume a reader of greater intelligence (or greater faith). We will use the notation introduced in [10] to write hierarchical proofs.

Two fundamental laws of mathematics are used to decompose proofs:

$$\text{$\wedge$-Composition}\quad \frac{A \Rightarrow B \qquad A \Rightarrow C}{A \Rightarrow B \wedge C} \qquad\qquad \text{$\vee$-Composition}\quad \frac{A \Rightarrow C \qquad B \Rightarrow C}{A \vee B \Rightarrow C}$$

Logicians have other names for these laws, but our subject is compositionality, so we adopt these names. A special case of $\vee$-composition is:

$$\text{Case-Analysis}\quad \frac{A \wedge B \Rightarrow C \qquad A \wedge \neg B \Rightarrow C}{A \Rightarrow C}$$

The propositional $\wedge$- and $\vee$-composition rules have the following predicate-logic generalizations:

$$\forall\text{-Composition}\quad \frac{(i \in S) \wedge P \Rightarrow Q(i)}{P \Rightarrow (\forall\, i \in S \,:\, Q(i))}$$

$$\exists\text{-Composition}\quad \frac{(i \in S) \wedge P(i) \Rightarrow Q}{(\exists\, i \in S \,:\, P(i)) \Rightarrow Q}$$

Another rule that is often used (under a very different name) is

$$\text{Act-Stupid}\quad \frac{A \Rightarrow C}{A \wedge B \Rightarrow C}$$

We call it the act-stupid rule because it proves that $A \wedge B$ implies C by ignoring the hypothesis B. This rule is useful when B can't help in the proof, so we need only the hypothesis A. Applying it in a general method, when we don't know what A and B are, is usually a bad idea.

3 Describing a System with Mathematics

We now explain how to use mathematics to describe systems. We take as our example a digital clock that displays the hour and minute. For simplicity, we ignore the fact that a clock is supposed to tell the real time, and we instead just specify the sequence of times that it displays. A more formal explanation of the approach can be found in [9].

3.1 Discrete Dynamic Systems

Our clock is a dynamic system, meaning that it evolves over time. The classic way to model a dynamic system is by describing its state as a continuous function of time. Such a function would describe the continuum of states the display passes through when changing from 12:49 to 12:50. However, we view the clock as a discrete system. Discrete systems are, by definition, ones we consider to exhibit discrete state changes. Viewing the clock as a discrete system means ignoring the continuum of real states and pretending that it changes from 12:49 to 12:50 without passing through any intermediate state. We model the execution of a discrete system as a sequence of states. We call such a sequence a *behavior*. To describe a system, we describe all the behaviors that it can exhibit.

3.2 An Hour Clock

A First Attempt To illustrate how systems are described mathematically, we start with an even simpler example than the hour-minute clock—namely, a clock that displays only the hour. We describe its state by the value of the variable hr. A typical behavior of this system is

$$[hr = 11] \;\rightarrow\; [hr = 12] \;\rightarrow\; [hr = 1] \;\rightarrow\; [hr = 2] \;\rightarrow\; \cdots$$

We describe all possible behaviors by an *initial predicate* that specifies the possible initial values of hr, and a *next-state relation* that specifies how the value of hr can change in any step (pair of successive states).

The initial predicate is just $hr \in \{1, \ldots, 12\}$. The next-state relation is the following formula, in which hr denotes the old value and hr' denotes the new value.

$$((hr = 12) \land (hr' = 1)) \;\lor\; ((hr \neq 12) \land (hr' = hr + 1))$$

This kind of formula is easier to read when written with lists of conjuncts or disjuncts, using indentation to eliminate parentheses:

$$\begin{array}{l} \lor \land\; hr = 12 \\ \quad \land\; hr' = 1 \\ \lor \land\; hr \neq 12 \\ \quad \land\; hr' = hr + 1 \end{array}$$

There are many ways to write the same formula. Borrowing some notation from programming languages, we can write this next-state relation as

$$hr' \;=\; \textbf{if } hr = 12 \;\textbf{ then } 1 \;\textbf{ else } hr + 1$$

This kind of formula, a Boolean-valued expression containing primed and unprimed variables, is called an *action*.

Our model is easier to manipulate mathematically if it is written as a single formula. We can write it as

$$\begin{array}{ll} \land\; hr \in \{1, \ldots, 12\} & (1) \\ \land\; \Box\,(hr' = \textbf{if } hr = 12 \;\textbf{ then } 1 \;\textbf{ else } hr + 1) & \end{array}$$

This is a temporal formula, meaning that it is true or false of a behavior. A state predicate like $hr \in \{1, \ldots, 12\}$ is true for a behavior iff it is true in the first state. A formula of the form $\Box N$ asserts that the action N holds on all steps of the behavior.

By introducing the operator $\Box$, we have left the realm of everyday mathematics and entered the world of temporal logic. Temporal logic is more complicated than ordinary mathematics. Having a single formula as our mathematical description is worth the extra complication. However, we should use temporal reasoning as little as possible. In any event, temporal logic formulas are still much easier to reason about than programs in a pseudo-programming language.

Stuttering Before adopting (1) as our mathematical description of the hour clock, we ask the question, what is a state? For a simple clock, the obvious answer is that a state is an assignment of values to the variable *hr*. What about a railroad station with a clock? To model a railroad station, we would use a number of additional variables, perhaps including a variable *sig* to record the state of a particular signal in the station. One possible behavior of the system might be

$$\begin{bmatrix} hr = 11 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} hr = 12 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} hr = 12 \\ sig = \text{“green”} \\ \vdots \end{bmatrix} \rightarrow$$
$$\begin{bmatrix} hr = 12 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} hr = 1 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \ldots$$

We would expect our description of a clock to describe the clock in the railroad station. However, formula (1) doesn't do this. It asserts that *hr* is incremented in every step, but the behavior of the railroad station with clock includes steps like the second and third, which change *sig* but leave *hr* unchanged.

To write a single description that applies to any clock, we let a state consist of an assignment of values to all possible variables. In mathematics, the equation $x + y = 1$, doesn't assert that there is no z. It simply says nothing about the value of z. In other words, the formula $x + y = 1$ is not an assertion about some universe containing only x and y. It is an assertion about a universe containing x, y, and all other variables; it constrains the values of only the variables x and y.

Similarly, a mathematical formula that describes a clock should be an assertion not about the variable *hr*, but about the entire universe of possible variables. It should constrain the value only of *hr* and should allow arbitrary changes to the other variables—including changes that occur while the value of *hr* stays the same. We obtain such a formula by modifying (1) to allow “stuttering” steps

that leave hr unchanged, obtaining:

$$\begin{array}{l} \land\ hr \in \{1, \ldots, 12\} \\ \land\ \Box \begin{pmatrix} \lor\ hr' = \textbf{if } hr = 12 \textbf{ then } 1 \textbf{ else } hr + 1 \\ \lor\ hr' = hr \end{pmatrix} \end{array} \tag{2}$$

Clearly, every next-state relation we write is going to have a disjunct that leaves variables unchanged. So, it's convenient to introduce the notation that $[A]_v$ equals $A \lor (v' = v)$, where v' is obtained from the expression v by priming all its free variables. We can then write (2) more compactly as

$$\begin{array}{l} \land\ hr \in \{1, \ldots, 12\} \\ \land\ \Box[hr' = \textbf{if } hr = 12 \textbf{ then } 1 \textbf{ else } hr + 1]_{hr} \end{array} \tag{3}$$

This formula allows behaviors that stutter forever, such as

$$[hr = 11] \rightarrow [hr = 12] \rightarrow [hr = 12] \rightarrow [hr = 12] \rightarrow \cdots$$

Such a behavior describes a stopped clock. It illustrates that we can assume all behaviors are infinite, because systems that halt are described by behaviors that end with infinite stuttering. But, we usually want our clocks not to stop.

Fairness To describe a clock that doesn't stop, we must add a conjunct to (3) to rule out infinite stuttering. Experience has shown that the best way to write this conjunct is with *fairness* formulas. There are two types of fairness, weak and strong, expressed with the WF and SF operators that are defined as follows.

$\mathrm{WF}_v(A)$ If $A \land (v' \neq v)$ is enabled forever, then infinitely many $A \land (v' \neq v)$ steps must occur.

$\mathrm{SF}_v(A)$ If $A \land (v' \neq v)$ is enabled infinitely often, then infinitely many $A \land (v' \neq v)$ steps must occur.

The $v' \neq v$ conjuncts make it impossible to use WF or SF to write a formula that rules out finite stuttering.

We can now write our description of the hour clock as the formula Π, defined by

$$\begin{array}{rcl} N & \triangleq & hr' = \textbf{if } hr = 12 \textbf{ then } 1 \textbf{ else } hr + 1 \\ \Pi & \triangleq & (hr \in \{1, \ldots, 12\}) \land \Box[N]_{hr} \land \mathrm{WF}_{hr}(N) \end{array}$$

The first two conjuncts of Π (which equal (3)), express a *safety* property. Intuitively, a safety property is characterized by any of the following equivalent conditions.

- It asserts that the system never does something bad.
- It asserts that the system starts in a good state and never takes a wrong step.

– It is finitely refutable—if it is violated, then it is violated at some particular point in the behavior.

The last conjunct of Π (the WF formula) is an example of a *liveness* property. Intuitively, a liveness property is characterized by any of the following equivalent conditions.

– It asserts that the system eventually does something good.

– It asserts that the system eventually takes a good step.

– It is not finitely refutable—it is possible to satisfy it after any finite portion of the behavior.

Formal definitions of safety and liveness are due to Alpern and Schneider [4].

Safety properties are proved using only ordinary mathematics (plus a couple of lines of temporal reasoning). Liveness properties are proved by combining temporal logic with ordinary mathematics. Here, we will mostly ignore liveness and concentrate on safety properties.

3.3 An Hour-Minute Clock

The Internal Specification It is now straightforward to describe a clock with an hour and minute display. The two displays are represented by the values of the variables *hr* and *min*. To make the specification more interesting, we describe a clock in which the two displays don't change simultaneously when the hour changes. When the display changes from 8:59 to 9:00, it transiently reads 8:00 or 9:59. Since we are ignoring the actual times at which state changes occur, these transient states are no different from the states when the clock displays the "correct" time.

Figure 1 defines a formula Φ that describes the hour-minute clock. It uses an additional variable *chg* that equals TRUE when the display is in a transient state. Action M_m describes the changing of *min*; action M_h describes the changing of *hr*. The testing and setting of *chg* by these actions is a bit tricky, but a little thought reveals what's going on. Action M_h introduces a gratuitous bit of cleverness to remove the **if/then** construct from the specification of the new value of *hr*. The next-state relation for the hour-minute clock is $M_m \vee M_h$, because a step of the clock increments either *min* or *hr*. Since $\langle hr, min\ chg\rangle'$ equals $\langle hr', min', chg'\rangle$, it equals $\langle hr, min, chg\rangle$ iff *hr*, *min*, and, *chg* are all unchanged.

Existential Quantification Formula Φ of Figure 1 contains the free variables *hr*, *min*, and *chg*. However, the description of a clock should mention only *hr* and *min*, not *chg*. We need to "hide" *chg*. In mathematics, hiding means existential quantification. The formula $\exists\, x : y = x^2$ asserts that there is some value of x that makes $y = x^2$ true; it says nothing about the actual value of x. The formula describing an hour-minute clock is $\exists\, chg : \Phi$. The quantifier $\exists$ is a temporal operator, asserting that there is a *sequence* of values of *chg* that makes Φ true. The precise definition of $\exists$ is a bit subtle and can be found in [9].

$$
\begin{array}{lcl}
\mathit{Init}_{\Phi} & \triangleq & \land\ hr \in \{1, \ldots, 12\} \\
& & \land\ min \in \{0, \ldots, 59\} \\
& & \land\ chg = \text{FALSE} \\
M_m & \triangleq & \land\ \neg((min = 0) \land chg) \\
& & \land\ min' = (min + 1) \bmod 60 \\
& & \land\ chg' = (min = 59) \land \neg chg \\
& & \land\ hr' = hr \\
M_h & \triangleq & \land\ \lor\ (min = 59) \land \neg chg \\
& & \quad \lor\ (min = 0) \land chg \\
& & \land\ hr' = (hr \bmod 12) + 1 \\
& & \land\ chg' = \neg chg \\
& & \land\ min' = min \\
\Phi & \triangleq & \land\ \mathit{Init}_{\Phi} \\
& & \land\ \Box[M_m \lor M_h]_{\langle hr, min, chg \rangle} \\
& & \land\ \mathrm{WF}_{\langle hr, min, chg \rangle}(M_m \lor M_h)
\end{array}
$$

Fig. 1. The internal specification of an hour-minute clock.

3.4 Implementation and Implication

An hour-minute clock implements an hour clock. (If we ask someone to build a device that displays the hour, we can't complain if the device also displays the minute.) Every behavior that satisfies the description of an hour-minute clock also satisfies the description of an hour clock. Formally, this means that the formula $(\exists\, chg : \Phi) \Rightarrow \Pi$ is true. In mathematics, if something is true, we should be able to prove it. The rules of mathematics allow us to decompose the proof hierarchically. Here is the statement of the theorem, and the first two levels of its proof. (See [10] for an explanation of the proof style.)

Theorem 1. $(\exists\, chg : \Phi) \Rightarrow \Pi$

$\langle 1 \rangle 1.$ $\Phi \Rightarrow \Pi$
  $\langle 2 \rangle 1.$ $\mathit{Init}_{\Phi} \Rightarrow hr \in \{1, \ldots, 12\}$
  $\langle 2 \rangle 2.$ $\Box[M_m \lor M_h]_{\langle hr, min, chg \rangle} \Rightarrow \Box[N]_{hr}$
  $\langle 2 \rangle 3.$ $\Phi \Rightarrow \mathrm{WF}_{hr}(N)$
  $\langle 2 \rangle 4.$ Q.E.D.
    PROOF: By $\langle 2 \rangle 1$–$\langle 2 \rangle 3$ and the $\land$-composition and act-stupid rules.
$\langle 1 \rangle 2.$ Q.E.D.
  PROOF: By $\langle 1 \rangle 1$, the definition of Φ, and predicate logic[1], since *chg* does not occur free in Π.

[1] We are actually reasoning about the temporal operator $\exists$ rather than ordinary existential quantification, but it obeys the usual rules of predicate logic.

Let's now go deeper into the hierarchical proof. The proof of $\langle 2\rangle 1$ is trivial, since $Init_\Phi$ contains the conjunct $hr \in \{1, \ldots, 12\}$. Proving liveness requires more temporal logic than we want to delve into here, so we will not show the proof of $\langle 2\rangle 3$ or of any other liveness properties. We expand the proof of $\langle 2\rangle 2$ two more levels as follows.

$\langle 2\rangle 2.\ \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box[N]_{hr}$
  $\langle 3\rangle 1.\ [M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow [N]_{hr}$
    $\langle 4\rangle 1.\ M_m \Rightarrow [N]_{hr}$
    $\langle 4\rangle 2.\ M_h \Rightarrow [N]_{hr}$
    $\langle 4\rangle 3.\ (\langle hr, min, chg\rangle' = \langle hr, min, chg\rangle) \Rightarrow [N]_{hr}$
    $\langle 4\rangle 4.$ Q.E.D.
      PROOF: By $\langle 4\rangle 1$–$\langle 4\rangle 3$ and the $\vee$-composition rule.
  $\langle 3\rangle 2.$ Q.E.D.
    PROOF: By $\langle 3\rangle 1$ and the rule $\dfrac{A \Rightarrow B}{\Box A \Rightarrow \Box B}$.

The proof of $\langle 4\rangle 1$ is easy, since M_m implies $hr' = hr$. The proof of $\langle 4\rangle 3$ is equally easy. The proof of $\langle 4\rangle 2$ looks easy enough.

$\langle 4\rangle 2.\ M_h \Rightarrow [N]_{hr}$
  PROOF: $M_h \Rightarrow hr' = (hr \bmod 12) + 1$
  $\Rightarrow hr' = \textbf{if } hr = 12 \textbf{ then } 1 \textbf{ else } hr + 1$
  $\triangleq N$

However, this proof is wrong! The second implication is not valid. For example, if hr equals 25, then the first equation asserts $hr' = 2$, while the second asserts $hr' = 26$. The implication is valid only under the additional assumption $hr \in \{1, \ldots, 12\}$.

Define Inv to equal the predicate $hr \in \{1, \ldots, 12\}$. We must show that Inv is true throughout the execution, and use that fact in the proof of step $\langle 4\rangle 2$. Here are the top levels of the corrected proof.

$\langle 1\rangle 1.\ \Phi \Rightarrow \Pi$
  $\langle 2\rangle 1.\ Init_\Phi \Rightarrow hr \in \{1, \ldots, 12\}$
  $\langle 2\rangle 2.\ Init_\Phi \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box Inv$
  $\langle 2\rangle 3.\ \Box Inv \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box[N]_{hr}$
  $\langle 2\rangle 4.\ \Box Inv \wedge \Phi \Rightarrow \mathrm{WF}_{hr}(N)$
  $\langle 2\rangle 5.$ Q.E.D.
    PROOF: By $\langle 2\rangle 1$–$\langle 2\rangle 4$, and the $\wedge$-composition and act-stupid rules.
$\langle 1\rangle 2.$ Q.E.D.
  PROOF: By $\langle 1\rangle 1$, the definition of Φ, and predicate logic, since chg does not occur free in Π.

The high-level proofs of $\langle 2\rangle 2$ and $\langle 2\rangle 3$ are

$\langle 2\rangle 2.\ Init_\Phi \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box Inv$
  $\langle 3\rangle 1.\ Init_\Phi \Rightarrow Inv$
  $\langle 3\rangle 2.\ Inv \wedge [M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow Inv'$
  $\langle 3\rangle 3.$ Q.E.D.
    PROOF: By $\langle 3\rangle 1$, $\langle 3\rangle 2$ and the rule $\dfrac{P \wedge [A]_v \Rightarrow P'}{P \wedge \Box[A]_v \Rightarrow \Box P}$.
$\langle 2\rangle 3.\ \Box Inv \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box[N]_{hr}$

⟨3⟩1. $Inv \wedge [M_m \vee M_h]_{\langle hr, min, chg \rangle} \Rightarrow [N]_{hr}$

⟨3⟩2. Q.E.D.

PROOF: By ⟨3⟩1 and the rules $\dfrac{A \Rightarrow B}{\Box A \Rightarrow \Box B}$ and $\Box(A \wedge B) \equiv \Box A \wedge \Box B$.

The further expansion of the proofs is straightforward and is left as an exercise for the diligent reader.

3.5 Invariance and Step Simulation

The part of the proof shown above is completely standard. It contains all the temporal-logic reasoning used in proving safety properties. The formula *Inv* satisfying ⟨2⟩2 is called an *invariant*. Substep ⟨3⟩2 of step ⟨2⟩3 is called proving *step simulation*. The invariant is crucial in this step and in step ⟨2⟩4 (the proof of liveness). In general, the hard parts of the proof are discovering the invariant, substep ⟨3⟩2 of step ⟨2⟩2 (the crucial step in the proof of invariance), step simulation, and liveness.

In our example, *Inv* asserts that the value of *hr* always lies in the correct set. Computer scientists call this assertion *type correctness*, and call the set of correct values the *type* of *hr*. Hence, *Inv* is called a type-correctness invariant. This is the simplest form of invariant. Computer scientists usually add a type system just to handle this particular kind of invariant, since they tend to prefer formalisms that are more complicated and less powerful than simple mathematics.

Most invariants express more interesting properties than just type correctness. The invariant captures the essence of what makes an implementation correct. Finding the right invariant, and proving its invariance, suffices to prove the desired safety properties of many concurrent algorithms. This is the basis of the first practical method for reasoning about concurrent algorithms, which is due to Ashcroft [5].

3.6 A Formula by any Other Name

We have been calling formulas like Φ and Π "descriptions" or "models" of a system. It is customary to call them *specifications*. This term is sometimes reserved for high-level description of systems, with low-level descriptions being called *implementations*. We make no distinction between specifications and implementations. They are all descriptions of a system at various levels of detail. We use the terms algorithm, description, model, and specification as different names for the same thing: a mathematical formula.

4 Invariance in a Pseudo-Programming Language

Invariance is a simple concept. We now show how a popular method for proving invariance in terms of a pseudo-programming language is a straightforward consequence of the rules of mathematics.

4.1 The Owicki-Gries Method

In the Owicki-Gries method [8, 11], the invariant is written as a program annotation. For simplicity, let's assume a multiprocess program in which each process i in a set $\mathbf{P}$ of processes repeatedly executes a sequence of atomic instructions $S_0^{(i)}$, ..., $S_{n-1}^{(i)}$. The invariant is written as an annotation, in which each statement $S_j^{(i)}$ is preceded by an assertion $A_j^{(i)}$, as shown in Figure 2.

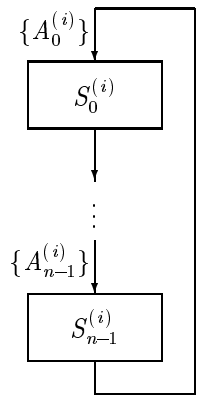


Fig. 2. An Owicki-Gries style annotation of a process.

To make sense of this picture, we must translate it into mathematics. We first rewrite each operation $S_j^{(i)}$ as an action, which we also call $S_j^{(i)}$. This rewriting is easy. For example, an assignment statement $x := x + 1$ is written as the action $(x' = x + 1) \wedge (\langle \ldots \rangle' = \langle \ldots \rangle)$, where "..." is the list of other variables. We represent the program's control state with a variable pc, where $pc[i] = j$ means that control in process i is immediately before statement $S_j^{(i)}$. The program and its invariant are then described by the formulas Π and Inv of Figure 3.

We can derive the Owicki-Gries rules for proving invariance by applying the proof rules we used before. The top-level proof is:

Theorem 2. (Owicki-Gries) $\Pi \Rightarrow \Box I$

$\langle 1 \rangle 1$. $Init \Rightarrow Inv$
$\langle 1 \rangle 2$. $Inv \wedge [N]_{\langle vbl,\, pc \rangle} \Rightarrow Inv'$
  $\langle 2 \rangle 1$. $Inv \wedge N \Rightarrow Inv'$
  $\langle 2 \rangle 2$. $Inv \wedge (\langle vbl,\, pc \rangle' = \langle vbl,\, pc \rangle) \Rightarrow Inv'$
  $\langle 2 \rangle 3$. Q.E.D.
    PROOF: By $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, and the $\vee$-composition rule.
$\langle 1 \rangle 3$. Q.E.D.
  PROOF: By $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and the rule $\dfrac{P \wedge [A]_v \Rightarrow P'}{P \wedge \Box[A]_v \Rightarrow \Box P}$.

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & \land\ \forall\, i \in \mathbf{P} \,:\, pc[i] = 0 \\
 & & \land\ \ldots \quad \text{[The initial conditions on program variables.]} \\
Go_j^{(i)} & \triangleq & \land\ pc[i] = j \\
 & & \land\ pc[i]' = (j+1) \bmod n \\
 & & \land\ \forall\, k \in \mathbf{P} \,:\, (k \neq i) \Rightarrow (pc[k]' = pc[k]) \\
N & \triangleq & \exists\, i \in \mathbf{P},\ j \in \{0, \ldots, n-1\} \,:\, Go_j^{(i)} \land S_j^{(i)} \\
\mathit{vbl} & \triangleq & \langle \ldots \rangle \quad \text{[The tuple of all program variables.]} \\
\Pi & \triangleq & \mathit{Init} \land \Box[N]_{\langle \mathit{vbl},\, pc \rangle} \\
\mathit{Inv} & \triangleq & \forall\, i \in \mathbf{P},\ j \in \{0, \ldots, n-1\} \,:\, (pc[i] = j) \Rightarrow A_j^{(i)}
\end{array}
$$

Fig. 3. The formulas describing the program and annotation of Figure 2.

The hard part is the proof of $\langle 2 \rangle 1$. We first decompose it using the $\forall$- and $\exists$-composition rules.

$\langle 2 \rangle 1.\ \mathit{Inv} \land N \Rightarrow \mathit{Inv}'$

$\langle 3 \rangle 1.\ \begin{pmatrix} \land\ i \in \mathbf{P} \\ \land\ j \in \{0, \ldots, n-1\} \\ \land\ \mathit{Inv} \land Go_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow \mathit{Inv}'$

$\langle 4 \rangle 1.\ \begin{pmatrix} \land\ i \in \mathbf{P} \\ \land\ j \in \{0, \ldots, n-1\} \\ \land\ k \in \mathbf{P} \\ \land\ l \in \{0, \ldots, n-1\} \\ \land\ \mathit{Inv} \land Go_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow ((pc[k]' = l) \Rightarrow (A_l^{(k)})')$

$\langle 4 \rangle 2.$ Q.E.D.

PROOF: By $\langle 4 \rangle 1$, the definition of *Inv*, and the $\forall$-composition rule.

$\langle 3 \rangle 2.$ Q.E.D.

PROOF: By $\langle 3 \rangle 1$, the definition of N, and the $\exists$-composition rule.

We prove $\langle 4 \rangle 1$ by cases, after first using propositional logic to simplify its statement. We let $j \oplus 1$ equal $(j+1)$ mod n.

$\langle 4 \rangle 1.\ \begin{pmatrix} \land\ i, k \in \mathbf{P} \\ \land\ j, l \in \{0, \ldots, n-1\} \\ \land\ pc[k]' = l \\ \land\ \mathit{Inv} \land Go_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow (A_l^{(k)})'$

$\langle 5 \rangle 1.$ CASE: $i = k$

$\langle 6 \rangle 1.\ \begin{pmatrix} \land\ i \in \mathbf{P} \\ \land\ j \in \{0, \ldots, n-1\} \\ \land\ A_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow (A_{j \oplus 1}^{(i)})'$

$\langle 6 \rangle 2.$ Q.E.D.

PROOF: By $\langle 6 \rangle 1$, the level-$\langle 5 \rangle$ assumption, the definition of *Inv*, and

the act-stupid rule, since $(pc[i]' = l) \wedge Go_j^{(i)}$ implies $(l = j \oplus 1)$.

$\langle 5 \rangle 2$. CASE: $i \neq k$

$\langle 6 \rangle 1$. $\begin{pmatrix} \wedge\ i, k \in \mathbf{P} \\ \wedge\ j, l \in \{0, \ldots, n-1\} \\ \wedge\ A_j^{(i)} \wedge A_l^{(k)} \wedge S_j^{(i)} \end{pmatrix} \Rightarrow (A_l^{(k)})'$

$\langle 6 \rangle 2$. Q.E.D.

PROOF: By $\langle 6 \rangle 1$, the level-$\langle 5 \rangle$ assumption, the definition of *Inv*, and the act-stupid rule, since $(pc[k]' = l) \wedge Go_j^{(i)}$ implies $(pc[k] = l)$, for $k \neq i$, and $(pc[k] = l) \wedge Inv$ implies $A_l^{(k)}$.

We are finally left with the two subgoals numbered $\langle 6 \rangle 1$. Summarizing, we see that to prove $Init \Rightarrow \Box Inv$, it suffices to prove the two conditions

$$A_j^{(i)} \wedge S_j^{(i)} \Rightarrow (A_{j \oplus 1}^{(i)})'$$
$$A_j^{(i)} \wedge A_l^{(k)} \wedge S_j^{(i)} \Rightarrow (A_l^{(k)})'$$

for all i, k in $\mathbf{P}$ with $i \neq k$, and all j, l in $\{0, \ldots, n-1\}$. These conditions are called *Sequential Correctness* and *Interference Freedom*, respectively.

4.2 Why Bother?

We now consider just what have has been accomplished by describing by proving invariance in terms of a pseudo-programming language instead of directly in mathematics.

Computer scientists are quick to point out that using "$:=$" instead of "$=$" avoids the need to state explicitly what variables are left unchanged. In practice, this reduces the length of a specification by anywhere from about 10% (for a very simple algorithm) to 4% (for a more complicated system). For this minor gain, it introduces the vexing problem of figuring out exactly what variables can and cannot be changed by executing $x := x + 1$. The obvious requirement that no other variable is changed would not allow us to implement x as the sum $lh * 2^{32} + rh$ of two 32-bit values, since it forbids lh and rh to change when x is incremented. The difficulty of deciding what can and cannot be changed by an assignment statement is one of the things that makes the semantics of programming languages (both real and pseudo) complicated. By using mathematics, we avoid this problem completely.

A major achievement of the Owicki-Gries method is eliminating the explicit mention of the variable *pc*. By writing the invariant as an annotation, one can write $A_j^{(i)}$ instead of $(pc[i] = j) \Rightarrow A_j^{(i)}$. At the time, computer scientists seemed to think that mentioning *pc* was a sin. However, when reasoning about a concurrent algorithm, we must refer to the control state in the invariant. Owicki and Gries therefore had to introduce dummy variables to serve as euphemisms for *pc*. When using mathematics, any valid formula of the form $Init \wedge \Box[N]_v \Rightarrow \Box P$, for a state predicate P, can be proved without adding dummy variables.

One major drawback of the Owicki-Gries method arises from the use of the act-stupid rule in the proofs of the two steps numbered $\langle 6 \rangle 2$. The rule was applied

without regard for whether the hypotheses being ignored are useful. This means that there are annotations for which step $\langle 2\rangle 1$ (which asserts $N \wedge Inv \Rightarrow Inv'$) is valid but cannot be proved with the Owicki-Gries method. Such invariants must be rewritten as different, more complicated annotations.

Perhaps the thing about the Owicki-Gries method is that it obscures the underlying concept of invariance. We refer the reader to [6] for an example of how complicated this simple concept becomes when expressed in terms of a pseudo-programming language. In 1976, the Owicki-Gries method seemed like a major advance over Ashcroft's simple notion of invariance. We have since learned better.

5 Refinement

5.1 Refinement in General

We showed above that an hour-minute clock implements an hour clock by proving $(\exists\, chg : \Phi) \Rightarrow \Pi$. That proof does not illustrate the general case of proving that one specification implements another because the higher-level specification Π has no internal (bound) variable. The general case is covered by the following proof outline, where x, y, and z denote arbitrary tuples of variables, and the internal variables y and z of the two specifications are distinct from the free variables x. The proof involves finding a function f, which is called a *refinement mapping* [1].

Theorem 3. (Refinement) $(\exists\, y : \Phi(x, y)) \Rightarrow (\exists\, z : \Pi(x, z))$

LET: $\overline{z} \triangleq f(x, y)$
$\langle 1\rangle 1.$ $\Phi(x, y) \Rightarrow \Pi(x, \overline{z})$
$\langle 1\rangle 2.$ $\Phi(x, y) \Rightarrow (\exists\, z : \Pi(x, z))$
PROOF: By $\langle 1\rangle 1$ and predicate logic, since the variables of z are distinct from those of x.

The proof of step $\langle 1\rangle 1$ has the same structure as in our clock example.

5.2 Hierarchical Refinement

In mathematics, it is common to prove a theorem of the form $P \Rightarrow Q$ by introducing a new formula R and proving $P \Rightarrow R$ and $R \Rightarrow Q$. We can prove that a lower-level specification $\exists\, y : \Phi(x, y)$ implies a higher-level specification $\exists\, z : \Pi(x, z)$ by introducing an intermediate-level specification $\exists\, w : \Psi(x, w)$ and using the following proof outline.

LET: $\Psi(x, w) \triangleq \ldots$
$\langle 1\rangle 1.$ $(\exists\, y : \Phi(x, y)) \Rightarrow (\exists\, w : \Psi(x, w))$
LET: $\overline{w} \triangleq g(x, y)$
$\ldots$
$\langle 1\rangle 2.$ $(\exists\, w : \Psi(x, w)) \Rightarrow (\exists\, z : \Pi(x, z))$

LET: $\overline{z} \triangleq h(x, w)$
...

$\langle 1 \rangle 3$. Q.E.D.
PROOF: By $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$.

This proof method is called *hierarchical decomposition*. It's a good way to explain a proof. By using a sequence multiple intermediate specifications, each differing from the next in only one aspect, we can decompose the proof into conceptually simple steps.

Although it is a useful pedagogical tool, hierarchical decomposition does not simplify the total proof. In fact, it usually adds extra work. Hierarchical decomposition adds the task of writing the extra intermediate-level specification. It also restricts how the proof is decomposed. The single refinement mapping f in the outline of the direct proof can be defined in terms of the two mappings g and h of the hierarchical proof by $f(x, y) \triangleq h(x, g(x, y))$. The steps of a hierarchical proof can then be reshuffled to form a particular way of decomposing the lower levels of the direct proof. However, there could be better ways to decompose those levels.

5.3 Interface Refinement

We have said that implementation is implication. For this to be true, the two specifications must have the same free variables. If the high-level specification describes the sending of messages on a network whose state is represented by the variable *net*, then the low-level specification must also describe the sending of messages on *net*.

We often implement a specification by refining the interface. For example, we might implement a specification $\Sigma(net)$ of sending messages on *net* by a specification $\Lambda(tran)$ of sending packets on a "transport layer" whose state is represented by a variable *tran*. A single message could be broken into multiple packets. Correctness of the implementation cannot mean validity of $\Lambda(tran) \Rightarrow \Sigma(net)$, since $\Lambda(tran)$ and $\Sigma(net)$ have different free variables.

To define what it means for $\Lambda(tran)$ to implement $\Sigma(net)$, we must first define what it means for sending a set of packets to represent the sending of a message. This definition is written as a temporal formula $R(net, trans)$, which is true of a behavior iff the sequence of values of *trans* represents the sending of packets that correspond to the sending of messages represented by the sequence of values of *net*. We call R an *interface refinement*. For R to be a sensible interface refinement, the formula $\Lambda(trans) \Rightarrow \exists\, net : R(net, trans)$ must be valid, meaning that every set of packet transmissions allowed by $\Lambda(trans)$ represents some set of message transmissions. We say that $\Lambda(tran)$ implements $\Sigma(net)$ under the interface refinement $R(net, trans)$ iff $\Lambda(tran) \wedge R(net, trans)$ implies $\Sigma(net)$.

6 Decomposing Specifications

Pseudo-programming languages usually have some parallel composition operator $\|$, where $S_1 \| S_2$ is the parallel composition of specifications S_1 and S_2. We

observed in our hour-clock example that a mathematical specification S_1 does not describe only a particular system; rather, it describes a universe containing (the variables that represent) the system. Composing two systems means ensuring that the universe satisfies both of their specifications. Hence, when the specifications S_1 and S_2 are mathematical formulas, their composition is just $S_1 \wedge S_2$.

6.1 Decomposing a Clock into its Hour and Minute Displays

We illustrate how composition becomes conjunction by specifying the hour-minute clock as the conjunction of the specifications of an hour process and a minute process. It is simpler to do this if each variable is modified by only one process. So, we rewrite the specification of the hour-minute clock by replacing the variable chg with the expression $chg_h \neq chg_m$, where chg_h and chg_m are two new variables, chg_h being modified by the hour process and chg_m by the minute process. The new specification is $\exists\, chg_h, chg_m : \Psi$, where Ψ is defined in Figure 4. Proving that this specification is equivalent to $\exists\, chg : \Phi$,

$$
\begin{array}{lcl}
Init_\Psi & \triangleq & \wedge\ hr \in \{1, \ldots, 12\} \\
 & & \wedge\ min \in \{0, \ldots, 59\} \\
 & & \wedge\ chg_m = chg_h = \text{TRUE} \\
N_m & \triangleq & \wedge\ \neg((min = 0) \wedge (chg_m \neq chg_h)) \\
 & & \wedge\ min' = (min + 1) \bmod 60 \\
 & & \wedge\ chg'_m = \textbf{if } min = 59 \textbf{ then } \neg chg_m \textbf{ else } chg_h \\
 & & \wedge\ \langle hr, chg_h \rangle' = \langle hr, chg_h \rangle \\
N_h & \triangleq & \wedge\ \vee\ (min = 59) \wedge (chg_m = chg_h) \\
 & & \quad\ \vee\ (min = 0) \wedge (chg_m \neq chg_h) \\
 & & \wedge\ hr' = (hr \bmod 12) + 1 \\
 & & \wedge\ chg'_h = \neg chg_h \\
 & & \wedge\ \langle min, chg_m \rangle' = \langle min, chg_m \rangle \\
\Psi & \triangleq & \wedge\ Init_\Psi \\
 & & \wedge\ \Box[N_m \vee N_h]_{\langle hr,\, min,\, chg_m,\, chg_h \rangle} \\
 & & \wedge\ \mathrm{WF}_{\langle hr,\, min,\, chg_m,\, chg_h \rangle}(N_m \vee N_h)
\end{array}
$$

Fig. 4. Another internal specification of the hour-minute clock.

where Φ is defined in Figure 1, is left as a nice exercise for the reader. The proof that $\exists\, chg_h, chg_m : \Psi$ implies $\exists\, chg : \Phi$ uses the refinement mapping $\overline{chg} \triangleq (chg_h \neq chg_m)$. The proof of the converse implication uses the refinement mapping

$$\overline{chg_h} \triangleq chg \wedge (min = 59) \qquad \overline{chg_m} \triangleq chg \wedge (min = 0)$$

The specifications Ψ_h and Ψ_m of the hour and minute processes appear in Figure 5. We now sketch the proof that Ψ is the composition of those two specifi-

$$
\begin{array}{lcl}
Init_m & \triangleq & \wedge\ min \in \{0, \ldots, 59\} \\
 & & \wedge\ chg_m = \text{TRUE} \\
Init_h & \triangleq & \wedge\ hr \in \{1, \ldots, 12\} \\
 & & \wedge\ chg_h = \text{TRUE} \\
\Psi_h & \triangleq & Init_h \wedge \Box[N_h]_{\langle hr,\, chg_h \rangle} \quad \wedge \mathrm{WF}_{\langle hr,\, chg_h \rangle}(N_h) \\
\Psi_m & \triangleq & Init_m \wedge \Box[N_m]_{\langle min,\, chg_m \rangle} \wedge \mathrm{WF}_{\langle min,\, chg_m \rangle}(N_m)
\end{array}
$$

Fig. 5. Definition of the specifications Ψ_h and Ψ_m.

cations.

Theorem 4. $\Psi \equiv \Psi_m \wedge \Psi_h$

$\langle 1 \rangle 1.$ $Init_\Psi \equiv Init_m \wedge Init_h$
$\langle 1 \rangle 2.$ $\Box[N_m \vee N_h]_{\langle hr,\, min,\, chg_m,\, chg_h \rangle} \equiv \Box[N_m]_{\langle min,\, chg_m \rangle} \wedge \Box[N_h]_{\langle hr,\, chg_h \rangle}$
  $\langle 2 \rangle 1.$ $[N_m \vee N_h]_{\langle hr,\, min,\, chg_m,\, chg_h \rangle} \equiv [N_m]_{\langle min,\, chg_m \rangle} \wedge [N_h]_{\langle hr,\, chg_h \rangle}$
  $\langle 2 \rangle 2.$ Q.E.D.
    PROOF: By $\langle 2 \rangle 1$ and the rules $\dfrac{A \Rightarrow B}{\Box A \Rightarrow \Box B}$ and $\Box(A \wedge B) \equiv \Box A \wedge \Box B$.
$\langle 1 \rangle 3.$ $\wedge\ \Psi \Rightarrow \mathrm{WF}_{\langle min,\, chg_m \rangle}(N_m) \wedge \mathrm{WF}_{\langle hr,\, chg_h \rangle}(N_h)$
    $\wedge\ \Psi_m \wedge \Psi_h \Rightarrow \mathrm{WF}_{\langle hr,\, min,\, chg_m,\, chg_h \rangle}(N_m \vee N_h)$
$\langle 1 \rangle 4.$ Q.E.D.
  PROOF: By $\langle 1 \rangle 1$–$\langle 1 \rangle 3$.

Ignoring liveness (step $\langle 1 \rangle 3$), the hard part is proving $\langle 2 \rangle 1$. This step is an immediate consequence of the following propositional logic tautology, which we call the $\vee \leftrightarrow \wedge$ rule.

$$\frac{N_i \wedge (j \neq i) \Rightarrow (v'_j = v_j) \ \text{ for } 1 \leq i, j \leq n}{[N_1 \vee \ldots \vee N_n]_{\langle v_1, \ldots, v_n \rangle} = [N_1]_{v_1} \wedge \ldots \wedge [N_n]_{v_n}}$$

Its proof is left as an exercise for the reader.

6.2 Decomposing Proofs

In pseudo-programming language terminology, a compositional proof of refinement (implementation) is one performed by breaking a specification into the parallel composition of processes and separately proving the refinement of each process.

The most naive translation of this into mathematics is that we want to prove $\Lambda \Rightarrow \Sigma$ by writing Σ as $\Sigma_1 \wedge \Sigma_2$ and proving $\Lambda \Rightarrow \Sigma_1$ and $\Lambda \Rightarrow \Sigma_2$ separately. Such a decomposition accomplishes little. The lower-level specification

Λ is usually much more complicated than the higher-level specification Σ, so decomposing Σ is of no interest.

A slightly less naive translation of compositional reasoning into mathematics involves writing both Λ and Σ as compositions. This leads to the following proof of $\Lambda \Rightarrow \Sigma$.

$\langle 1 \rangle 1.\ \wedge \Lambda \equiv \Lambda_1 \wedge \Lambda_2$
$\qquad \wedge \Sigma \equiv \Sigma_1 \wedge \Sigma_2$
PROOF: Use the $\vee \leftrightarrow \wedge$ rule.
$\langle 1 \rangle 2.\ \Lambda_1 \Rightarrow \Sigma_1$
$\langle 1 \rangle 3.\ \Lambda_2 \Rightarrow \Sigma_2$
$\langle 1 \rangle 4.$ Q.E.D.
PROOF: By $\langle 1 \rangle 1$–$\langle 1 \rangle 3$ and the $\wedge$-composition and act-stupid rules.

The use of the act-stupid rule in the final step tells us that we have a problem. Indeed, this method works only in the most trivial case. Proving each of the implications $\Lambda_i \Rightarrow \Sigma_i$ requires proving $\Lambda_i \Rightarrow Inv_i$ for some invariant Inv_i. Except when each process accesses only its own variables, so there is no communication between the two processes, Inv_i will have to mention the variables of both processes. As our clock example illustrates, the next-state relation of each process's specification allows arbitrary changes to the other process's variables. Hence, Λ_i can't imply any nontrivial invariant that mentions the other process's variables. So, this proof method doesn't work.

Think of each process Λ_i as the other process's *environment*. We can't prove $\Lambda_i \Rightarrow \Sigma_i$ because it asserts that Λ_i implements Σ_i in the presence of arbitrary behavior by its environment—that is, arbitrary changes to the environment variables. No real process works in the face of completely arbitrary environment behavior.

Our next attempt at compositional reasoning is to write a specification E_i of the assumptions that process i requires of its environment and prove $\Lambda_i \wedge E_i \Rightarrow \Sigma_i$. We hope that one process doesn't depend on all the details of the other process's specification, so E_i will be much simpler than the other process's specification Λ_{2-i}. We can then prove $\Lambda \Rightarrow \Sigma$ using the following propositional logic tautology.

$$\frac{\begin{array}{ll} \Lambda_1 \wedge \Lambda_2 \Rightarrow E_1 & \Lambda_1 \wedge \Lambda_2 \Rightarrow E_2 \\ \Lambda_1 \wedge E_1 \Rightarrow \Sigma_1 & \Lambda_2 \wedge E_2 \Rightarrow \Sigma_2 \end{array}}{\Lambda_1 \wedge \Lambda_2 \Rightarrow \Sigma_1 \wedge \Sigma_2}$$

However, this requires proving $\Lambda \Rightarrow E_i$, so we still have to reason about the complete lower-level specification Λ. What we need is a proof rule of the following form

$$\frac{\begin{array}{ll} \Sigma_1 \wedge \Sigma_2 \Rightarrow E_1 & \Sigma_1 \wedge \Sigma_2 \Rightarrow E_2 \\ \Lambda_1 \wedge E_1 \Rightarrow \Sigma_1 & \Lambda_2 \wedge E_2 \Rightarrow \Sigma_2 \end{array}}{\Lambda_1 \wedge \Lambda_2 \Rightarrow \Sigma_1 \wedge \Sigma_2} \tag{4}$$

In this rule, the hypotheses $\Lambda \Rightarrow E_i$ of the previous rule are replaced by $\Sigma \Rightarrow E_i$. This is a great improvement because Σ is usually much simpler than Λ. A rule like (4) is called a *decomposition theorem*.

Unfortunately, (4) is not valid for arbitrary formulas. (For example, let the Λ_i equal TRUE and all the other formulas equal FALSE.) Roughly speaking, (4) is valid if all the properties are safety properties, and if Σ_i and E_i modify disjoint sets of variables, for each i. A more complicated version of the rule allows the Λ_i and Σ_i to include liveness properties; and the condition that Σ_i and E_i modify disjoint sets of variables can be replaced by a weaker, more complicated requirement. Moreover, everything generalizes from two conjuncts to n in a straightforward way. All the details can be found in [2].

6.3 Why Bother?

What have we accomplished by using a decomposition theorem of the form (4)? As our clock example shows, writing a specification as the conjunction of n processes rests on an equivalence of the form

$$\Box[N_1 \vee \ldots \vee N_n]_{\langle v_1, \ldots v_n \rangle} \equiv \Box[N_1]_{v_1} \wedge \ldots \wedge \Box[N_n]_{v_n}$$

Replacing the left-hand side by the right-hand side essentially means changing from disjunctive normal form to conjunctive normal form. In a proof, this replaces $\vee$-composition with $\wedge$-composition. Such a trivial transformation is not going to simplify a proof. It just changes the high-level structure of the proof and rearranges the lower-level steps.

Not only does this transformation not simplify the final proof, it may add extra work. We have to invent the environment specifications E_i, and we have to check the hypotheses of the decomposition theorem. Moreover, handling liveness can be problematic. In the best of all possible cases, the specifications E_i will provide useful abstractions, the extra hypotheses will follow directly from existing theorems, and the decomposition theorem will handle the liveness properties. In this best of all possible scenarios, we still wind up only doing exactly the same proof steps as we would in proving the implementation directly without decomposing it.

This form of decomposition is popular among computer scientists because it can be done in a pseudo-programming language. A conjunction of complete specifications like $\Lambda_1 \wedge \Lambda_2$ corresponds to parallel composition, which can be written in a PPL as $\Lambda_1 \| \Lambda_2$. The PPL is often sufficiently inexpressive that all the specifications one can write trivially satisfy the hypotheses of the decomposition theorem. For example, the complications introduced by liveness are avoided if the PPL provides no way to express liveness.

Many computer scientists prefer to do as much of a proof as possible in the pseudo-programming language, using its special-purpose rules, before being forced to enter the realm of mathematics with its simple, powerful laws. They denigrate the use of ordinary mathematics as mere "semantic reasoning". Because mathematics can so easily express the underlying semantics of a pseudo-programming language, any proof in the PPL can be translated to a semantic proof. Any law for manipulating language constructs will have a counterpart that is a theorem of ordinary mathematics for manipulating a particular class of

formulas. Mathematics can also provide methods of reasoning that have no counterpart in the PPL because of the PPL's limited expressiveness. For example, because it can directly mention the control state, an invariance proof based on ordinary mathematics is often simpler than one using the Owicki-Gries method.

Many computer scientists believe that their favorite pseudo-programming language is better than mathematics because it provides wonderful abstractions such as message passing, or synchronous communication, or objects, or some other popular fad. For centuries, bridge builders, rocket scientists, nuclear physicists, and number theorists have used their own abstractions. They have all expressed those abstractions directly in mathematics, and have reasoned "at the semantic level". Only computer scientists have felt the need to invent new languages for reasoning about the objects they study.

Two empirical laws seem to govern the difficulty of proving the correctness of an implementation, and no pseudo-programming language is likely to circumvent them: (1) the length of a proof is proportional to the product of the length of the low-level specification and the length of the invariant, and (2) the length of the invariant is proportional to the length of the low-level specification. Thus, the length of the proof is quadratic in the length of the low-level specification. To appreciate what this means, consider two examples. The specification of the lazy caching algorithm of Afek, Brown, Merritt [3], a typical high-level algorithm, is 50 lines long. The specification of the cache coherence protocol for a new computer that we worked on is 1900 lines long. We expect the lengths of the two corresponding correctness proofs to differ by a factor of 1500.

The most effective way to reduce the length of an implementation proof is to reduce the length of the low-level specification. A specification is a mathematical abstraction of a real system. When writing the specification, we must choose the level of abstraction. A higher-level abstraction yields a shorter specification. But a higher-level abstraction leaves out details of the real system, and a proof cannot detect errors in omitted details. Verifying a real system involves a tradeoff between the level of detail and the size (and hence difficulty) of the proof.

A quadratic relation between one length and another implies the existence of a constant factor. Reducing this constant factor will shorten the proof. There are several ways to do this. One is to use better abstractions. The right abstraction can make a big difference in the difficulty of a proof. However, unless one has been really stupid, inventing a clever new abstraction is unlikely to help by more than a factor of five. Another way to shorten a proof is to be less rigorous, which means stopping a hierarchical proof one or more levels sooner. (For real systems, proofs reach a depth of perhaps 12 to 20 levels.) Choosing the depth of a proof provides a tradeoff between its length and its reliability. There are also silly ways to reduce the size of a proof, such as using small print or writing unstructured, hand-waving proofs (which are known to be completely unreliable).

Reducing the constant factor still does not alter the essential quadratic nature of the problem. With systems getting ever more complicated, people who try to verify them must run very hard to stay in the same place. Philosophically motivated theories of compositionality will not help.

6.4 When a Decomposition Theorem is Worth the Bother

As we have observed, using a decomposition theorem can only increase the total amount of work involved in proving that one specification implements another. There is one case in which it's worth doing the extra work: when the computer does a lot of it for you. If we decompose the specifications Λ and Σ into n conjuncts Λ_i and Σ_i, the hypotheses of the decomposition theorem become $\Sigma \Rightarrow E_i$ and $\Lambda_i \wedge E_i \Rightarrow \Sigma_i$, for $i = 1, \ldots, n$. The specification Λ is broken into the smaller components Λ_i. Sometimes, these components will be small enough that the proof of $\Lambda_i \wedge E_i \Rightarrow \Sigma_i$ can be done by model checking—using a computer to examine all possible equivalence classes of behaviors. In that case, the extra work introduced by decomposition will be more than offset by the enormous benefit of using model checking instead of human reasoning. An example of such a decomposition is described in [7].

7 Composing Specifications

There is one situation in which compositional reasoning cannot be avoided: when one wants to reason about a component that may be used in several different systems.

The specifications we have described thus far have been *complete-system* specifications. Such specifications describe all behaviors in which both the system and its environment behave correctly. They can be written in the form $S \wedge E$, where S describes the system and E the environment. For example, if we take the component to be our clock example's hour process, then S is the formula Ψ_h and E is Ψ_m. (The hour process's environment consists of the minute process.)

If a component may be used in multiple systems, we need to write an *open-system* specification—one that specifies the component itself, not the complete system containing it. Intuitively, the component's specification asserts that it satisfies S if the environment satisfies E. This suggests that the component's open-system specification should be the formula $E \Rightarrow S$. This specification allows behaviors in which the system misbehaves, if the environment also misbehaves. It turns out to be convenient to rule out behaviors in which the system misbehaves first. (Such behaviors could never be allowed by a real implementation, which cannot know in advance that the environment will misbehave.) We therefore take as the specification the formula $E \stackrel{+}{\triangleright} S$, which is satisfied by a behavior in which S holds as long as E does. The precise definition of $\stackrel{+}{\triangleright}$ and the precise statement of the results about open-system specifications can be found in [2].

The basic problem of compositional reasoning is showing that the composition of component specifications satisfies a higher-level specification. This means proving that the conjunction of specifications of the form $E \stackrel{+}{\triangleright} S$ implies another specification of that form. For two components, the proof rule we want is:

$$\frac{E \wedge S_1 \wedge S_2 \;\Rightarrow\; E_1 \wedge E_2 \wedge S}{(E_1 \stackrel{+}{\triangleright} S_1) \;\wedge\; (E_2 \stackrel{+}{\triangleright} S_2) \;\Rightarrow\; (E \stackrel{+}{\triangleright} S)}$$

Such a rule is called a *composition theorem*. As with the decomposition theorem (4), it is valid only for safety properties under certain disjointness assumptions; a more complicated version is required if S and the S_i include liveness properties.

Composition of open-system specifications is an attractive problem, having obvious application to reusable software and other trendy concerns. But in 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It is naive to expect them to go to the extra effort of proving properties of open-system component specifications because they might re-use those components in other systems. It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years. Formal specifications of systems, with no accompanying verification, may become common sooner. However, the difference between the open-system specification $E \stackrel{+}{\triangleright} M$ and the complete-system specification $E \wedge M$ is one symbol—hardly a major concern in a specification that may be 50 or 200 pages long.

8 Conclusion

What should we do if faced with the problem of finding errors in the design of a real system? The complete design will almost always be too complicated to handle by formal methods. We must reason about an abstraction that represents as much of the design as possible, given the limited time and manpower available.

The ideal approach is to let a computer do the verification, which means model checking. Model checkers can handle only a limited class of specifications. These specifications are generally small and simple enough that it makes little difference in what language they are written—conventional mathematics or pseudo-programming languages should work fine. For many systems, abstractions that are amenable to model checking omit too many important aspects of the design. Human reasoning—that is, mathematical proof—is then needed. Occasionally, this reasoning can be restricted to rewriting the specification as the composition of multiple processes, decomposing the problem into subproblems suitable for model checking. In many cases, such a decomposition is not feasible, and mathematical reasoning is the only option.

Any proof in mathematics is compositional—a hierarchical decomposition of the desired result into simpler subgoals. A sensible method of writing proofs will make that hierarchical decomposition explicit, permitting a tradeoff between the length of the proof and its rigor. Mathematics provides more general and more powerful ways of decomposing a proof than just writing a specification as the parallel composition of separate components. That particular form of decomposition is popular only because it can be expressed in terms of the pseudo-programming languages favored by computer scientists.

Mathematics has been developed over two millennia as the best approach to rigorous human reasoning. A couple of decades of pseudo-programming language design poses no threat to its pre-eminence. The best way to reason mathematically is to use mathematics, not a pseudo-programming language.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
3. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
4. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
5. E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
6. Edsger W. Dijkstra. A personal summary of the Gries-Owicki theory. In Edsger W. Dijkstra, editor, *Selected Writings on Computing: A Personal Perspective*, chapter EWD554, pages 188–199. Springer-Verlag, New York, Heidelberg, Berlin, 1982.
7. R. P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179, Berlin, June 1993. Springer-Verlag. Proceedings of the Fifth International Conference, CAV'93.
8. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
9. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
10. Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1995.
11. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

Compositionality Criteria for Defining Mixed-Styles Synchronous Languages*

Florence Maraninchi and Yann Rémond

VERIMAG** – Centre Equation, 2 Av. de Vignate – F38610 GIERES
http://www.imag.fr/VERIMAG/PEOPLE/Florence.Maraninchi

Abstract. This is not a paper about compositionality in itself, nor a general paper about mixing synchronous languages. We first recall that *compositionality* appears in three places in the definition of synchronous languages : 1) the synchrony hypothesis guarantees that the formal semantics of the language is compositional (in the sense that there exists an appropriate congruence) ; 2) programming environments offer separate compilation, at various levels ; 3) the idea of using *synchronous observers* for describing the properties of a program provides a kind of assume/guarantee scheme, thus enabling compositional proofs. Then we take an example in order to illustrate these good properties of synchronous languages : the idea is to extend a dataflow language like Lustre with a construct that supports the description of *running modes*. We show how to use compositionality arguments when choosing the semantics of a such a mixed-style language. The technical part is taken from [MR98].

1 Introduction

1.1 About compositionality

The call for participation contains a definition of compositionality. It says: "*Any method by which the properties of a system can be inferred from the properties of its constituents, without additional information about the internal structure of the constituents*".

This implies the existence of two domains: the *constituents* of systems, and their *properties*. The relationship between these two domains is a function that associates properties to constituents. Constituents may be *composed* in order to form bigger systems; properties may be combined so that another property is *inferred* from a set of already given ones.

Moreover, "*without additional information about the internal structure of the constituents*" clearly states that the function that associates the properties to a constituent is non injective. Hence there exist two distinct systems having the

* This work has been partially supported by Esprit Long Term Research Project SYRF 22703

** Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 424-438, 1998.
Springer-Verlag Berlin Heidelberg 1998

same "properties". Having the same properties defines an *equivalence* of systems. And in any context where objects may be composed and declared equivalent, it is of a particular interest that the equivalence be a congruence for the available compositions. This scheme may be instanciated in a number of ways.

1.2 Compositionality and logical properties

One instance is given by associating *logical properties* to *systems*. In this case, two problems are of interest:

- Given a program P, which is the composition of two "constituents" P_1 and P_2 — i.e. $P = op(P_1, P_2)$ —, and a property Φ we want to prove for P, how to find a property Φ_1 holding for P_1, and a property Φ_2 holding for P_2, such that we can logically "infer" Φ from Φ_1 and Φ_2 ?
- Given properties of the program constituents, how can we combine them into a property of the global program (depending on the way constituents are composed)?

1.3 Compositionality, separate compilation and linkers

Another instance of the general scheme is given by separate compilation and linking.

If the system we consider is a *program*, written in a classical sequential language, and we call "properties" the *object code* generated by a compiler, then the general scheme for compositionality describes *separate compilation* and *link-editing*. Everybody knows that the executable code of a program written as a set of C source files (or modules) can be built from the set of separately compiled object files, without additional information about the internal structure of the C files. The intermediate form contains all information needed in order to merge to objects codes produced by compiling two source programs. In sequential languages like C, merging consists in putting together the definition of a function (or procedure) and all the calls to this function, which may appear in different source files. Merging is performed according to the *names* of the objects. Hence the intermediate *object code* must contain information about *names*. This *symbol table* is clearly something that is not needed any more in the executable code one wants to obtain at the end.

In this case "*Composition*" is the concatenation of source files (provided they do not define the same global objects) ; "*infer*" is the linking process.

This instance of the general scheme has the congruence property: there is no context that would allow to distinguish between two source files having the same objet code (this seems to be a very strong property, but one can define notions of equivalences for object code, that loosen the strict syntactic identity).

1.4 Compositionality and Synchronous Languages

In the whole process of designing, implementing and using a synchronous language [BB91], compositionality appears in several places: definition of the formal

semantics; issues related to separate compilation; compositional proof of properties.

Compositional semantics The synchrony hypothesis, which states that the program can be considered to react in zero time, is the key point for the compositionality of the semantics. Arguments are given in [BB91]. The idea is that a component may be replaced safely by the parallel composition of two components (for instance) without modifying the global reaction time, since $0 + 0 = 0$. This is illustrated in a very simple case in [Mar92]: the semantics of Argos is given by associating a single Boolean Mealy machine to an Argos program, which is a composition of several such machines (in parallel or hierarchically). The compositionality of the semantics means that the usual equivalences that can be defined for Boolean Mealy machines (e.g. bisimulation) are indeed *congruences* for the operations of the language.

Another important point for imperative synchronous languages is the notion of *causality*. It has long been considered as an intrinsically non compositional property of programs, meaning that, even if two components are causal, their parallel composition (with communication and potential feedback) is not necessarily causal. However, G. Berry gave a compositional definition of causality for Esterel [Ber95] and this notion can be adapted to other imperative synchronous languages like Argos [Mar92].

Separate compilation and linking for synchronous languages As far as the mixing of languages is concerned, the "linker" approach is the proper "multi-language" approach, in which each language involved has its own semantics and compiler, and there exists a common semantical model, which justifies the intermediate code format and the linking process.

This has been used for ArgoLus [JLMR93,JLMR94], or, more recently, for Argos+Lustre via DC [MH96]. DC is the common target format for the compilers of Lustre, Esterel and Argos. The linking approach is also used in the *Synchronie Workbench* [SWB] (see paper by Axel Poigné and Leszek Holenderski, same volume)

Allowing separate compilation of one (or several) synchronous languages amounts to defining the appropriate intermediate form between source programs and target code.

Synchronous languages all offer a notion of parallelism between components (be it explicit like in Esterel or Argos, or implicit like in Lustre and Signal). On the contrary, the target code is purely sequential. Hence a *scheduling* operation has to be performed somewhere between source programs and target code.

Choosing the appropriate intermediate form for separate compilation has a lot to do with this scheduling phase. In the DC format, the scheduling is not already performed, hence programs still have a parallel structure (DC is a dataflow language, hence the parallelism is intrinsic between nodes). The linking process, when performed at the DC level, consists in connecting several dataflow networks together. The scheduling phase is then performed globally on the linked

DC program, when translating DC into C, JAVA, or ADA sequential code (see paper by Albert Benveniste, Paul Le Guernic and Pascal Aubry, same volume) for details on the separate compilation of Signal programs, where the intermediate form is already (partially) scheduled.

Compositional proofs In the framework of synchronous languages, verification of safety properties is done using *synchronous observers* [HLR93]. Moreover, since synchronous languages are indeed *programming* languages for reactive systems, they all offer the explicit distinction between *inputs* and *outputs*. Therefore, the verification process always takes into account some assumptions about the *environment*. The classical verification scheme is the following:

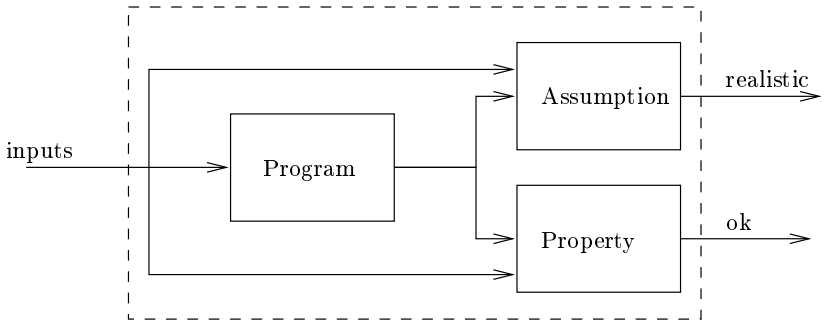


Fig. 1. verification with synchronous observers

The assumption on the behaviour of the environment is a synchronous program, which recognizes the correct sequences of inputs (assumptions do not need to be instantaneous. One can express, for instance, that two Boolean inputs alternate. Hence the assumption can have memory). As soon as the inputs become unrealistic, the output `realistic` is false, and remains false forever. The safety property is also encoded by a synchronous program that *observes* the inputs and outputs of the program to be verified, and outputs a single Boolean value `ok`. The key point is that *observing a program does not modify its behaviour*. This is possible because the components of synchronous programs communicate with each other by synchronous broadcast, which is asymmetric: emission is non blocking, and receivers may be added without modifying the emitter (this is clearly not the case with rendez-vous communication for instance). The only constraint to be respected is that the output of the observer should not be connected to the input of the program (no feedback).

The three programs are connected together and the resulting program is compiled. The verification tool then has to prove (it can be done by enumerative

methods or symbolically) that, as long as the output `realistic` is true, the output `ok` is also true.

This verification scheme allows to verify that a property ϕ holds for a program P under the assumption ψ on the environment of P. Hence, verifying that a property ϕ holds for a parallel program $P \| Q$ can be divided into two tasks: 1) find a property ψ holding for P, 2) prove that ϕ holds for Q under assumption ψ.

1.5 Another way of mixing synchronous languages

In this paper, we investigate another approach for mixing synchronous languages. This approach is not "multi-language", but rather "multi-style/single language". It's the approach taken in Oz [MMVR95] and a lot of so-called "multi-paradigm" languages, in which one can freely mix functional and imperative styles, constraint or concurrent programming and objects, etc. Of course this makes sense only if there is a common semantical model, but one has to define the semantics of this new language, i.e., mainly, the way two contructs from different programming styles interact with each other.

There is a particular case, when the new language is designed by mixing the constructs of two already existing languages, having their own semantics. In this case, one could reasonably hope that the new language is, in some sense, a *"conservative" extension* of each of the original languages. Meaning that, if one uses only the constructs from language A, in the mixing of A and B, then the new semantics coincides with the usual semantics of A.

In this paper, we study the mixing of Lustre and Argos, and three criteria we used in defining the semantics of this mixed language.

Overview of the paper

Section 2 briefly introduces the synchronous approach for the programming of reactive systems. Section 3 defines *mini-Lustre*, a small subset of Lustre which is sufficient for presenting our notion of mode-automaton, in which Lustre equations are associated to the states of an Argos program (Section 4). Section 5 lists three criteria for the semantics of these mode-automata, which are instances of the general compositionality scheme described above. Section 6 proposes two semantics of mode-automata, by translation into pure Lustre. Only one of them respects the criteria. Section 7 is the conclusion.

2 The Synchronous Approach for Reactive Systems

A reactive system has *inputs* and *outputs*. It reacts to inputs from what we call its *environment*, by sending outputs to it. The output that is sent to the environment at a given point in time may depend on the history of inputs, from the origin of time to the current instant.

In the *synchronous* approach, time is logical — the relation between this logical time and the physical time is not dealt with in the languages — and discrete. We can think of a sequence of *instants*, numbered with integers. A consequence of this notion of time is the ability to define the reaction of the system to the *absence* of some event in the environment.

At each instant n, the system reacts to input i_n by sending output o_n. In the general case, $o_n = f(i_0, ..., i_n)$, i.e. the current output depends on the whole history of inputs.

Note that o_n may not depend on further inputs ($i_{n+...}$), which represent the future; it may depend on i_n, i.e. the input at the same instant, which is called *synchrony hypothesis* [BG92].

The synchronous languages for reactive systems are restricted to systems for which the output at instant n depends on a bounded abstraction of the input history only. In other words, *bounded-memory* systems. In this case, $o_n = f(i_0, ..., i_n)$, where f is a function such that $\exists B. \mid \text{Image}(f) \mid < B$. And, of course, f must be computable incrementally, which means:

$$\exists h. \quad f(i_0, ..., i_n) = h(f(i_0, ..., i_{n-1}), i_n)$$

This allows to consider that a reactive system is always an instance of a very simple program of the form:

Initialize the memory M
while true do
    Acquire input i
    Emit output $o(M, i)$
    Update the memory: $M = h(M, i)$

The synchronous languages are designed for describing the *reactive kernel* of such a system, i.e. the *output function* o and the *transition function* h.

Argos and Lustre have very different styles for that. Argos is based upon the *explicit memory* paradigm: a program is a Mealy machine, which means that states and transitions are given in extension. Operations of the language include *parallel composition* and *hierarchic composition* of Mealy machines communicating by synchronous broadcast.

Lustre is based upon the *implicit memory* paradigm. The transition function h and the output function o are described by sets of equations (without instantaneous dependencies). The compiler is able to produce a Mealy machine from a Lustre program, if needed.

3 Mini-Lustre: a (very) Small Subset of Lustre

For the rest of the paper, we use a very small subset of Lustre. A program is a single node, and we avoid the complexity related to types as much as possible. In some sense, the mini-Lustre model we present below is closer to the DC [CS95] format used as an intermediate form in the Lustre, Esterel and Argos compilers.

Definition 1 (mini-Lustre programs). $N = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l, f, I)$ *where:*
$\mathcal{V}_i$, $\mathcal{V}_o$ *and* $\mathcal{V}_l$ *are pairwise disjoint sets of* input, output *and* local *variable names.* I *is a total function from* $\mathcal{V}_o \cup \mathcal{V}_l$ *to constants.* f *is a total function from* $\mathcal{V}_o \cup \mathcal{V}_l$ *to the set* $Eq(\mathcal{V}_i \cup \mathcal{V}_o \cup \mathcal{V}_l)$ *and* $Eq(V)$ *is the set of expressions with variables in* V*, defined by the following grammar:* $e ::= c \mid x \mid op(e, ..., e) \mid pre(x)$*.* c *stands for constants,* x *stands for a name in* V*, and* op *stands for all combinational operators. An interesting one is the conditional* `if` $\mathtt{e}_1$ `then` $\mathtt{e}_2$ `else` $\mathtt{e}_3$ *where* $\mathtt{e}_1$ *should be a Boolean expression, and* $\mathtt{e}_2$, $\mathtt{e}_3$ *should have the same type.* $pre(x)$ *stands for the* previous *value of the flow denoted by* x*. In case one needs* $pre(x)$ *at the first instant,* $I(x)$ *should be used.* 2

We restrict mini-Lustre to integer and Boolean values. All expressions are assumed to be typed correctly. As in Lustre, we require that the dependency graph between variables be acyclic. A dependency of X onto Y appears whenever there exists an equation of the form $X = ...Y...$ and Y does not appear inside a `pre` operator. In the syntax of mini-Lustre programs, it means that Y appears in the expression $f(X)$, not in a `pre` operator.

Definition 2 (Trace semantics of mini-Lustre). *Each variable name* v *in the mini-Lustre program describes a* flow *of values of its type, i.e. an infinite sequence* $v_0, v_1, ...$ *Given a sequence of inputs, i.e. the values* v_n*, for each* $v \in \mathcal{V}_i$ *and each* $n \geq 0$*, we describe below how to compute the sequences (or* traces*) of local and output flows of the program. The initialization function gives values to variables for the instant "before time starts", since it provides values in case* `pre`(x) *is needed at instant 0. Hence we can call it* x_{-1}*:*

$$\forall v \in \mathcal{V}_o \cup \mathcal{V}_l. \quad v_{-1} = I(v)$$

For all instants in time, the value of an output or local variable is computed according to its definition as given by f*:*

$$\forall n \geq 0. \quad \forall v \in \mathcal{V}_o \cup \mathcal{V}_l. \quad v_n = f(v)[x_n/x][x_{n-1}/pre(x)]$$

We take the expression $f(v)$*, in which we replace each variable name* x *by its current value* x_n*, and each occurrence of* `pre`(x) *by the previous value* x_{n-1}*. This yields an expression in which combinational operators are applied to constants. The set of equations we obtain for defining the values of all the flows over time is acyclic, and is a sound definition.* 2

Definition 3 (Union of mini-Lustre nodes). *Provided they do not define the same outputs, i.e.* $\mathcal{V}_o^1 \cap \mathcal{V}_o^2 = \emptyset$*, we can put together two mini-Lustre programs. This operation consists in connecting the outputs of one of them to the inputs of the other, if they have the same name. These connecting variables should be removed from the inputs of the global program, since we now provide definitions for them. This corresponds to the usual dataflow connection of two nodes.*

$(\mathcal{V}_i^1, \mathcal{V}_o^1, \mathcal{V}_l^1, f^1, I^1) \cup (\mathcal{V}_i^2, \mathcal{V}_o^2, \mathcal{V}_l^2, f^2, I^2) =$
$((\mathcal{V}_i^1 \cup \mathcal{V}_i^2) \setminus \mathcal{V}_o^1 \setminus \mathcal{V}_o^2, \quad \mathcal{V}_o^1 \cup \mathcal{V}_o^2, \quad \mathcal{V}_l^1 \cup \mathcal{V}_l^2,$
$\lambda x.\mathit{if}\ x \in \mathcal{V}_o^1 \cup \mathcal{V}_l^1\ \mathit{then}\ f^1(x)\ \mathit{else}\ f^2(x),$
$\lambda x.\mathit{if}\ x \in \mathcal{V}_o^1\ \mathit{then}\ I^1(x)\ \mathit{else}\ I^2(x))$

Local variables should be disjoint also, but we can assume that a renaming is performed before two mini-Lustre programs are put together. Hence $\mathcal{V}_l^1 \cap \mathcal{V}_l^2 = \emptyset$ *is guaranteed. The union of sets of equations should still satisfy the acyclicity constraint.* 2

Definition 4 (Trace equivalence for mini-Lustre). *Two programs* $L_1 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^1, f^1, I^1)$ *and* $L_2 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^2, f^2, I^2)$ *having the same input/output interface are trace-equivalent (denoted by* $L_1 \sim L_2$*) if and only if they give the same sequence of outputs when fed with the same sequence of inputs.*

2

Definition 5 (Trace equivalence for mini-Lustre with no initial specification). *We consider mini-Lustre programs without initial specification, i.e. mini-Lustre programs without the function I that gives values for the flows "before time starts". Two such objects* $L_1 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^1, f^1)$ *and* $L_2 = (\mathcal{V}_i, \mathcal{V}_o, \mathcal{V}_l^2, f^2)$ *having the same input/output interface are trace-equivalent (denoted by* $L_1 \approx L_2$*) if and only if,* for all initial configuration *I, they give the same sequence of outputs when fed with the same sequence of inputs.* 2

Property 1 : Trace equivalence is preserved by union
$L \sim L' \Longrightarrow L \cup M \sim L' \cup M$ 2

4 Proposal for a Mixed-Style Language

4.1 Motivations

We want to introduce *running modes* in a Lustre dataflow program, such that the switching of modes can be clearly identified, and described independently from the description of the modes. See [MR98] for a detailed introduction to the notion of *running modes*, and comparison with other works.

4.2 Mode-Automata

In [MR98] we propose a programming model called "*mode-automata*", made of: operations on automata taken from the definition of Argos [Mar92]; dataflow equations taken from Lustre [BCH+85].

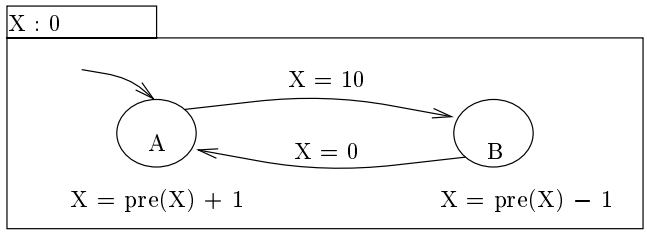


Fig. 2. Mode-automata: a simple example

Example The mode-automaton of figure 2 describes a program that outputs an integer X. The initial value is 0. Then, the program has two *modes*: an incrementing mode, and a decrementing one. Changing modes is done according to the value reached by variable X: when it reaches 10, the mode is switched to "decrementing"; when X reaches 0 again, the mode is switched to "incrementing".

Definition For simplicity, we give the definition for a simple case where the equations define only *integer* variables. One could easily extend this framework to all types of variables.

Definition 6 (Mode-automata).
A mode-automaton is a tuple $(Q, q_0, \mathcal{V}_i, \mathcal{V}_o, \mathcal{I}, f, T)$ *where:*

- Q *is the set of* states *of the automaton part*
- $q_0 \in Q$ *is the* initial *state*
- $\mathcal{V}_i$ *and* $\mathcal{V}_o$ *are sets of names for* input *and* output *integer* variables.
- $T \subseteq Q \times C(\mathcal{V}) \times Q$ *is the set of* transitions, *labeled by* conditions *on the variables of* $\mathcal{V} = \mathcal{V}_i \cup \mathcal{V}_o$
- $\mathcal{I} : \mathcal{V}_o \longrightarrow N$ *is a function defining the* initial *value of output variables*
- $f : Q \longrightarrow \mathcal{V}_o \longrightarrow \mathsf{EqR}$ *defines the labeling of states by total functions from* $\mathcal{V}_o$ *to the set* $\mathsf{EqR}(\mathcal{V}_i \cup \mathcal{V}_o)$ *of expressions that constitute the right parts of the* equations *defining the variables of* $\mathcal{V}_o$.

The expressions in $\mathsf{EqR}(\mathcal{V}_i \cup \mathcal{V}_o)$ *have the same syntax as in mini-Lustre nodes:* $e ::= c \mid x \mid op(e, ..., e) \mid \mathtt{pre}(x)$, *where* c *stands for constants,* x *stands for a name in* $\mathcal{V}_i \cup \mathcal{V}_o$, *and* op *stands for all combinational operators. The conditions in* $C(\mathcal{V}_i \cup \mathcal{V}_o)$ *are expressions of the same form, but without* `pre` *operators; the type of an expression serving as a condition is Boolean.* 2

Note that *Input* variables are used only in the right parts of the equations, or in the conditions. *Output* variables are used in the left parts of the equations, or in the conditions.

We require that the automaton part of a mode-automaton be *deterministic*, i.e., for each state $q \in Q$, if there exist two outgoing transitions (q, c_1, q_1) and (q, c_2, q_2), then $c_1 \wedge c_2$ is not satisfiable.
We also require that the automaton be *reactive*, i.e., for each state $q \in Q$, the formula $\bigvee_{(q,c,q') \in T} c$ is true.
With these definitions, the example of figure 2 is written as:

$$(\{A, B\}, A, \emptyset, \{X\}, I : X \to 0,$$
$$f(A) = \{\ \texttt{X = pre(X) + 1}\ \}, f(B) = \{\ \texttt{X = pre(X) - 1}\ \}),$$
$$\{(A, \texttt{X = 10}, B), (B, \texttt{X = 0}, A), (A, \texttt{X} \neq \texttt{10}, A), (B, \texttt{X} \neq \texttt{0}, B)\})$$

In the graphical notation of the example, we omitted the two loops $(A, \texttt{X} \neq \texttt{10}, A)$ and $(B, \texttt{X} \neq \texttt{0}, B)$.

5 Three Constraints on the Semantics, and the Associated Compositionality Criteria

The purpose of this section is not to detail the semantics and implementation of mode-automata. Before defining this semantics, we established a list of its desirable properties, and it appears that they can be viewed as instances of the general compositionality scheme given in section 1.1 above.

We could provide a new semantics, in any style. However, Lustre has a formal semantics in itself. Providing a translation of mode-automata into pure Lustre is therefore a simple way of defining their semantics. Section 6 defines the translation function $\mathcal{L}$. It gives a first translation, for which the properties hold. We also suggest another translation, for which one of the properties does not hold (section 6.3 below).

5.1 Parallel Composition of Mode-Automata

The question about parallel composition of mode-automata is an instance of the general compositionality scheme. Indeed:

- Consider that mode-automata are the systems we are interested in, and the Lustre programs we obtain as their semantics are the "properties".
- Putting two Lustre programs "together" makes sense (see Definition 3). This could be the "infer" process.
- Then, how should we define a composition of two mode-automata (denoted by $\|$) in such a way that the diagram commute? In other words, we should have:

$$\mathcal{L}(M_1 \| M_2) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$$

See Section 6.2 for the definition of $\mathcal{L}$ and Definition 3 for $\cup$.

Provided $\mathcal{V}_o^1 \cap \mathcal{V}_o^2 = \emptyset$, we define the parallel composition of two mode-automata by :

$$(Q^1, q_0^1, T^1, \mathcal{V}_i^1, \mathcal{V}_o^1, \mathcal{I}^1, f^1) \| (Q^2, q_0^2, T^2, \mathcal{V}_i^2, \mathcal{V}_o^2, \mathcal{I}^2, f^2) =$$
$$(Q^1 \times Q^2, (q_0^1, q_0^2), (\mathcal{V}_i^1 \cup \mathcal{V}_i^2) \setminus \mathcal{V}_o^1 \setminus \mathcal{V}_o^2, \mathcal{V}_o^1 \cup \mathcal{V}_o^2, \mathcal{I}, f)$$

Where:

$$f(q^1,q^2)(X) = \begin{cases} f^1(q^1)(X) \text{ if } X \in \mathcal{V}_o^1 \\ f^2(q^2)(X) \text{ otherwise, i.e. if } X \in \mathcal{V}_o^2 \end{cases}$$

Similarly:

$$\mathcal{I}(X) = \begin{cases} \mathcal{I}^1(X) \text{ if } X \in \mathcal{V}_o^1 \\ \mathcal{I}^2(X) \text{ if } X \in \mathcal{V}_o^2 \end{cases}$$

And the set T of global transitions is defined by:

$$(q^1,C^1,q'^1) \in T^1 \quad \wedge (q^2,C^2,q'^2) \in T^2 \qquad \Longrightarrow \quad ((q^1,q^2),C^1 \wedge C^2,(q'^1,q'^2)) \in T$$

5.2 Congruences of Mode-Automata

We try to define an equivalence relation for mode-automata, which is a congruence for the parallel composition. There are essentially two ways of defining such an equivalence :

- Either as a relation induced by the existing equivalence of Lustre programs (the trace equivalence)
- Or by an explicit definition on the structure of mode-automata, inspired by the trace equivalence of automata. The idea is that, if two states have equivalent sets of equations, then they can be identified.

Definition 7 (Induced equivalence of mode-automata).
$M_1 \equiv_i M_2 \iff \mathcal{L}(M_1) \sim \mathcal{L}(M_2)$ *(see definition 4 for $\sim$).* 2

Definition 8 (Direct equivalence of mode-automata). *The direct equivalence is a bisimulation, taking the labeling of states into account:*

$(Q^1,q_0^1,T^1,\mathcal{V}_i^1,\mathcal{V}_o^1,\mathcal{I}^1,f^1) \equiv_d (Q^2,q_0^2,T^2,\mathcal{V}_i^2,\mathcal{V}_o^2,\mathcal{I}^2,f^2) \iff$
$\exists R \subseteq R_s$ *such that:*
$(q_0^1,q_0^2) \in R \quad \wedge$

$$(q^1,q^2) \in R \quad \Longrightarrow \quad \begin{cases} (q^1,c^1,q'^1) \in T^1 \qquad \Longrightarrow \quad \exists q'^2,c^2 \text{ s. t. } (q^2,c^2,q'^2) \in T^2 \\ \qquad\qquad\qquad\qquad\qquad\quad \wedge\ (q'^1,q'^2) \in R \\ \qquad\qquad\qquad\qquad\qquad\quad \wedge\ c^1 = c^2 \\ \quad \textit{and conversely.} \end{cases}$$

Where $R_s \subseteq Q^1 \times Q^2$ is the relation on states induced by the equivalence of the attached sets of equations: $(q^1,q^2) \in R_s \iff f^1(q^1) \approx f^2(q^2)$ (see definition 5 for $\approx$). 2

These two equivalences probably do not coincide ($\equiv_i$ identifies more programs than $\equiv_d$), but both should be congruences for the parallel composition.

5.3 Argos or Lustre Expressed as Mode-automata

The last question is the following: can the model of mode-automata be considered as a conservative extension of both Lustre and Argos? i.e., for a given Lustre program L, can we build a mode-automaton M such that $\mathcal{L}(M) \sim L$; and, similarly, for a given Argos program A, can we build M such that $\mathcal{L}(M) \sim A$? (the equivalence between a Lustre and an Argos program is the identity of the sets of input/ouput traces).

The answer is trivial for Lustre: just build a single-state mode-automaton, and attach the whole Lustre program to the unique state. The semantics of mode-automata should ensure the property.

For Argos, it is less trivial, since we have to transform the Mealy style (inputs and outputs on the transitions) into the mode-automaton style. The key point is that, in the imperative style of Argos (it would be the same for Esterel), outputs are supposed to be absent if nobody emits them; in the equationnal style of mode-automata (inherited from Lustre), all signal values have to be explicitly defined.

Figure 3 gives an example. a, b, c, d are inputs, α and β are outputs. **(a)** is a Mealy machine (like the components of Argos programs). **(b)** is a mode-automaton that should have the same behaviour. Note that we obtain a particular form of mode-automata, where the equations attached to states make no use of the `pre` operator. As expected, all the "memory" is encoded into states.

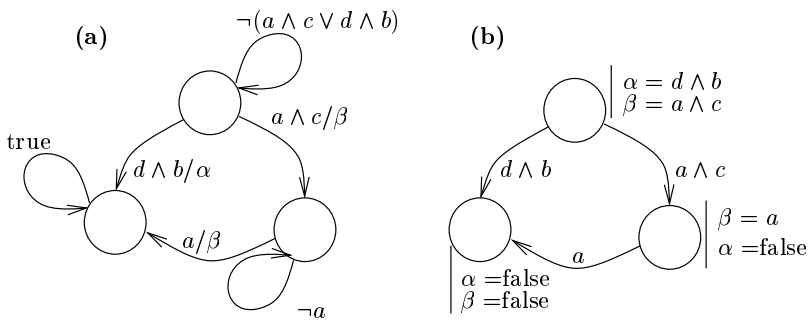


Fig. 3. Translating Argos into mode-automata

The translation of full Argos is intended to be structural: for instance, for an Argos program $P \| Q$ made of two Mealy machines P and Q, the corresponding mode-automata program should be the parallel composition of M_P and M_Q obtained respectively from the machines P and Q as described above.

6 A possible semantics

6.1 Intuition

The main idea is to translate the automaton structure into Lustre, in a very classical and straightforward way. Once the automaton structure is encoded, one has to relate the sets of equations to states. This is done by gathering all the sets of equations attached to states in a single conditional structure.

We give a possible translation below.

6.2 Translation into pure Lustre

The function $\mathcal{L}$ associates a mini-Lustre program with a mode-automaton. We associate a Boolean local variable with each state in $Q = \{q_0, q_1, ..., q_n\}$, with the same name. Hence:

$$\mathcal{L}((Q, q_0, \mathcal{V}_i, \mathcal{V}_o, \mathcal{I}, f, T)) = (\mathcal{V}_i, \mathcal{V}_o, Q, e, J)$$

The initial values of the variables in $\mathcal{V}_o$ are given by the initialization function $\mathcal{I}$ of the mode-automaton, hence $\forall x \in \mathcal{V}_o,\ J(x) = \mathcal{I}(x)$. For the local variables of the mini-Lustre program, which correspond to the states of the mode-automaton, we have: $J(q_0) = \text{true}$ and $J(q) = \text{false}, \forall q \neq q_0$.
The equations of the mini-Lustre program are obtained by:

for $x \in \mathcal{V}_o, e(x)$ is the expression:	for $q \in Q, e(q)$ is the expression:
if q_0 then $f(q_0)$ else if q_1 then $f(q_1)$ ... else if q_n then $f(q_n)$	$\bigvee_{(q',c,q)\in T} \text{pre}\ (q' \wedge c)$

The equation for a local variable q that encodes a state q expresses that we are in state q at a given instant if and only if we were in some state q', and a transition (q', c, q) could be taken. Note that, because the automaton is reactive, the system can always take a transition, in any state. A particular case is $q' = q$: staying in a state means taking a loop on that state, at each instant.

The mini-Lustre program obtained for the example is the following (note that `pre(A and X = 10)` is the same as `pre(A) and pre(X) = 10`, hence the equations have the form required in the definition of mini-Lustre).

$\mathcal{V}_i = \emptyset$ $\quad \mathcal{V}_o = \{X\}$ $\quad \mathcal{V}_l = \{A, B\}$
$f(X)$: `if A then pre(X)+1 else pre(X)-1`
$f(A)$: `pre (A and not X=10) or pre(B and X = 0)`
$f(B)$: `pre (B and not X=0) or pre(A and X = 10)`
$I(X) = 0$ $\quad I(A) = true$ $\quad I(B) = false$

6.3 Another translation

We could have chosen another translation of mode-automata into Lustre, by deciding that "transitions do not take time".

If we observe the semantics proposed above carefully, we see that, in the instant when the automaton changes states, the values of the output variables are updated according to the equations attached to the source state of the transition. We could have chosen to update the variables according to the equations attached to the *target* state of the transition (it yields a slightly different semantics).

But we could *not* have chosen not to update the variables during transitions. Indeed, if we choose this solution, the equivalence of mode-automata based upon the notion of bisimulation is no longer a congruence.

7 Conclusions

In the first part of this paper, we recalled three aspects of compositionality in the framework of synchronous languages: definition of the semantics ; separate compilation ; compositional proofs. These three points are illustrated by a large number of papers (we could not cite all of them) and by industrials tools.

The second part of the paper is an exercice, motivated by the need to talk about *running modes* in a dataflow synchronous language. The proposal consists in attaching dataflow equations to the states of an automaton, which represent *modes*. Defining the semantics of such objects is not difficult, since we can simply provide a translation into pure Lustre (we could also define the trace semantics directly). However, it is desirable that the semantics have some compositionality properties. We listed three of them. Requiring that these properties hold is a way to rule out some semantic proposals. For instance (section 5.1), a semantics in which the Lustre program associated to the parallel composition of two mode-automata is *not* the usual dataflow connection of the two individual Lustre programs would have no meaning in a multi-language framework. Similarly, a translation in which the variables of the dataflow part are not updated during the transitions of the automaton part ("transitions do not take time") cannot be accepted: the usual equivalence of automata is no longer a congruence for the parallel composition we have in mind.

References

[BB91] A. Benveniste and G. Berry. Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[BCH+85] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, San Diego, September 1985.

[Ber95] Gérard Berry. *The Constructive Semantics of Esterel.* Draft book `http://www.inria.fr/meije/esterel`, 1995.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[CS95] C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC version 1.0. Technical report, SYNCHRON project, October 1995.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[JLMR93] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. Embedding declarative subprograms into imperative constructs. In *Fifth International Symposium on Programming Language Implementation and Logic Programming,* Tallin, Estonia. Springer Verlag, LNCS 714, August 1993.

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *In 5th IEEE International Conference on Computer Languages*, Toulouse, May 1994. IEEE Computer Society Press.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. LNCS 630, Springer Verlag, August 1992.

[MH96] F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Formal Techniques for Real-Time and Fault Tolerance (FTRTFT)*, Uppsala (Sweden), September 1996. Springer verlag, LNCS 1135.

[MMVR95] Martin Müller, Tobias Müller, and Peter Van Roy. Multi-paradigm programming in Oz. In Donald Smith, Olivier Ridoux, and Peter Van Roy, editors, *Visions for the Future of Logic Programming: Laying the Foundations for a Modern successor of Prolog*, Portland, Oregon, 7 December 1995. A Workshop in Association with ILPS'95.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer Verlag.

[SWB] The synchronie workbench. `http://set.gmd.de/SET/ees_f.html` – GMD SET-EES, Schloss Birlinghoven, 53754 Sankt Augustin, Germany.

Compositional Reasoning using Interval Temporal Logic and Tempura

B. C. Moszkowski*

Department of Electrical and Electronic Engineering,
University of Newcastle upon Tyne, Newcastle NE1 7RU, Great Britain
email: Ben.Moszkowski@ncl.ac.uk

Abstract. We present a compositional methodology for specification and proof using Interval Temporal Logic (ITL). After given an introduction to ITL, we show how fixpoints of various ITL operators provide a flexible way to modularly reason about safety and liveness. In addition, some new techniques are described for compositionally transforming and refining ITL specifications. We also consider the use of ITL's programming language subset Tempura as a tool for testing the kinds of specifications dealt with here.

1 Introduction

Modularity is of great importance in computer science. Its desirability in formal methods is evidenced by the growing interest in *compositional* specification and proof techniques. Work by us over the last few years has shown that a powerful generalization of the increasing popular assumption/commitment approach to compositionality can be naturally embedded in *Interval Temporal Logic* (ITL) [12] through the use of temporal fixpoints [14]. Reasoning about safety, liveness and multiple time granularities are all feasible [15, 17].

In the present paper, we extend our methods to compositional transformation of specifications into other specifications. Basically, we show how to sequentially combine commitments containing specification fragments. The process continues until we have obtained the desired result. This is useful when verifying, say, the equivalence of two specifications. One sequentially transforms each specification into the other. The transformation techniques can also be applied to the refinement of relatively abstract specifications into more concrete programs.

We also show that various compositional ITL specification and proof techniques have executable variants. An interpreter for ITL's programming-language subset Tempura [13] serves as a prototype tool. Generally speaking, our approach represents theorems as Tempura programs annotated with temporal assertions over periods of time. This can be viewed as a generalization of the use of pre- and post-conditions as annotations for documenting and run-time checking of conventional sequential programs. Because our assertion language is an executable

* The research described here has been kindly supported by EPSRC research grant GR/K25922.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 439-464, 1998.
Springer-Verlag Berlin Heidelberg 1998

subset of ITL, we can specify and check for behavior over periods of time whereas conventional assertions are limited to single states.

The remaining sections of the paper are organized as follows. Section 2 gives a summary of ITL's syntax and semantics. In Sect. 3 we overview compositionality in ITL. Section 4 looks at compositional reasoning about liveness. Section 5 presents a compositional approach to transformation of specifications. Section 6 considers execution of compositional specifications. The appendix discusses a practical ITL axiom system for compositional proofs.

2 Review of Interval Temporal Logic

We now describe Interval Temporal Logic for finite time. The presentation is rather brief and the reader should refer to references such as [11, 3, 12, 14] for more details. Infinite intervals can also be handled by us but for simplicity we do not consider them until Subsect. 2.1. An ITL proof system is contained in the appendix.

ITL is a linear-time temporal logic with a discrete model of time. An interval σ in general has a length $|\sigma| \geq 0$ and a finite, nonempty sequence of $|\sigma| + 1$ states $\sigma_0, \ldots, \sigma_{|\sigma|}$. Thus the smallest intervals have length 0 and one state. Each state σ_i for $i \leq |\sigma|$ maps variables a, b, c, ..., A, B, C, ... to data values. Lower-case variables a, b, c, ... are called *static* and do not vary over time. Basic ITL contains conventional propositional operators such as $\wedge$ and first-order ones such as $\forall$ and $=$. Normally expressions and formulas are evaluated relative to the beginning of the interval. For example, the formula $J = I + 1$ is true on an interval σ iff the J's value in σ's initial state is one more that I's value in that state.

There are three primitive temporal operators *skip*, ";" (*chop*) and "*" (*chop-star*). Here is their syntax, assuming that S and T are themselves formulas:

$$\mathit{skip} \qquad S; T \qquad S^* \ .$$

The formula *skip* has no operands and is true on an interval iff the interval has length 1 (i. e., exactly two states). Both *chop* and *chop-star* permit evaluation within various subintervals. A formula $S; T$ is true on an interval σ with states $\sigma_0, \ldots, \sigma_{|\sigma|}$ iff the interval can be chopped into two sequential parts sharing a single state σ_k for some $k \leq |\sigma|$ and in which the subformula S is true on the left part $\sigma_0, \ldots, \sigma_k$ and the subformula T is true on the right part $\sigma_k, \ldots, \sigma_{|\sigma|}$. For instance, the formula *skip*; $(J = I+1)$ is true on an interval σ iff σ has at least two states $\sigma_0, \sigma_1, \ldots$ and $J = I+1$ is true in the second one σ_1. A formula S^* is true on an interval iff the interval can be chopped into zero or more sequential parts and the subformula S is true on each. An empty interval (one having exactly one state) trivially satisfies any formula of the form S^* (including false^*). The following sometimes serves as an alternative syntax for S^*:

$$\mathit{chopstar}\, S \ .$$

Figure 1 pictorially illustrates the semantics of *skip*, *chop*, and *chopstar*. Some simple ITL formulas together with intervals which satisfy them are shown in Fig. 2. Some further propositional operators definable in ITL are shown in Table 1.

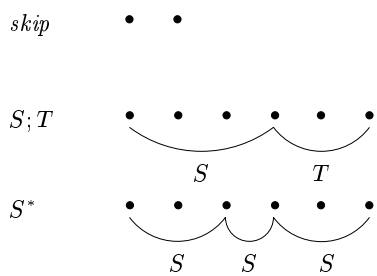


Fig. 1. Informal illustration of ITL semantics

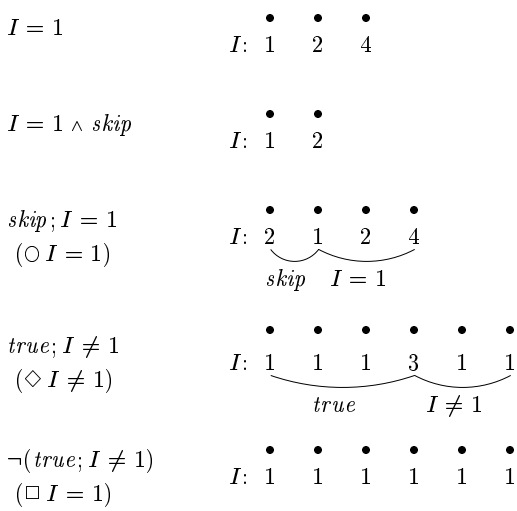


Fig. 2. Some sample ITL formulas and satisfying intervals

We generally use w, w', x, x' and so forth to denote *state formulas* with no temporal operators in them. Expressions are denoted by e, e' and so on.

In [14] we make use of the conventional logical notion of *definite descriptions* of the form $\imath v{:}\, S$ where v is a variable and S is a formula (see for example Kleene [7, pp. 167–171]). These allow a uniform semantic and axiomatic treatment in ITL of expressions such as $\bigcirc e$ (e's next value), *fin* e (e's final value)

Table 1. Some other definable propositional ITL operators

$\textcircled{w}\, S$	$\stackrel{\text{def}}{\equiv}$	$\neg \bigcirc \neg S$	Weak next
more	$\stackrel{\text{def}}{\equiv}$	$\bigcirc\, true$	Nonempty interval
empty	$\stackrel{\text{def}}{\equiv}$	$\neg more$	Empty interval
$\diamondsuit\, S$	$\stackrel{\text{def}}{\equiv}$	$S; true$	Some initial subinterval
$\boxdot\, S$	$\stackrel{\text{def}}{\equiv}$	$\neg \diamondsuit \neg S$	All initial subintervals
$\circledcirc\, S$	$\stackrel{\text{def}}{\equiv}$	$true; S; true$	Some subinterval
$\boxed{a}\, S$	$\stackrel{\text{def}}{\equiv}$	$\neg \circledcirc \neg S$	All subintervals
keep S	$\stackrel{\text{def}}{\equiv}$	$\boxed{a}(skip \supset S)$	All *unit* subintervals
fin S	$\stackrel{\text{def}}{\equiv}$	$\Box(empty \supset S)$	Final state
halt S	$\stackrel{\text{def}}{\equiv}$	$\Box(S \equiv empty)$	Exactly final state

and *len* (the interval's length). For example, $\bigcirc e$ can be defined as follows:

$$\bigcirc e \quad \stackrel{\text{def}}{=} \quad \imath a{:}\bigcirc(e = a) \ ,$$

where a does not occur freely in e. Here is a way to define temporal assignment using a *fin* term:

$$e \leftarrow e' \quad \stackrel{\text{def}}{\equiv} \quad (fin\ e) = e' \ .$$

The following operator *stable* tests whether an expression's value changes:

$$stable\ e \quad \stackrel{\text{def}}{\equiv} \quad \exists a{:}\Box(e = a) \ ,$$

where the static variable a is chosen so as not to occur freely in the expression e. The formula $e\ gets\ e'$ is true iff in every unit subinterval, the initial value of the expression e' equals the final value of the expression e:

$$e\ gets\ e' \quad \stackrel{\text{def}}{\equiv} \quad keep\,(e \leftarrow e') \ .$$

An expression is said to be *padded* iff it is stable except for possibly the last state in the interval:

$$padded\ e \quad \stackrel{\text{def}}{\equiv} \quad \exists a{:}\, keep\,(e = a) \ ,$$

where the static variable a does not to occur freely in e. A useful version of assignment called *padded temporal assignment* can then be defined:

$$e \lessapprox e' \quad \stackrel{\text{def}}{\equiv} \quad (fin\ e) = e' \wedge padded\ e \ .$$

This ensures that e does not change until possibly the very end of the interval when the assignment takes effect. Figure 3 shows examples of these operators.

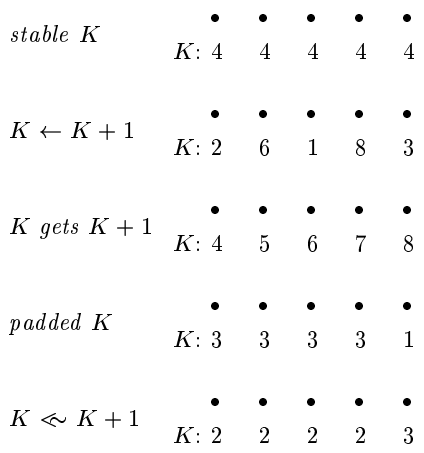


Fig. 3. Sample formulas illustrating *stable*, etc.

2.1 ITL with Infinite Time

The semantics so far presented is suitable for reasoning about finite intervals. We now discuss some modifications needed to permit infinite intervals as well. First, we apply our semantics of $S;T$ and S^* to infinite intervals. As before, $S;T$ is true on an interval if the interval can be divided into one part for S and another adjacent part for T and that S^* is true if the interval can be divided into a finite number of parts, each satisfying S. In addition, we now also let $S;T$ be true on an infinite interval which satisfies S. For such an interval, we can ignore T. Furthermore, we let S^* be true on an infinite interval that is divisible into a finite number of subintervals where the last one has infinite length and each satisfies S or alternatively into an infinite number of finite intervals each satisfying S. We define new constructs for testing whether an interval is infinite or finite, and alter the definition of $\Diamond$:

$$\begin{array}{lcllcl} \mathit{inf} & \stackrel{\text{def}}{\equiv} & \mathit{true};\mathit{false} & \mathit{finite} & \stackrel{\text{def}}{\equiv} & \neg\mathit{inf} \\ \Diamond S & \stackrel{\text{def}}{\equiv} & \mathit{finite};S & \mathit{sfin}\ S & \stackrel{\text{def}}{\equiv} & \Diamond(\mathit{empty} \wedge S)\ . \end{array}$$

Here *sfin* S is a strong version of *fin* S and is true only on finite intervals. In contrast, *fin* S is vacuously true on all infinite intervals. The first-order operators for *temporal assignment* and *padded temporal assignment* are redefined to deal with both finite and infinite intervals:

$$\begin{array}{lcl} e \leftarrow e' & \stackrel{\text{def}}{\equiv} & \mathit{finite} \supset (\mathit{fin}\ e) = e'\ , \\ e \lessapprox e' & \stackrel{\text{def}}{\equiv} & \mathit{finite} \wedge (\mathit{fin}\ e) = e' \wedge \mathit{padded}\ e\ . \end{array}$$

Our experience seems to suggest that it is preferable to define $e \leftarrow e'$ to be vacuously true on infinite intervals and to define $e \lessapprox e'$ to be false on them.

3 Introduction to Compositionality in ITL

Modularity is a desirable attribute of any formal method. One of the best known modular logical notations is Hoare logic [4]. It uses the important insight that proofs about the pre/post-condition behavior of a sequential program can be decomposed into subproofs of the program's parts. In ITL we can express a Hoare clause as a theorem about discrete intervals of time consisting of one or more states:

$$\vdash \quad w \wedge Sys \supset \mathit{fin}\ w' \ .$$

Here w and w' are state formulas containing no temporal operators and Sys is some arbitrary temporal formula we wish to reason about. The temporal formula $\mathit{fin}\ w'$ is true on an interval iff w' is true in the interval's final state.

The pre/post-condition approach is not particularly well suited for specifying and verifying systems in which ongoing and parallel behavior are important. However, this can be remedied through the addition of what are commonly known as *assumptions* and *commitments*. Francez and Pnueli [2] are the first to consider them and refer to them as *interface predicates*. The following implication shows the basic form of an ITL theorem incorporating an assumption As and a commitment Co:

$$w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w' \ .$$

Table 2 briefly describes the role of each logical variable in such an implication. This can be seen as an embedding of Jones' *rely* and *guarantee conditions* [5] in ITL. In Fig. 4, we show a graphical representation of the implication called a *proof outline*.

Table 2. Compositional specification of system Sys

$$w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w' \quad ,$$

where:

w:	state formula about initial state,
As:	*assumption* about *overall* interval,
Sys:	the system under consideration,
Co:	*commitment* about *overall* interval,
w':	state formula about final state.

In general As and Co can be arbitrary temporal formulas. However, when compositional reasoning about sequential parts of a system is needed, it is useful to select assumptions and commitments for which the following derived ITL proof rule is sound:

$$\frac{\begin{array}{l}\vdash\ w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w' \ , \\ \vdash\ w' \wedge As \wedge Sys' \supset Co \wedge \mathit{fin}\ w''\end{array}}{\vdash\ w \wedge As \wedge (Sys; Sys') \supset Co \wedge \mathit{fin}\ w'' \ .} \qquad (1)$$

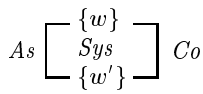


Fig. 4. Proof outline for specification *Sys*

The rule uses the ITL operator *chop* to combine the formulas *Sys* and *Sys'* sequentially. An associated proof outline is shown in Fig. 5. Here is an analogous rule for decomposing a proof for zero or more iterations of a formula *Sys*:

$$\frac{\vdash\ w \wedge As \wedge Sys\ \supset\ Co \wedge fin\ w}{\vdash\ w \wedge As \wedge Sys^*\ \supset\ Co \wedge fin\ w\ .} \tag{2}$$

Figure 6 shows a corresponding proof outline. Similar rules are possible for *if*, *while* and other constructs.

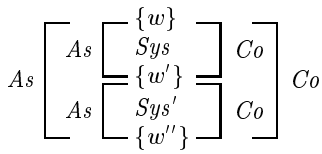


Fig. 5. Proof outline for specification *Sys*; *Sys'*

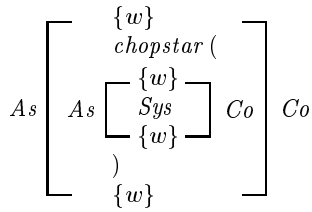


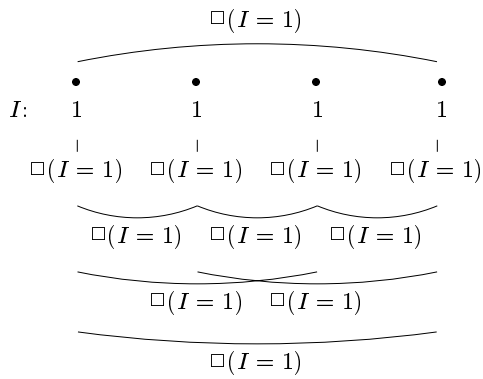
Fig. 6. Proof outline for specification *Sys**

To ensure soundness of proof rules 1 and 2, we require that *As* and *Co* be respective fixpoints of the ITL operators $\boxed{a}$ and *chop-star* as is now shown:

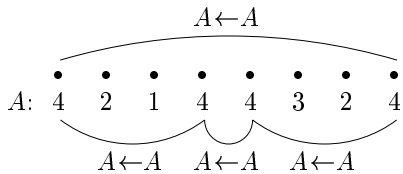
$$As \equiv \boxed{a}\, As\ , \qquad Co \equiv Co^*\ .$$

The first equivalence ensures that if the assumption *As* is true on an interval, it is also true in all subintervals. We say that such an assumption is *importable*.

The second equivalence ensures that if zero or more sequential instances of the commitment *Co* span an interval, *Co* is also true on the interval itself. A commitment with this property is said to be *exportable*. Importable assumptions and exportable commitments are collectively referred to as *sequentially compositional*. The temporal formula $\Box(I = 1)$ (read "*I always equals 1*") is a typical importable assumption. An example of its behavior can be pictorially represented as follows:



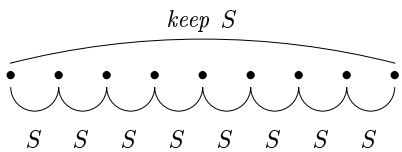
The set of importable assumptions turns out to consist exactly of those formulas expressible as $\boxed{a}\, S$ for some arbitrary subformula S. The temporal formula $A \leftarrow A$ ("*A's initial and final values on the interval are equal*") is an exportable commitment. Here is an interval illustrating this:



One can show that a formula is an exportable commitment if and only if it can be expressed in the form S^* for some arbitrary S. Some formulas such as *stable K* ("*K's value remains the same throughout the interval*") can be used both as assumptions and commitments. These are precisely the fixpoints of the ITL operator *keep* defined earlier in Table 1. We recall that formula *keep S*, for some subformula S, is defined to be true on an interval iff S is true on every unit subinterval (i.e., consisting of exactly two adjacent states):

$$keep\ S \stackrel{\text{def}}{\equiv} \boxed{a}(skip \supset S)\ .$$

Here is a graphical representation of the semantics of a formula *keep S* on a typical interval:



The formula $keep\ (K \leftarrow K+1)$ is an example of such a fixpoint. It states that K increases by 1 between every pair of adjacent states. In Fig. 7 we show a proof outline for the following lemma:

$$\begin{array}{lll} \vdash & J = 1 \wedge stable\ J \wedge (stable\ K; K \ll K + J) \\ & \supset & keep\ (K \leq \bigcirc K \leq K + 1) \wedge fin\ (J = 1)\ . \end{array} \tag{3}$$

$$stable\ J \left[\begin{array}{l} stable\ J \left[\begin{array}{l} \{J=1\} \\ stable\ K \\ \{J=1\} \end{array} \right] Co \\ stable\ J \left[\begin{array}{l} \{J=1\} \\ K \ll K+J \\ \{J=1\} \end{array} \right] Co \end{array} \right] Co$$

where Co is $keep\ (K \leq \bigcirc K \leq K + 1)$.

Fig. 7. Proof outline for lemma (3).

Note that our approach only requires that assumptions and commitments which are used directly in rules such (1) and (2) are sequentially compositional. Compositional proofs about a system in ITL typically also involve reasoning about assumptions and commitments which are not sequentially compositional. For instance, there is an important class of formulas using the standard temporal operator $\Box$. In general they can neither be used directly as sequentially compositional assumptions or commitments. Nevertheless, those of the form $\Box\, w$, for some state formula w, can be used as importable assumptions since they are fixpoints of the operator $\boxed{a}$:

$$\vdash \quad \Box\, w \equiv \boxed{a}\, \Box\, w\ .$$

Unfortunately, even these cannot be used as exportable commitments since, for example, the formula $(\Box\, w)^*$ (and indeed any formula S^*) is vacuously true on intervals having exactly one state whereas $\Box\, w$ is not necessarily true on them. In other words $(\Box\, w)^* \wedge \neg\, \Box\, w$ is satisfiable for some w and therefore $\Box\, w \equiv (\Box\, w)^*$ is normally not a theorem. However, there are simple ways around this. For instance, we can express $\Box\, w$ as the conjunction of $keep\ w$ and $fin\ w$:

$$\vdash \quad \Box\, w \equiv keep\ w \wedge fin\ w\ .$$

Since w is a state formula, $keep\ w$ turns out to be true on an interval iff w is true on all of the interval's states except possibly the last one. Since we already mentioned that $keep\ S$ for *any* formula S is a perfectly good exportable commitment, we can use $keep\ w$ in compositional proofs and at the very end combine it with $fin\ w$ to obtain the desired (generally nonexportable) commitment $\Box\, w$.

4 Compositional Analysis of Liveness

The techniques so far presented do not address reasoning about formulas involving liveness such as $\Box \Diamond x$ and $\Box(x \supset \Diamond x')$, where x and x' are state formulas. We now briefly discuss how to handle such temporal formulas in compositional proofs. More details and examples of proofs can be found in [17]. Let us now use the temporal operator $\boxdot$ ("*box-m*" or "*mostly*") defined as follows:

$$\boxdot S \stackrel{\text{def}}{\equiv} \Box(more \supset S) \ .$$

A formula $\boxdot S$ is true on an interval iff the subformula S is true on all terminal (suffix) subintervals with *more* than one state, that is all the interval's *nonempty* terminal subintervals. Therefore $\boxdot$ ignores the last (empty) terminal subinterval consisting of one state and is slightly weaker than $\Box$. In Fig. 8 we illustrate the difference between the two operators. On infinite intervals, their behavior is identical.

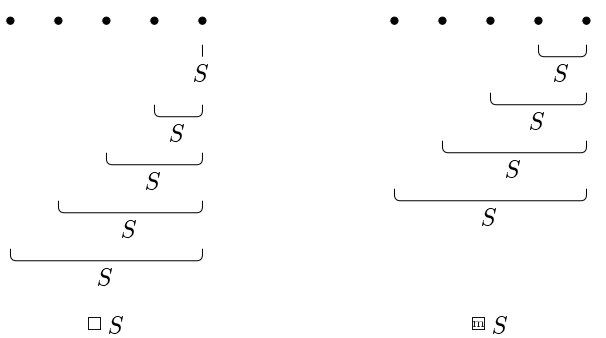


Fig. 8. Comparison of $\Box S$ with $\boxdot S$

It turns out that for any state formulas w and w' and an arbitrary formula S, the formula $\boxdot(w \supset S; w')$ is a fixpoint of *chop-star*:

$$\vdash \quad \boxdot(w \supset S; w') \equiv \big(\boxdot(w \supset S; w')\big)^* \ .$$

This is because $\boxdot(w \supset S; w')$ can be expressed as $\boxdot \diamondsuit\big(w \supset (S \wedge \mathit{fin}\ w')\big)$ and any formula of the form $\boxdot \diamondsuit T$ for some arbitrary formula T is a fixpoint of *chop-star*.

For state formulas x and x', the implication $x \supset \Diamond x'$ can be expressed as $x \supset \mathit{finite}; x'$. Consequently, the formula $\boxdot(x \supset \Diamond x')$ is a fixpoint of *chop-star*. Table 3 gives examples of exportable commitments expressible in the form $\boxdot(x \supset S; x')$ for suitable x, x' and S.

One way to prove a formula $\Box(x \supset \Diamond x')$, is by establishing the related formula $\boxdot(x \supset \Diamond x')$ through sequential composition and also showing the formula

Table 3. Examples of formulas expressible as $\boxplus(x \supset S; x')$

$\boxplus\, x$	
$\boxplus \Diamond x$	
$\boxplus(x \supset \Diamond x')$	
$\boxplus \diamondsuit(\mathit{skip} \wedge S)$	(same as $\mathit{keep}\ S$)

$\mathit{fin}\ (x \supset \Diamond x')$. We then use the following lemma relating $\Box$ with $\boxplus$ and fin:

$$\vdash \quad \Box S \equiv \boxplus S \wedge \mathit{fin}\ S \ .$$

The fixpoints of the ITL operator $\diamondsuit$ (read "*diamond-a*") are important when we reason about liveness. In general, $\diamondsuit S$ is true on an interval iff S is true on some subinterval (possibly the interval itself). Formulas such as $\Diamond x$, where x is a state formula, and $\neg\, \mathit{stable}\ A$ (meaning "*The variable A has more than one value over the interval*") are fixpoints of $\diamondsuit$. If DA is a fixpoint of $\diamondsuit$, $\boxplus\, DA$ is a fixpoint of *chop-star* and hence an exportable commitment. More generally, for any state formula x and $\diamondsuit$-fixpoint DA, a formula of the form $\boxplus(x \supset DA)$ is always a fixpoint of *chop-star*. This is because $\boxplus(x \supset DA)$ can be expressed as $\boxplus(x \supset DA; \mathit{true})$. The fixpoints of $\diamondsuit$ are closed under disjunction.

Let us consider another benefit of fixpoints of $\diamondsuit$. Suppose one wishes to prove that a formula $\mathit{Sys}; \mathit{Sys}'$ with a suitable precondition and an importable assumption implies a commitment $\boxplus(x \supset DA)$ for some state formula x and some fixpoint DA of $\diamondsuit$. The most straightforward thing to do is to first show the commitment both for Sys and Sys' and then combine the results using proof rule (1). However, this is not always possible since DA might never be true in Sys and only occur in Sys' even though x is perhaps somewhere true in Sys. In such cases, we can use the following derivable proof rule for all intervals, both finite and infinite:

$$\frac{\begin{array}{l}\vdash\ w \wedge As \wedge \mathit{Sys} \ \supset\ \mathit{finite} \wedge \mathit{fin}\ w' \ , \\ \vdash\ w' \wedge As \wedge \mathit{Sys}' \ \supset\ \boxplus(x \supset DA) \wedge DA \wedge \mathit{fin}\ w''\end{array}}{\vdash\ w \wedge As \wedge (\mathit{Sys}; \mathit{Sys}') \ \supset\ \boxplus(x \supset DA) \wedge DA \wedge \mathit{fin}\ w'' \ .} \tag{4}$$

This shows that the only thing we need to verify about Sys is that it terminates with the formula w' true in its final state. Both the desired commitment $\boxplus(x \supset DA)$ and DA itself can be obtained for $\mathit{Sys}; \mathit{Sys}'$ from Sys' alone because DA is a fixpoint of $\diamondsuit$. A proof outline for this is given in Fig. 9.

Figure 10 shows a proof outline for the following lemma in which the variable K is never stable, except trivially in the last state (if the overall interval is finite):

$$\begin{array}{rl}\vdash & J \geq 1 \wedge \mathit{keep}\ (J \leq \bigcirc J) \wedge \big((\mathit{stable}\ K \wedge \mathit{finite}); K \lessapprox K + J\big) \\ \supset & \boxplus \neg\, \mathit{stable}\ K \wedge \neg\, \mathit{stable}\ K \wedge \mathit{fin}\ (J \geq 1) \ .\end{array} \tag{5}$$

$$As \left[\begin{array}{l} As \left[\begin{array}{l} \{w\} \\ Sys \\ \{w'\} \end{array} \right] \; finite \\ As \left[\begin{array}{l} Sys' \\ \{w''\} \end{array} \right] \; \boxplus(x \supset DA) \wedge DA \end{array} \right] \boxplus(x \supset DA) \wedge DA$$

Fig. 9. A proof outline for rule (4)

$$keep\ (J \leq \bigcirc J) \left[\begin{array}{l} keep\ (J \leq \bigcirc J) \left[\begin{array}{l} \{J \geq 1\} \\ stable\ K \wedge finite \\ \{J \geq 1\} \end{array} \right] \; finite \\ keep\ (J \leq \bigcirc J) \left[\begin{array}{l} K \ll K + J \\ \{J \geq 1\} \end{array} \right] \; \boxplus \neg\ stable\ K \wedge \neg\ stable\ K \end{array} \right] \begin{array}{l} \boxplus \neg\ stable\ K \\ \wedge \neg\ stable\ K \end{array}$$

Fig. 10. Proof outline for lemma (5).

Sometimes a more powerful technique for analyzing reachability is needed. We originally introduced the notion of *markers* in [11, p. 127]. A marker is a boolean state variable, called here Mk, which is true exactly at the start and end of loop iterations. For example, a variant of *chop-star* having a marker can be defined as follows:

$$chopstar_{Mk}\, S \quad \stackrel{\text{def}}{\equiv} \quad (S \wedge \bigcirc halt\ Mk)^* \ .$$

Without loss of generality, we can always existentially introduce a marker as an auxiliary variable. The following provable lemma states this:

$$\vdash \quad S^* \equiv \exists Mk\colon (Mk \wedge chopstar_{Mk}\, S) \ ,$$

where Mk does not occur freely in the formula S. The use of markers in liveness proofs is discussed in more detail in [17].

5 Compositional Transformation of Specifications

Assumptions and commitments are usually thought of as being simpler than the systems they describe. However in ITL it is possible to embed arbitrary formulas in them. This provides a framework for compositional transformation and refinement of specifications. For example, we can specify that one system *Sys* implies that whenever some state formula x is true, the behavior of another system Sys' is observed followed by another state formula x' being true:

$$w \wedge As \wedge Sys \quad \supset \quad \Box(x \ \supset \ Sys'; x') \wedge fin\ w' \ .$$

The use of formulas of the form $\Box(x \ \supset \ S; x')$ provides a powerful means for decomposition. For example, suppose we wish to establish the following commitment which embeds $S; S'$:

$$\Box(x \ \supset \ S; S'; x') \ .$$

This can be split into two smaller commitments for S and S' using the general ITL theorem shown below:

$$\vdash \quad \Box(x \supset S; y) \wedge \Box(y \supset S'; x') \quad \supset \quad \Box(x \supset S; S'; x') \ . \tag{6}$$

Here we introduce a new state formula y to connect the two individual commitments. A similar decomposition theorem can be used for while-loops which are themselves expressible in ITL as follows:

$$\mathit{while}\ w\ \mathit{do}\ S \ \stackrel{\text{def}}{\equiv}\ (w \wedge S)^* \wedge \mathit{fin}\ \neg w \ .$$

A commitment with an embedded while-loop has the following form:

$$\Box\big(x \supset (\mathit{while}\ w\ \mathit{do}\ S); x'\big) \ .$$

It can be broken down using the theorem now given:

$$\begin{array}{ll} \vdash & \Box\big(x \wedge w \supset (S \wedge \mathit{more}); x\big) \wedge \Box\big(x \wedge \neg w \supset x'\big) \\ & \quad \supset \quad \Box\big(x \supset (\mathit{while}\ w\ \mathit{do}\ \mathit{Sys}); x'\big) \ . \end{array}$$

The formula x serves as the while-loop's invariant. Here is a corollary of this for introducing a while-loop itself:

$$\begin{array}{ll} \vdash & \Box\big(x \wedge w \supset (S \wedge \mathit{more}); x\big) \wedge \Box\big(x \wedge \neg w \supset \mathit{empty}\big) \\ & \quad \supset \quad x \supset (\mathit{while}\ w\ \mathit{do}\ \mathit{Sys}) \wedge \mathit{fin}\,(x \wedge \neg w) \ . \end{array} \tag{7}$$

Sometimes, we wish to prove that one system implies another:

$$w \wedge \mathit{As} \wedge \mathit{Sys} \quad \supset \quad \mathit{Sys}' \wedge \mathit{fin}\ w' \ .$$

This can be thought of as stating the existence of a transformation from *Sys* to *Sys'*. If we have already compositionally demonstrated a commitment $\Box(x \supset S; x')$, we can obtain S from it through the next theorem:

$$\vdash \quad x \wedge \Box(x \supset S; x') \wedge \Box(x' \supset \mathit{empty}) \quad \supset \quad S \ .$$

A commitment expressed as $\Box(x \supset S; x')$ is in general not exportable. However, we noted in Sect. 4 that a formula such as $\boxplus(x \supset S; x')$ when used as a commitment is exportable since it is always a fixpoint of *chop-star*. This greatly facilitates modular proofs since we obtain the benefits of sequential compositionality. The following lemmas assist in moving between the two types of commitments:

$$\begin{array}{ll} \vdash & \Box(x \supset S; x') \quad \supset \quad \boxplus(x \supset S; x') \\ \vdash & \boxplus(x \supset S; x') \wedge \mathit{fin}\ \neg x \quad \supset \quad \Box(x \supset S; x') \ . \end{array}$$

The subformula $\mathit{fin}\ \neg x$ in the second lemma ensures that the implication $x \supset S; x'$ is trivially true in the interval's final state if the interval is finite.

5.1 An Example

Figure 11 shows two logically equivalent specifications $p1(K,n)$ and $p2(K,n)$ which monotonically increase a variable K until it equals $2n$. Here is the behavior of K and n in a sample interval having 12 states:

	•	•	•	•	•	•	•	•	•	•	•	•
n:	3	3	3	3	3	3	3	3	3	3	3	3
K:	0	0	1	2	2	2	3	4	5	5	5	6

$$
\begin{array}{ll}
p1(K,n): & \mathit{while}\ K \neq 2n\ \mathit{do}\ (K \ll\sim K+1) \\
p2(K,n): & \mathit{halt}\,(K = 2n) \\
& \quad \wedge \big(K \ll\sim K+1;\ \mathit{halt}\ \mathit{even}(K)\big)^* \\
& \quad \wedge \big(\mathit{halt}\ \mathit{odd}(K);\ K \ll\sim K+1\big)^*
\end{array}
$$

Fig. 11. Two equivalent specifications

We will consider how to establish equivalence when K is initially even. This can be reduced to proving the following two implications:

$$\vdash \quad \mathit{even}(K) \wedge p1\,(K,n) \quad \supset \quad p2\,(K,n) \tag{8}$$

$$\vdash \quad \mathit{even}(K) \wedge p2\,(K,n) \quad \supset \quad p1\,(K,n)\ . \tag{9}$$

Each of these is analyzed individually.

Proof of $\mathit{even}(K) \wedge p1(K,n) \supset p2(K,n)$. In order to prove lemma (8), we give names to $p2$'s conjuncts as shown in Table 4 and prove the following lemmas which demonstrate that $p1$ implies each of them:

$$\vdash \quad \mathit{even}(K) \wedge p1\,(K,n) \quad \supset \quad p2a(K,n) \tag{10}$$

$$\vdash \quad \mathit{even}(K) \wedge p1\,(K,n) \quad \supset \quad p2b(K) \tag{11}$$

$$\vdash \quad \mathit{even}(K) \wedge p1\,(K,n) \quad \supset \quad p2c(K)\ . \tag{12}$$

The simplest of the three lemmas is the first one (10). A proof outline is shown in Fig. 12. It uses the following equivalence for the *halt* construct:

$$\vdash \quad \mathit{halt}\ w \quad \equiv \quad \boxplus \neg w \wedge \mathit{fin}\ w\ . \tag{13}$$

The proofs of lemma (11) for $p2b$ and lemma (12) for $p2c$ are similar to each other so we only look at the one for $p2b$. The lemma's proof uses an auxiliary boolean state variable X which is acts as follows:

$$X \wedge X\ \mathit{gets}\ (K \neq \bigcirc K)\ .$$

Table 4. Decomposition of specification $p2(K, n)$

$$
\begin{array}{ll}
p2(K,n): & p2a(K,n) \wedge p2b(K) \wedge p2c(K) \\
p2a(K,n): & halt\,(K = 2n) \\
p2b(K): & \big(K \lessapprox K+1; halt\ even(K)\big)^* \\
p2c(K): & \big(halt\ odd(K); K \lessapprox K+1\big)^*
\end{array}
$$

$$
\begin{array}{l}
\{true\} \\
while\, K \neq 2n\ do\ (\\
\quad \left.\begin{array}{l} \{K \neq 2n\} \\ K \lessapprox K+1 \\ \{true\} \end{array}\right] \boxplus K \neq 2n \\
) \\
\{K = 2n\}
\end{array}
\left[\begin{array}{l} \\ \\ \boxplus K \neq 2n \end{array}\right]
\left[\begin{array}{l} \\ \\ \boxplus K \neq 2n \\ \wedge\ fin\,(K = 2n) \end{array}\right]
\left[\begin{array}{l} \\ \\ halt\,(K = 2n) \end{array}\right]
$$

Fig. 12. Proof outline for lemma (10).

It is initially true and subsequently is true exactly whenever K's value changes. We can introduce X without loss of generality using existential quantification. Here is the behavior of K and X in the interval described earlier:

	•	•	•	•	•	•	•	•	•	•	•	•
K:	0	0	1	2	2	2	3	4	5	5	5	6
X:	*true*	*false*	*true*	*true*	*false*	*false*	*true*	*true*	*true*	*false*	*false*	*true*

The subformula X *gets* $(K \neq \bigcirc K)$ is used as an importable assumption in the proofs for *p2b*.

The outermost operator used in *p2b* is *chop-star*. The following general theorem provides a way to introduce a formula S^* from a commitment which embeds S in it:

$$x \wedge \boxplus\big(x \supset (S \wedge more); x\big) \quad \supset \quad S^* \ .$$

In the case of *p2b*, we use $X \wedge even(K)$ as an instance of x and take the following as an instance of $\boxplus\big(x \supset (S \wedge more); x\big)$:

$$
\begin{array}{l}
\boxplus\Big(X \wedge even(K) \supset \\
\qquad \big((K \lessapprox K+1; halt\ even(K)) \wedge more\big); \\
\qquad \big(X \wedge even(K)\big)\Big) \ .
\end{array}
$$

This can be further split into two commitments using a variant of lemma (6) given below for sequentially decomposing loop bodies:

$$
\begin{array}{ll}
\vdash & \boxplus\big(x \supset (S \wedge more); x'\big) \wedge \Box(x' \supset S'; x'') \\
& \quad \supset \quad \boxplus\big(x \supset ((S; S') \wedge more); x''\big) \ .
\end{array}
$$

One commitment is for $K \ll\sim K+1 \wedge more$ and the other is for $halt\ even(K)$:

$$\boxplus\Big(X \wedge even(K)\ \supset\ (K \ll\sim K+1 \wedge more);\ \big(X \wedge odd(K)\big)\Big) \qquad \Box\Big(X \wedge odd(K)\ \supset\ halt\ even(K);\ \big(X \wedge even(K)\big)\Big)\ .$$

The associated lemmas are now given:

$$\begin{array}{lll} \vdash & even(K) \wedge X \wedge X\ gets\ (K \neq \bigcirc K) \wedge P1(K,n) & \\ \supset & \boxplus\Big(X \wedge even(K)\ \supset\ (K \ll\sim K+1 \wedge more);\ \big(X \wedge odd(K)\big)\Big) & (14) \\ \vdash & even(K) \wedge X \wedge X\ gets\ (K \neq \bigcirc K) \wedge P1(K,n) & \\ \supset & \Box\Big(X \wedge odd(K)\ \supset\ halt\ even(K);\ \big(X \wedge even(K)\big)\Big)\ . & (15) \end{array}$$

Proof outlines for these are shown in Figs. 13 and 14, respectively. Figure 15 summarizes the overall proof of lemma (11).

$$As \left[\begin{array}{l} \{X\} \\ while\ K \neq 2n\ do\ (\\ \quad As \left[\begin{array}{l} \{X \wedge K \neq 2n\} \\ K \ll\sim K+1 \\ \{X\} \end{array}\right] \left[\begin{array}{l} even(K)\ \supset \\ \quad (K \ll\sim K+1 \wedge more); \\ \quad \big(X \wedge odd(K)\big) \end{array}\right] \\ \qquad \wedge\ \textcircled{w}\ \boxplus \neg X \\ ) \\ \{X\} \end{array}\right] Co \Big] Co$$

where As is $X\ gets\ (K \neq \bigcirc K)$

and Co is $\boxplus\Big(X \wedge even(K)\ \supset\ (K \ll\sim K+1 \wedge more);\ \big(X \wedge odd(K)\big)\Big)$.

Fig. 13. Proof outline for lemma (14).

Proof of $even(K) \wedge p2(K,n)\ \supset\ p1(K,n)$. In order to obtain $p1(K,n)$ from $p2(K,n)$, we first use the fact that $p1(K,n)$ is expressed as a while-loop and can therefore be decomposed using the lemma now given which is provable from corollary (7):

$$\vdash\quad halt\ \neg w \wedge S^*\quad \supset\quad while\ w\ do\ S\ . \tag{16}$$

$$
As \left[\begin{array}{l}
\{X\} \\
\mathit{while}\ K \neq 2n\ \mathit{do}\ (\\
\quad As \left[\begin{array}{l} \{X \wedge K \neq 2n\} \\ K \ll\sim K+1 \\ \{X\} \end{array}\right] \\
) \\
\{X \wedge K = 2n\}
\end{array}\right]
\left[\begin{array}{l}
\mathit{odd}(K)\ \supset \\
\quad \mathit{halt}\ \mathit{even}(K); \\
\quad (X \wedge \mathit{even}(K)) \\
\wedge\ \textcircled{w}\ \boxplus \neg X
\end{array}\right] Co
\left[\begin{array}{l}
Co \\
\wedge\ \mathit{fin}\ \neg \mathit{odd}(K)
\end{array}\right] Co'
$$

where As is $X\ \mathit{gets}\ (K \neq \bigcirc K)$,

Co is $\boxplus\big(X \wedge \mathit{odd}(K)\ \supset$
$\mathit{halt}\ \mathit{even}(K);$
$(X \wedge \mathit{even}(K))\big)$

and Co' is $\Box\big(X \wedge \mathit{odd}(K)\ \supset$
$\mathit{halt}\ \mathit{even}(K);$
$(X \wedge \mathit{even}(K))\big)$.

Fig. 14. Proof outline for lemma (15).

$$\mathit{even}(K) \wedge X \wedge X\ \mathit{gets}\ (K \neq \bigcirc K) \wedge p1(K,n)$$

a: $\supset$

$$\boxplus\big(X \wedge \mathit{even}(K)\ \supset (K \ll\sim K+1 \wedge \mathit{more});\ (X \wedge \mathit{odd}(K))\big) \quad \bigwedge \quad \Box\big(X \wedge \mathit{odd}(K)\ \supset \mathit{halt}\ \mathit{even}(K);\ (X \wedge \mathit{even}(K))\big)$$

b: $\supset$

$$\boxplus\big(X \wedge \mathit{even}(K)\ \supset ((K \ll\sim K+1; \mathit{halt}\ \mathit{even}(K)) \wedge \mathit{more});\ (X \wedge \mathit{even}(K))\big)$$

c: $\supset$

$$X \wedge \mathit{even}(K)\ \supset (K \ll\sim K+1; \mathit{halt}\ \mathit{even}(K))^*$$

Fig. 15. Overview of proof of lemma (8)

Here is the particular instance of this that we need to show:

$$\vdash \quad \mathit{halt}\ \neg(K \neq 2n) \wedge (K \lessapprox K+1)^* \quad \supset \quad \mathit{while}\ K \neq 2n\ \mathit{do}\ (K \lessapprox K+1)\ .$$

The antecedent of this can be obtained from $p2\,(K, n)$ in the following manner:

$$\vdash \quad \mathit{even}(K) \wedge p2\,(K, n)\ \supset\ \mathit{halt}\ \neg(K \neq 2n) \wedge (K \lessapprox K+1)^*\ . \tag{17}$$

The main work in proving lemma (17) involves obtaining $(K \lessapprox K+1)^*$:

$$\vdash \quad \mathit{even}(K) \wedge p2\,(K, n)\ \supset\ (K \lessapprox K+1)^*\ . \tag{18}$$

As done previously, we use an auxiliary variable X which is initially true and subsequently is true exactly whenever K's value changes. Figure 16 summarizes the overall proof of lemma (18).

$$\begin{array}{c}
\mathit{even}(K) \wedge X \wedge X\ \mathit{gets}\ (K \neq \bigcirc K) \wedge p2b(K) \qquad \mathit{even}(K) \wedge X \wedge X\ \mathit{gets}\ (K \neq \bigcirc K) \wedge p2c(K) \\
\supset \qquad\qquad\qquad\qquad\qquad\qquad \supset \\
\boxplus\big(X \wedge \mathit{even}(K)\ \supset\ (K \lessapprox K+1 \wedge \mathit{more}); X\big) \quad \wedge \quad \boxplus\big(X \wedge \mathit{odd}(K)\ \supset\ (K \lessapprox K+1 \wedge \mathit{more}); X\big) \\
\supset \\
\boxplus\big(X\ \supset\ (K \lessapprox K+1 \wedge \mathit{more}); X\big) \\
\supset \\
X\ \supset\ (K \lessapprox K+1)^*
\end{array}$$

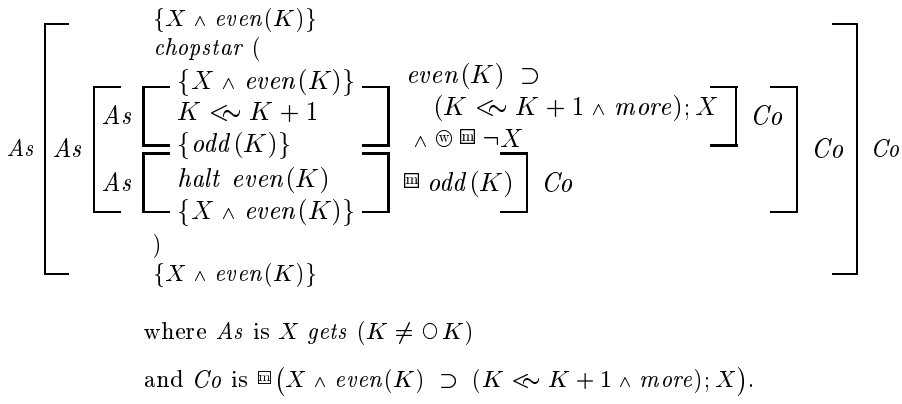
Fig. 16. Overview of proof of lemma (18)

A proof outline for the following lemma about $p2b(K)$ is shown in Fig. 17.

$$\begin{array}{ll}
\vdash & \mathit{even}(K) \wedge X \wedge X\ \mathit{gets}\ (K \neq \bigcirc K) \wedge p2b(K) \\
\supset & \boxplus\big(X \wedge \mathit{even}(K)\ \supset\ (K \lessapprox K+1 \wedge \mathit{more}); X\big)\ .
\end{array} \tag{19}$$

6 Executable Compositional Specifications

The Tempura programming language [13] is based on an *executable* subset of ITL. With some care, many interesting ITL specifications can be directly run by a Tempura interpreter. This consequently provides a valuable tool for "hands-on" access to ITL. It appears to be worthwhile to explore ways of exploiting Tempura



where As is X *gets* $(K \neq \bigcirc K)$

and Co is $\boxdot\big(X \wedge even(K) \supset (K \ll K+1 \wedge more); X\big)$.

Fig. 17. Proof outline for lemma (19).

for testing executable specifications which have assumptions and commitments in them. At present we are experimenting with various Tempura programming styles and interpreter implementation techniques to improve facilities for carrying this out. For instance, consider the following simple Tempura conjunction:

$$K = 0 \wedge p1\,(K, 3) \wedge p2(K, 3) \wedge \Box\, output(K) \quad . \tag{20}$$

This initializes the variable K to 0 and runs $p1\,(K, 3)$ and $p2(K, 3)$ in parallel on a state-by-state basis. Furthermore, the value of K in each state is displayed. Figure 18 shows a typical run. The construct $K \ll K + 1$ is nondeterministic since it does not specifying any particular interval length. The interpreter therefore generates a pseudo-random value in some user-adjustable range. We note that the Tempura source code for *p1* and *p2* is annotated with many of assumptions and commitments described earlier in various proofs in Subsect. 5.1. Therefore the run shown in Fig. 18 also extensively checks them.

Now consider the following general ITL formula containing an assumption and a commitment:

$$w \wedge As \wedge Sys \;\supset\; Co \wedge fin\ w' \quad .$$

This can sometimes be *tested* in Tempura for inconsistencies using the following conjunction:

$$w \wedge As \wedge Sys \wedge Co \wedge fin\ w' \quad .$$

Of course, it is not feasible to attempt to execute arbitrary assumptions and commitments. Here are two reasons why:

- They can contain arbitrary undecidable first-order ITL subformulas.
- Satisfiability can be nonelementary even for decidable propositional ITL formulas (Kozen in [11, p. 24]).

However, there are interesting and useful classes. For example, Table 5 shows various importable assumptions which can be tested. Similarly, Table 6 contains

```
Tempura 4> run (K=0 and p1(K,3) and p2(K,3) and always output(K)).
State   0: K=0
State   1: K=0
State   2: K=0
State   3: K=1
State   4: K=1
State   5: K=2
State   6: K=2
State   7: K=2
State   8: K=3
State   9: K=3
State  10: K=4
State  11: K=5
State  12: K=5
State  13: K=5
State  14: K=6

Done!  Computation length:  14.  Total Passes:  40.
```

Fig. 18. Sample Tempura output for formula (20)

a number of checkable exportable commitments. Indeed, we discovered that formulas having the form $\boxplus(w \supset S; w')$ were suitable as exportable commitments only after we tried to prove compositionally the equivalence of some experimental Tempura specifications. Many assumptions and commitments which are not sequentially compositional can also be handled by Tempura. Examples include commitments of the form $\Box(w \supset S; w')$ as long as w, S and w' are themselves executable. We are even investigating ways of implementing negation of suitable Tempura programs. This would permit empirical testing of the validity of an implication of the form $Sys \supset Sys'$ by examining satisfiability of a program such as $Sys \wedge \neg Sys'$.

Table 5. Some executable importable assumptions

stable A	A's value remains stable
keep $(K \leq \bigcirc K \leq K + 1)$	K's value weakly increases monotonically
$\Box(K = 0)$	K always equals 0
$\Box \Diamond(K = 1 \vee \mathit{empty})$	Always eventually either K equals 1 or the interval terminates

Table 6. Some executable exportable commitments

stable A	A's value remains stable
keep $(K \leq \bigcirc K \leq K+1)$	K's value weakly increases monotonically
$\boxdot(K=0)$	K's value is *mostly* zero
$\boxdot \Diamond K=1$	K's value is *mostly* sometimes 1
$\boxdot(K=j \supset \Diamond K=j+1)$	*Mostly* when $K=j$, eventually $K=j+1$
$\boxdot \exists i\colon \big(i=A \wedge \Diamond(A \neq i)\big)$	A is *mostly* not stable
$\boxdot(w \supset S; w')$	*Mostly* w implies S then w'

Let us now enumerate some benefits of using Tempura for testing compositional specifications:

- Tempura offers a "learning-by-doing" approach to ITL.
- Larger ITL specifications can be developed and tested than with pencil and paper alone.
- Modular, reusable Tempura test suites can be developed.
- Several specifications can be compared over a range of test data.
- The use of specialized theorem provers and model checkers can be postponed until after a preliminary run-time consistency check of candidate specifications and proofs.
- In contrast to model checking, execution can be used to test theorems which are not decidable.
- ITL and Tempura both improve through the increased feedback between theory and practice. Particular benefits are:
  - The discovery of further executable assumptions and commitments.
  - The development of more and better compositional proof techniques.
- Interval Temporal Logic serves as the single unifying formalization at all stages of analysis.

Of course, we do not realistically expect the use of an interpreter to replace theorem provers and model checkers. However, this does seem to be an intriguing alternative suitable in various circumstances. For example, we already mentioned that the compositional equivalence proofs discussed in this paper have been partially checked using a Tempura interpreter. We have also been able to do a run-time parallel check of seven different ITL specifications for doing a breadth-first walk down a tree. The specifications range from a register-transfer level description to a somewhat object-oriented approach based on parallel recursive decent by several processes. Furthermore, we checked some safety and liveness proofs for mutual exclusion presented in [17]. As time goes on, we hope to obtain more experience with the advantages and limitations of using Tempura for run-time checking of ITL assertions.

7 Discussions

We have presented the basis of a compositional methodology of specification and proof using fixpoints of various ITL operators. Issues considered include reasoning about safety, liveness and even equivalence of specifications. Our current work has identified an interesting class of commitments which can be used for compositional transformation and refinement of specifications. The exploitation of executable specifications based on ITL's programming language subset Tempura helps to accelerate development of both the underlying theory as well as practical tool support.

Much work remains to be done. We need to conduct larger case studies using by ITL and Tempura to ensure scalability of the techniques. Also, at present there is little experience with using compositionality in ITL together with a frame semantics for imperative destructive assignments developed by us in [16]. Furthermore, programming with Tempura has some difficulties. In particular, support for debugging of parallel programs needs improvement.

Acknowledgements

We wish to thank Antonio Cau, Nick Coleman, Li Xiaoshan and Hussein Zedan for discussions. The Engineering and Physical Sciences Research Council kindly funded our research.

References

1. Dutertre, B.: On first order interval temporal logic. In: 10th Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos, California (1995) 36–43
2. Francez, N., Pnueli, A.: A proof method for cyclic programs. Acta Inf. **9** (1978) 133–157
3. Halpern J., Manna Z., Moszkowski B.: A hardware semantics based on temporal intervals. In: Diaz, J. (Ed.) Proceedings of the 10th International Colloquium on Automata, Languages and Programming (ICALP'83). Lecture Notes in Computer Science Vol. 154. Springer-Verlag, Berlin Heidelberg New York (1983) 278–291
4. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12** (1969) 576–580, 583
5. Jones, C.B.: Specification and design of (parallel) programs. In: Mason, R.E.A. (Ed.) Proceedings of Information Processing '83. North Holland Publishing Co., Amsterdam (1983) 321–332
6. Kesten, Y., Pnueli, A.: A complete proof system for QPTL. In: Proc. 10th IEEE Symp. on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos, California (1995) 2–12
7. Kleene, S.C.: Mathematical Logic. John Wiley & Sons, Inc., New York (1967)
8. Kono, S.: A combination of clausal and non clausal temporal logic programs. In: Fisher, M., Owens, R. (Eds.) Executable Modal and Temporal Logics. Lecture Notes in Computer Science, Vol. 897. Springer-Verlag, Berlin Heidelberg New York (1995) 40–57

9. Kröger, F.: Temporal Logic of Programs. Springer-Verlag, Berlin Heidelberg New York (1987)
10. Manna, Z.: Verification of sequential programs: temporal axiomatization. In: Broy, M., Schmidt, G. (Eds.), Theoretical Foundations of Programming Methodology. D. Reidel Publishing Co. (1982) 53–102
11. Moszkowski B.: Reasoning about Digital Circuits. PhD thesis, Stanford University, Stanford, California (1983)
12. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. IEEE Computer **18** (1985) 10–19
13. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge, England (1986)
14. Moszkowski, B.: Some very compositional temporal properties. In: Olderog, E.-R. (Ed.) Programming Concepts, Methods and Calculi. IFIP Transactions, Vol. A-56, North-Holland (1994) 307–326
15. Moszkowski, B.: Compositional reasoning about projected and infinite time. In: Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95). IEEE Computer Society Press, Los Alamitos, California (1995) 238–245
16. Moszkowski, B.: Embedding imperative constructs in interval temporal logic. Internal memorandum EE/0895/M1. Dept. of Elec. and Elec. Eng., Univ. of Newcastle, Newcastle upon Type, UK (1995)
17. Moszkowski, B.: Using temporal fixpoints to compositionally reason about liveness. In: BCS-FACS 7th Refinement Workshop, "Electronic Workshops in Computing" series. Springer-Verlag, London (1996)
18. Paech, B.: Gentzen-systems for propositional temporal logics. In: Börger, E. et al. (Eds.) Proceedings of the 2nd Workshop on Computer Science Logic. Lecture Notes in Computer Science Vol. 385. Springer-Verlag, Berlin Heidelberg New York (1988) 240–253
19. Rosner, R., Pnueli, A.: A choppy logic. In: Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society Press, Los Alamitos, California (1986) 306–314

Appendix A Practical Proof System for ITL

In this appendix, we present a very powerful and practical compositional proof system for ITL. Our experience in rigorously developing hundreds of propositional and first-order proofs has helped us refine the axioms and convinced us they are sufficient for a very wide range of purposes. See Moszkowski [14] for more about this. The proof system is divided into a propositional part and a first-order part. Our discussion looks at each in turn.

Propositional Axioms and Inference Rules. The propositional axioms and inference rules mainly deal with *chop*, and *skip* and operators derived from them. Only one axiom is needed for *chop-star*. The proof system gives nearly equal treatment to initial and terminal subintervals. This is exceedingly important for the kinds of proofs we do. In addition, this makes the proof system easier to understand since much of it consists simply of duals in this sense. In contrast,

most temporal logics cannot handle initial subintervals and even other proof systems for ITL largely neglect them.

Rosner and Pnueli [19] and Paech [18] give propositional proof systems for ITL with infinite intervals and prove completeness. However, neither system has ever been used much. More recently, Kesten and Pnueli [6] were able to prove the completeness of a very nice proof system for *Quantified Propositional Temporal Logic* (QPTL) using Büchi Automata. Perhaps a similar technique can be applied to propositional ITL with infinite time since it has the same expressiveness as QPTL and can even be translated into it as shown by Halpern and Moszkowski in [11, pp. 23–24]. Our proof system presented here contains some of the propositional axioms suggested by Rosner and Pnueli but also includes our own axioms and inference rule for the operators $\boxdot$, *halt* , and *chop-star*. These assist in deducing propositional and first-order theorems and in deriving rules for importing, exporting and other important aspects of composition.

$$
\begin{array}{ll}
\textbf{Prop} & \vdash \text{ Substitutions of tautologies} \\
\textbf{P2} & \vdash (S;T);U \equiv S;(T;U) \\
\textbf{P3} & \vdash (S \vee S');T \supset (S;T) \vee (S';T) \\
\textbf{P4} & \vdash S;(T \vee T') \supset (S;T) \vee (S;T') \\
\textbf{P5} & \vdash \mathit{empty};S \equiv S \\
\textbf{P6} & \vdash S;\mathit{empty} \equiv S \\
\textbf{P7} & \vdash w \supset \boxdot w \\
\textbf{P8} & \vdash \boxdot(S \supset S') \wedge \Box(T \supset T') \supset (S;T) \supset (S';T') \\
\textbf{P9} & \vdash \bigcirc S \supset \neg \bigcirc \neg S \\
\textbf{P10} & \vdash \diamondsuit\big((\bigcirc \mathit{halt}\ w) \wedge S\big) \supset \boxdot\big((\bigcirc \mathit{halt}\ w) \supset S\big) \\
\textbf{P11} & \vdash S \wedge \Box(S \supset \circledcirc S) \supset \Box S \\
\textbf{P12} & \vdash S^* \equiv \mathit{empty} \vee (S \wedge \mathit{more});S^* \\
\\
\textbf{MP} & \vdash S \supset T, \quad \vdash S \quad \Rightarrow \quad \vdash T \\
\Box\textbf{Gen} & \vdash S \quad \Rightarrow \quad \vdash \Box S \\
\boxdot\textbf{Gen} & \vdash S \quad \Rightarrow \quad \vdash \boxdot S
\end{array}
$$

We now give a sample theorem and its proof:

$$\vdash \quad \boxdot(S \supset T) \quad \supset \quad \diamondsuit S \supset \diamondsuit T \ .$$

Proof:

1	$\vdash$	$\mathit{true} \supset \mathit{true}$	**Prop**
2	$\vdash$	$\Box(\mathit{true} \supset \mathit{true})$	1,$\Box$**Gen**
3	$\vdash$	$\boxdot(S \supset T) \wedge \Box(\mathit{true} \supset \mathit{true})$	**P8**
	$\vdash$	$\supset (S;\mathit{true}) \supset (T;\mathit{true})$	
4	$\vdash$	$\boxdot(S \supset T) \supset (S;\mathit{true}) \supset (T;\mathit{true})$	2,3,**Prop**
5	$\vdash$	$\boxdot(S \supset T) \supset \diamondsuit S \supset \diamondsuit T$	4,def. of $\diamondsuit$

Theorem A. *The propositional proof system is complete for quantifier-free formulas containing only boolean-valued static and state variables.*

Outline of proof: For a given formula, we construct a finite tableau consisting of a number of states. Each state is represented as a disjunction whose disjuncts are themselves conjunctions of primitive propositions, *next* formulas and their negations. Now suppose S is a valid formula. Construct a tableau for its negation $\neg S$. Call a state in a tableau *final* if it is satisfiable by some empty interval. No state reachable from the initial state in our tableau for $\neg S$ is final, since otherwise we can use the path to construct a model for $\neg S$. Therefore the tableau reflects that $\neg S$ is not true in any finite intervals. We convert this to a proof-by-contradiction for S. This technique also applies to a version of Rosner and Pnueli's proof system restricted to finite intervals.

First-Order Axioms and Inference Rules. Below are axioms and inference rules for reasoning about first-order concepts. They are to be used together with the propositional ones already introduced. See Manna [10] and Kröger [9] for proof systems for *chop*-free first-order temporal logic. We let v and v' refer to both static and state variables.

F1 $\vdash$ All substitution instances of valid nonmodal formulas of conventional first-order logic with arithmetic.

F2 $\vdash \forall v\colon S \supset S_v^e$,
where the expression e is sort-compatible with v and v is free for e in S. If e contains any temporal operators, then v must be a state variable not occurring freely in S within the left side of a *chop* formula or within a *chop-star* formula.

F3 $\vdash \forall v\colon (S \supset T) \supset (S \supset \forall v\colon T)$,
where v doesn't occur freely in S.

F4 $\vdash (\imath v\colon S) = (\imath v'\colon S_v^{v'})$,
where v and v' are static variables of one sort and v is free for v' in S.

F5 $\vdash \forall v\colon (S \equiv T) \supset (\imath v\colon S) = (\imath v\colon T)$,
where v is static.

F6 $\vdash (\exists v\colon S) \wedge (\imath v\colon S) = v \supset S$,
where v is a static variable.

F7 $\vdash w \supset \Box w$,
where w only contains static variables.

F8 $\vdash \exists v\colon (S;T) \supset (\exists v\colon S);T$,
where v doesn't occur freely in T.

F9 $\vdash \exists v\colon (S;T) \supset S;(\exists v\colon T)$,
where v doesn't occur freely in S.

F10 $\vdash (\exists v\colon S);\bigcirc(\exists v\colon T) \supset \exists v\colon (S;\bigcirc T)$,
where v is a state variable.

$\forall$Gen $\vdash S \Rightarrow \vdash \forall v\colon S$,
for any variable v.

Induct $\vdash S_n^0, \vdash S \supset S_n^{n+1} \Rightarrow \vdash S$,
for any static variable n whose sort is the natural numbers.

The axiom **F1** permits using properties of conventional first-order logic with arithmetic without proof. Most of the other axioms and the two inference rules at the end are adaptations of conventional nonmodal equivalents for quantifiers and definite descriptions. Only four axioms actually contain temporal operators. Axiom **F7** deals with state formulas containing only static variables. The two axioms **F8** and **F9** show how to move an existential quantifier out of the scope of *chop*. The remaining temporal axiom **F10** shows how to combine two state variables in nearly adjacent subintervals into one state variable for the entire interval. We extensively use it and lemmas derived from it for constructing auxiliary variables. Dutertre [1] gives a complete first-order ITL proof system but unfortunately with a nonstandard semantics of intervals. In addition, it has not be developed with compositional proofs in mind.

A.1 Axioms for Infinite Time

The proof system for ITL with infinite time contains all the axioms and basic inference rules of the basic proof system. We also include the following two propositional axioms:

$$\textbf{P13} \vdash \quad (S \wedge \mathit{inf}\,); T \quad \equiv \quad S \wedge \mathit{inf} \quad ,$$
$$\textbf{P14} \vdash \quad S \wedge \Box\big(S \supset (T \wedge \mathit{more}); S\big) \quad \supset \quad T^* \quad .$$

The first-order axiom now given is sometimes needed for constructing auxiliary variables with *chop-star*:

$$\textbf{F11} \vdash \quad \big(\forall v\colon \exists v'\colon (v = v' \wedge S)\big)^* \quad \supset \quad \forall v\colon \exists v'\colon (v = v' \wedge S^*) \quad ,$$

where v and v' are state variables and v does not occur freely S.

It may be that a complete axiom system for even propositional ITL with infinite intervals can only be achieved by means of a nonconventional inference rule. This is not central to our approach.

Decomposing Real-Time Specifications *

Ernst-Rüdiger Olderog and Henning Dierks

Fachbereich Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg
Germany
E-mail: {olderog,dierks}@informatik.uni-oldenburg.de

Abstract. In this paper we show that every real-time system specified in a certain subset of Duration Calculus [24] can be decomposed into an untimed system communicating with suitable timers. Both asynchronous and synchronous communication are considered.

1 Introduction

Real-time systems are reactive systems where reactions to certain inputs have to occur within given time intervals [8, 16, 14, 11]. These systems usually consist of some physical *process* for which a suitable *controller* has to be constructed such that the controlled process exhibits the desired time dependent behaviour. The interaction between process and controller proceeds via *sensors* and *actuators* as shown in Fig. 1.

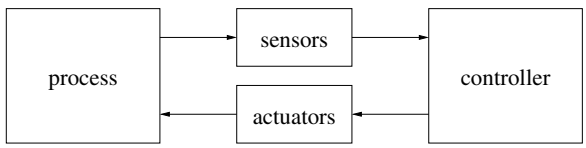


Fig. 1. Real-time system

When constructing the controller the reaction times of all components of this system have to be taken into account.

Since real-time systems are more difficult to design and verify than untimed reactive systems, methods for separating time-critical aspects from untimed causal behaviour are desirable. Possible approaches are *abstraction* and *decomposition*. Abstraction is used in the automatic verification (model checking) of real-time systems specified by timed automata. For example, R. Alur and D. Dill [1] observed that in order to decide whether the language $\mathcal{L}(\mathcal{A})$ of timed traces

* This work was partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 465-489, 1998.
Springer-Verlag Berlin Heidelberg 1998

of a timed automaton $\mathcal{A}$ is empty it suffices to check whether the corresponding untimed language $Untime(\mathcal{L}(\mathcal{A}))$ consisting of all communication traces with the time stamps removed is empty. Their main result is that $Untime(\mathcal{L}(\mathcal{A}))$ is an ω-regular language that can be recognised by a suitable Büchi automaton, the *region automaton*. This region automaton thus represents an abstract finite state view of the infinitely many timed configurations of the original timed automaton $\mathcal{A}$ tailored to the particular verification problem under consideration, here the emptiness problem of $\mathcal{L}(\mathcal{A})$.

Decomposition should preserve the overall real-time properties of the system, but restructure the system such that the time-critical aspects are localised in some components. The aim of this paper is to show how the specification of a real-time controller can be correctly decomposed into an *untimed controller* communicating with *timers* (see Fig. 2). Such a decomposition can provide the basis

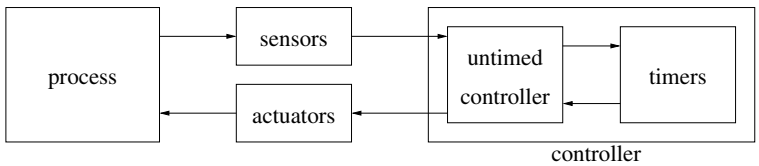


Fig. 2. Controller decomposition

for implementing and simulating real-time controllers and for a proof methodology that separates untimed and timed properties.

Our paper is motivated by two sources. Firstly, in previous work in the project ProCoS (Provably Correct Systems) [10] an approach to the transformational design of real-time systems was developed [23, 22]. In ProCoS the Durations Calculus [24] was taken as the basis for specifiying real-time systems. In [23, 22] it is shown how real-time systems specified in a subset of Duration Calculus [24], the so-called *implementables* due to A.P. Ravn [21], can be gradually transformed into timed occam programs. In the ProCoS example of a gas burner controller the final program consisted of an untimed occam program communicating synchronously with two separate timer components. While this decomposition was possible in the particluar gas burner example, it was unclear how general this construction was. In this paper we show that this decomposition works in general.

Secondly, in a similar style S. Dick and J. Peleska is pursuing an approach where a given Timed CSP process is decomposed into an untimed CSP process communicating synchronously with separate timer processes:

$$Timed_CSP_process = untimed_CSP_process \;||\; timers$$

where $||$ denotes the parallel composition operator of CSP [15, 3]. The advantage of this structural transformation is that real-time interpreters can be based

directly on this decomposition result. This facilitates rapid prototyping and further analysis of the original Timed CSP process. A difficulty lies in *proving* the above decomposition result. This is partly due to the semantics of Timed CSP which assumes a fixed minimal reaction time δ for all communications [2]. As a consequence some simple Timed CSP processes like

$$a \overset{2\cdot\delta}{\Leftrightarrow} b \Leftrightarrow SKIP$$

cannot be decomposed because it would take at least three communications to interact with a timer: one to set the timer, one to let the timer progress, and a third one to communicate the elapsing of the time back to time main process. These communications would need a time of $3 \cdot \delta$ which exceeds the time of $2 \cdot \delta$ allowed between a and b.

In this paper we avoid such difficulties by working with an unspecified non-zero reaction time described by some parameter, say ε. Then our decomposition techniques will *generate conditions* on the reaction times needed for the components and the timers such that the overall real-time requirements are still guaranteed. Formally, this is done by working in the setting of a *continuous time* domain where changes of the time scale can be easily accommodated.

At first sight it seems that the decomposition of real-time systems into untimed parts and timers is already solved by the model of timed automata. As phrased in [17], timed automata can be viewed as satisfying the equation

$$\textit{timed automata} = \textit{finite state machine} + \textit{finite set of clocks}.$$

However, the main difference between timed automata and the present approach is that in the timed automata model the clock operations are indivisibly coupled with the transitions whereas here we present a clear separation of untimed system and timers with explicit communications between them.

In this paper the decomposition result will be established for real-time specifications written as Duration Calculus implementables [21]. The decomposition algorithm reuses a technique of [6] for synthesising so-called PLC-Automata from a given set of Duration Calculus implementables. PLC-Automata are an abstract specification of polling controllers that can be easily implemented on Programmable Logic Controllers (PLCs) [5]. Whereas in [6] the timing conditions are an indivisible part of the PLC-Automata, the novelty of this paper is that the timers are kept as separate components that communicate with the untimed controller. We hope that this increases the conceptual clarity of the approach.

2 Real-Time Specifications

Our basic assumption is that a real-time system can be described by a set of time dependent *observables obs* which are functions

$$obs : Time \to D_{obs}$$

where $Time$ is a continuous *time domain*, here the nonnegative real numbers $\mathbb{R}_{\geq 0}$, and D_{obs} is the type or data domain of obs. For example, a gas valve might be described using a Boolean valued observable

$$gas : Time \to \{0, 1\}$$

indicating whether gas is present or not [20], a railway track by an observable

$$track : Time \to \{empty, appr, cross\}$$

where $appr$ means a train is approaching and $cross$ means that it is crossing the gate [19], and the current communication trace of a reactive system by an observable

$$tr : Time \to Comm^*$$

where $Comm^*$ denotes the set of all finite sequences over a set $Comm$ of possible communications [22]. Thus depending on the choice of observables we can describe a real-time system at various levels of abstraction.

To describe properties of observables we use *predicates* of a suitable logic. In such a predicative approach the semantics of different syntactic descriptions of a real-time system will be given in terms of predicates in the same logic. The advantage is that then *correctness* can be expressed as logic implication between predicates. For any two syntactic descriptions $term_1$ and $term_2$ we write

$$term_1 \mathrel{\equiv\!\!>} term_2$$

if the semantic predicate associated with a $term_1$ logically implies the semantic predicate associated with $term_2$. Conceptually, this means that $term_1$ *satisfies* or *refines* all the properties of $term_2$, i.e. is *correct* w.r.t. $term_2$. For example, if $term_2$ is a specification $spec$ and $term_1$ is a program $prog$ then

$$prog \mathrel{\equiv\!\!>} spec$$

expresses that $prog$ correctly implements $spec$.

In general, the picture gets more complicated if the predicates associated with $term_1$ and $term_2$ involve different observables, say $term_1$ involves more concrete observables c and $term_2$ more abstract ones a. Then we need a *linking invariant* that relates the values of a and c. Such an invariant is known from data refinement [7]; it can also be expressed as a predicate, say $link_{a,c}$. Then correctness becomes

$$term_1 \wedge link_{a,c} \mathrel{\equiv\!\!>} term_2,$$

i.e. the conjunction of $term_1$ and $link_{a,c}$ has to imply $term_2$.

2.1 Duration Calculus

In this paper we use Duration Calculus (abbreviated DC), a real-time interval temporal logic developed by Zhou Chaochen and others [24, 21, 9], as the predicate language to describe properties of real-time systems. This choice is mostly motivated by our previous experience and acquired fluency in this logic, but also by the convenience with which the interval and continuous time aspects of DC allow us to express and reason about reaction times of components.

Syntax. Formally, the syntax of Duration Calculus distinguishes *terms*, *duration terms* and *duration formulae*. *Terms* τ have a certain type and are built from time dependent observables *obs* like *gas* or *track*, *rigid variables* x representing time independent variables, and are closed under typed operators *op*:

$$\tau \ ::= \ obs \ \mid \ x \ \mid \ op(\overline{\tau})$$

where $\overline{\tau}$ is a vector of terms. Terms of Boolean type are called *state assertions*. We use S for a typical state assertion.

Duration terms θ are of type real but their values depend on a given time interval. The simplest duration term is the symbol ℓ denoting the *length* of the given interval. The name Duration Calculus stems from the fact that for each state assertion S there is a duration term $\int S$ measuring the *duration* of S, i.e. the accumulated time S holds in the given interval. Formally:

$$\theta \ ::= \ \ell \ \mid \ \int S \ \mid \ op_{real}(\overline{\theta})$$

where op_{real} is an real-valued operator and $\overline{\theta}$ a vector of duration terms.

Duration formulae denote truth values depending on a given time interval. They are built from the constants *true* and *false*, relations *rel* applied to duration terms, and are closed under the *chop operator* (denoted by ";"), propositional connectives op_{Boole}, and quantification $Q \in \{\forall, \exists\}$ over rigid variables x. We use F for a typical duration formula:

$$F \ ::= \ rel(\overline{\theta}) \ \mid \ F_1 \,;\ F_2 \ \mid \ op_{Boole}(\overline{F}) \ \mid \ Q\,x.F$$

where $\overline{F}$ is a vector of duration formulae. Besides this basic syntax various abbreviations are used for duration formulae:

$$\begin{array}{lrcl}
\textit{point interval}: & \lceil\rceil & \stackrel{\text{def}}{=} & \ell = 0 \\
\textit{everywhere}: & \lceil S \rceil & \stackrel{\text{def}}{=} & \int S = \ell \ \wedge \ \ell > 0 \\
\textit{somewhere}: & \Diamond F & \stackrel{\text{def}}{=} & \textit{true} \ ; \ F \ ; \ \textit{true} \\
\textit{always}: & \Box F & \stackrel{\text{def}}{=} & \neg \Diamond \neg F
\end{array}$$

Semantics. The semantics of Duration Calculus is based on an *interpretation* $\mathcal{I}$ that assigns a fixed meaning to each observable, rigid variable and operator

symbol of the language. To an observable *obs* the interpretation $\mathcal{I}$ assigns a function

$$obs_{\mathcal{I}} : Time \to D_{obs}.$$

This induces inductively the semantics of terms and hence state assertions. For a state assertions S it is a function

$$S_{\mathcal{I}} : Time \to Bool$$

where *Bool* is identified with the set $\{0, 1\}$.

The semantics of a duration term θ is denoted by $\mathcal{I}(\theta)$ and yields a real value depending on a given time interval $[b, e] \subseteq Time$. In particular, ℓ denotes the length of $[b, e]$ and $\int S$ the duration of the state assertion S in $[b, e]$ as given by the integral. Formally:

$$\begin{aligned} \mathcal{I}(\ell)[b, e] &= e \Leftrightarrow b, \\ \mathcal{I}(\textstyle\int S)[b, e] &= \int_b^e S_{\mathcal{I}}(t)\, dt. \end{aligned}$$

The semantics of a duration formula F denotes a truth value depending on $\mathcal{I}$ and a given time interval $[b, e]$. We write $\mathcal{I}, [b, e] \models F$ if that truth value is *true* for $\mathcal{I}$ and $[b, e]$. The definition is by induction on the structure of F. The cases of relations, propositional connectives and quantification are handled as usual. For example, $\mathcal{I}, [b, e] \models \int S \leq k$ if the duration $\int_b^e S_{\mathcal{I}}(t)\, dt$ is at most k. For $F_1 \ ; \ F_2$ (read as F_1 *chop* F_2) we define $\mathcal{I}, [b, e] \models F_1 \ ; \ F_2$ if the interval $[b, e]$ can be "chopped" into two subintervals $[b, m]$ and $[m, e]$ such that $\mathcal{I}, [b, m] \models F_1$ and $\mathcal{I}, [m, e] \models F_2$.

Since in our application to the design of real-time systems the initial values of observables are important, we especially consider time intervals starting at time 0 and define: a duration formula F *holds* in an interpretation $\mathcal{I}$ if $\mathcal{I}, [0, t] \models F$ for all $t \in Time$. To formalise requirements in DC one states a number of suitable duration formulae and considers all interpretations for which the conjunction of the DC formulae holds in this sense.

2.2 DC Implementables

In the following we consider real-time systems that are modelled as (a collection of) state machines with time dependent input *in*, state *st* and output *out* as illustrated in Fig. 3. Formally, such a system can be described using three

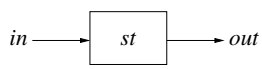


Fig. 3. State-based real-time system

observables

$$in : Time \to Inputs,$$
$$st : Time \to States,$$
$$out : Time \to Outputs$$

where *Inputs, States* and *Outputs* are finite sets. We use the following meta variables: $a, b, ... \in Inputs$ and $A, B, ... \subseteq Inputs$ and $p, q, ... \in States$ and $P, Q, ... \subseteq States$. We stipulate that the output depends only on the state by some function

$$\lambda : States \to Outputs.$$

If the sets *Inputs, States, Outputs* are Cartesian products, we use subscripts to denote the component functions. E.g. for $Outputs = Outputs_1 \times ... \times Outputs_n$ we use $out_i : Time \to Outputs_i$ to denote the i-th component function.

We shall specify the behaviour of this type of real-time system using a subset of Duration Calculus called DC *implementables* and due to A.P. Ravn [21]. DC implementables make use of the following idioms where $s, t \in Time$:

- *followed-by*: $F \Leftrightarrow\!\!\!\!\rightarrow \lceil S \rceil \stackrel{\text{def}}{=} \Box \neg (F \ ; \ \lceil \neg S \rceil)$
- *timed up-to*: $F \stackrel{<s}{\Leftrightarrow\!\!\!\!\rightarrow} \lceil S \rceil \stackrel{\text{def}}{=} (F \wedge \ell \leq t) \Leftrightarrow\!\!\!\!\rightarrow \lceil S \rceil$
- *timed leads-to*: $F \stackrel{t}{\Leftrightarrow\!\!\!\!\rightarrow} \lceil S \rceil \stackrel{\text{def}}{=} (F \wedge \ell = s) \Leftrightarrow\!\!\!\!\rightarrow \lceil S \rceil$

Intuitively, $F \Leftrightarrow\!\!\!\!\rightarrow \lceil S \rceil$ expresses that whenever a pattern given by a formula F is observed, it will be "followed by" an interval where S holds. In the "up-to" form the pattern is bounded by a length "up to" s, and in the "leads-to" form this pattern is required to have a length s. Note that the "leads-to" does not simply say that whenever F holds then t time units later $\lceil S \rceil$ holds, but it rather requires a *stability* of F for t time units before we can be certain that $\lceil S \rceil$ holds. It is this kind of stability requirement that ultimately enables a hardware with polling sensors to implement the "leads-to".

Implementables are certain formats of formulae about the observables *in* and *st* of a real-time system. In these formulae we use the following abbreviations:

a abbreviates $in = a$
A abbreviates $in \in A$
q abbreviates $st = q$
$\neg q$ abbreviates $st \neq q$
P abbreviates $st \in P$

The DC implementables are then of the form:

- Initialisation: $\lceil \rceil \vee \lceil q_0 \rceil ; \ true$
  says that the system must start in a state where q_0 holds. More precisely, each observation interval starting at time 0 is either a point interval or it has an initial (non-point) subinterval where q_0 holds everywhere.

- Sequencing: $\lceil q \rceil \Leftrightarrow\!\!\rightarrow \lceil q \vee P \rceil$
  says that being in state q the system can either remain in this state or at most evolve into a state in the set P.
- Unbounded Stability: $\lceil \neg q \rceil;\ \lceil q \wedge A \rceil \Leftrightarrow\!\!\rightarrow \lceil q \vee Q \rceil$
  says that if the system enters state q while A holds, it is guaranteed to stay in q or to evolve to one of the states in Q.
- Bounded Stability: $\lceil \neg q \rceil;\ \lceil q \wedge B \rceil \overset{<s}{\Leftrightarrow\!\!\rightarrow} \lceil q \vee R \rceil$
  says that if the system enters state q while B holds, it is guaranteed for t time units to stay in q or to evolve to one of the states in R.
- Timed Sequencing: $\lceil q \wedge C \rceil \overset{t}{\Leftrightarrow\!\!\rightarrow} \lceil q \vee T \rceil$
  says that if the system is for t time units in state q with C being fulfilled, it is guaranteed to stay in q or to evolve to one of the states in T.
- Progress: $\lceil q \wedge D \rceil \overset{t}{\Leftrightarrow\!\!\rightarrow} \lceil \neg q \rceil$
  is the only form of implementable requiring that the present state q must be left: being in q while D holds the system must leave q within t time units.

The direct dependence of output from the state can be specified by the DC formula $\Box\lceil out = \lambda(st) \rceil$, i.e. for every subinterval out depends on st as described by the function λ.

2.3 Parallel Composition with Asynchronous Communication

We wish to model parallel composition of state machines such that part of the outputs of one machine serve as inputs of the other and vice versa. We stipulate *asynchronous communication*, i.e. output of one machine can be produced at any time whereas reading this output as input by the other machine and reacting to it occurs somewhat later depending on the internal speed of that machine.

Formally, parallel composition with asynchronous communication of two state machines $\mathcal{M}1$ and $\mathcal{M}2$ specified in terms of observables $in1, st1, out1$ and $in2$, $st2$, $out2$ is denoted by $\mathcal{M}1 \,||_{asyn}\, \mathcal{M}2$. Semantically, the binary operator $||_{asyn}$ is modelled by the *conjunction* of the DC implementables describing $\mathcal{M}1$ and $\mathcal{M}2$ and a DC formula *link* describing the communication links between $\mathcal{M}1$ and $\mathcal{M}2$. One extreme is a disjoint parallel composition without any communication between the two state machines. This is specified by $link \equiv true$. The other extreme is a closed system where the output of $\mathcal{M}1$ is the input of $\mathcal{M}2$ and, vice versa, the output of $\mathcal{M}2$ is the input of $\mathcal{M}1$. This is specified by

$$link \equiv \Box(\lceil in2 = out1 \rceil \wedge \lceil in1 = out2 \rceil)$$

provided that the value domains agree, i.e. $Inputs2 = Outputs1$ and $Inputs1 = Outputs2$ hold. When these value domains are Cartesian products, we can specify the communication links more selectively. For example,

$$link \equiv \Box\lceil in2_j = out1_k \rceil$$

specifies that the k-th output line of $\mathcal{M}1$ is taken as the j-th input line of $\mathcal{M}2$.

Such equations between outputs and inputs of different machines model communication media without delays. Note that in DC it is also possible to specify more elaborate media with delays. However, in this paper we assume that the communication media are fast compared with the speed of the component state machines and that their delays can therefore be ignored.

3 Timers

We specify now a generic *alarm clock timer* for t seconds with a required reaction time of ε seconds where $t, \varepsilon \in Time \Leftrightarrow \{0\}$ are given time parameters. To this end, we specialise the three observables of a time dependent state machine (see also Fig. 4):

$$in : Time \to \{set, reset\}$$
$$st : Time \to \{off, on, alarm\}$$
$$out = st$$

Once an input signal *set* is received, the timer should proceed (after a reaction

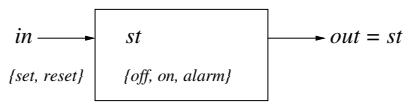


Fig. 4. Timer

time of at most ε) from its initial state *off* into the state *on*. After t seconds have elapsed the timer should proceed (after a reaction time of at most ε) to the *alarm* state. In this state the timer stays until it receives an input signal *reset*. Then it returns (after a reaction time of at most ε) to its initial state *off*. This typical behaviour is displayed by the timing diagram of Fig. 5.

Additionally, we require that the timer can be reset at any moment. Thus the untimed transition diagram is as shown in Fig. 6. Using DC implementables, this diagram can be specified as follows:

- Initialisation:

$$\lceil\rceil \vee \lceil off \rceil ;\ true$$

- Sequencing:

$$\lceil off \rceil \Leftrightarrow\!\!\!\to \lceil off \vee on \rceil$$
$$\lceil alarm \rceil \Leftrightarrow\!\!\!\to \lceil alarm \vee off \rceil$$

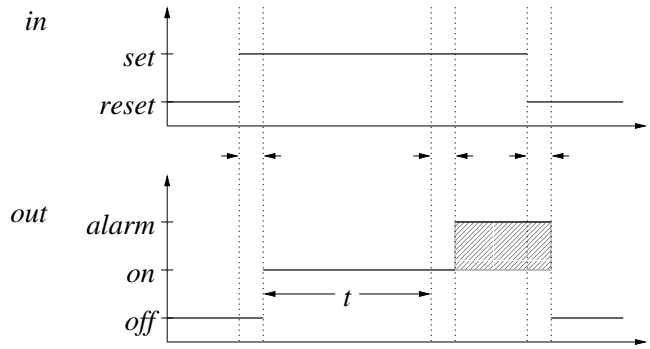


Fig. 5. Timer requirement

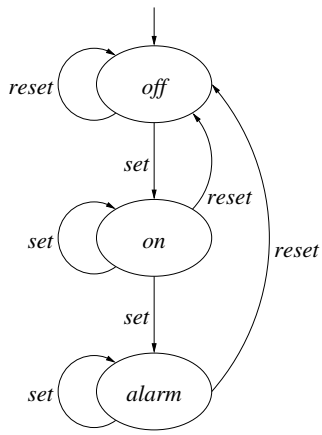


Fig. 6. Timer transition diagram

– Unbounded Stability:

$$\lceil \neg off \rceil ; \lceil off \wedge reset \rceil \Leftrightarrow\!\!\!\!\rightarrow \lceil off \rceil$$
$$\lceil \neg on \rceil ; \lceil on \wedge set \rceil \Leftrightarrow\!\!\!\!\rightarrow \lceil on \vee alarm \rceil$$
$$\lceil \neg on \rceil ; \lceil on \wedge reset \rceil \Leftrightarrow\!\!\!\!\rightarrow \lceil on \vee off \rceil$$
$$\lceil \neg alarm \rceil ; \lceil alarm \wedge set \rceil \Leftrightarrow\!\!\!\!\rightarrow \lceil alarm \rceil$$

The timing conditions are not visible in the transition diagram in Fig. 6 but they can be conveniently specified by the following DC implementables:

– Bounded Stability:

$$\lceil \neg on \rceil ; \lceil on \wedge set \rceil \overset{\leq t}{\Leftrightarrow\!\!\!\!\rightarrow} \lceil on \rceil$$

– Timed Sequencing:

$$\lceil on \wedge reset \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil on \vee off \rceil$$
$$\lceil on \wedge set \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil on \vee alarm \rceil$$

– Progress:

$$\lceil off \wedge set \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil \neg off \rceil$$
$$\lceil on \wedge set \rceil \overset{t+\varepsilon}{\Leftrightarrow} \lceil \neg on \rceil$$
$$\lceil on \wedge reset \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil \neg on \rceil$$
$$\lceil alarm \wedge reset \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil \neg alarm \rceil$$

Except for the *set* transition in the *on* state every transition is to be taken as fast as possible, i.e. within a reaction time of at most ε. The *set* transition in state *on* is delayed by t seconds and thereafter can take place within further ε seconds. In the subsequent sections we will reuse instances of this generic timer as components of real-time controllers.

4 Decomposition: Examples

In this paper we are interested in decomposing timed state machines specified by a set of DC implementables into an *untimed* state machine communicating asynchronously with suitable *timers*. Intuitively, for an untimed state machine there should be no time restrictions on the transition behaviour. Thus it seems that in terms of DC implementables this behaviour should be describable purely by initialisation, sequencing and unbounded stability constraints.

However, then we also allow an idling behaviour where after some time there is no reaction to inputs any more. This does not correspond to the intended behaviour of an ordinary state machine in the presence of inputs. To avoid this we model untimed machines in a timed setting as machines where progress is guaranteed. This is achieved by requiring a uniform progress bound ε for *all* transitions. Thus an "untimed" machine is actually modelled here as an *eager* machine that tries to perform its transitions as fast as possible, with an upper time bound of ε. The role of the timers will then be to slow down the eager machines at appropriate moments.

4.1 Gas burner

At first we consider the gas burner example due to [20, 10]. The gas burner is controlled through a thermostat, and can itself control a gas valve and monitor the flame. It can be modelled by the Boolean observables

$$hr,\ gas,\ fl : Time \to \{0, 1\}$$

which express the states of the thermostat (with *hr* standing for *heat request*), the *gas valve* and the *flame*. Besides several functional requirements there is

the safety requirement that gas must not leak for too long. Gas leakage can be modelled by the state assertion

$$leak \stackrel{\text{def}}{=} gas \wedge \neg fl.$$

Then a safety requirement is that for every time interval $[b, e]$ of length $e \Leftrightarrow b \leq 30$ the duration of *leak* within that interval is at most 4. In DC this can be conveniently stated using the integral:

$$Safe \stackrel{\text{def}}{=} \Box\,(\ell \leq 30 \Rightarrow \int leak \leq 4).$$

In [10] it is shown that this and other requirements are satisfied by a state-based gas burner controller *GBC* as in Fig. 7. Formally, the inputs, states and

Fig. 7. Gas burner control

outputs of *GBC* can be represented by the following observables:

$$in : Time \to \{hr, \neg hr\} \times \{fl, \neg fl\}$$
$$st : Time \to \{idle, purge, ignite, burn\}$$
$$out : Time \to \{gas, \neg gas\}$$

Note that input is modelled here by the Cartesian product of the Boolean observables *hr* and *fl*. The transition diagram of *GBC* and how the output *gas* and $\neg gas$ depends on the state is illustrated in Fig. 8. Note that the transitions between the states *purge* and *ignite* and between *ignite* and *burn* occur under every possible input value. All other combinations of input values that are not shown in the diagram yield *idling transitions*. For example, *GBC* idles in the *burn* state if the input is $hr \wedge fl$. As with the timer in Section 3 this transition diagram can be expressed using initialisation, sequencing and unbounded stability constraints. The timing conditions are stated separately.

– Bounded Stability:

$$\lceil \neg purge \rceil;\ \lceil purge \rceil \stackrel{<30}{\Leftrightarrow\!\!\rightarrow} \lceil purge \rceil$$
$$\lceil \neg ignite \rceil;\ \lceil ignite \rceil \stackrel{<1}{\Leftrightarrow\!\!\rightarrow} \lceil ignite \rceil$$

– Progress:

$$\lceil purge \rceil \stackrel{30+\varepsilon}{\Leftrightarrow\!\!\rightarrow} \lceil \neg purge \rceil$$
$$\lceil ignite \rceil \stackrel{1+\varepsilon}{\Leftrightarrow\!\!\rightarrow} \lceil \neg ignite \rceil$$

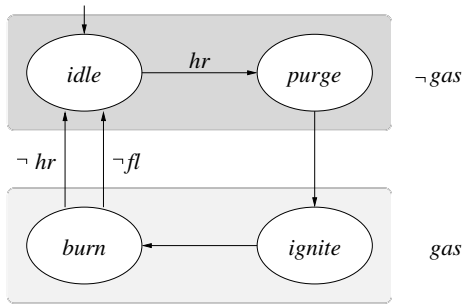


Fig. 8. Gas burner transition diagram

For all other transitions we require a reaction time of at most ε, for example

$$\lceil idle \wedge hr \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil \neg idle \rceil.$$

We now wish to make these timing conditions more explicit by decomposing the gas burner controller *GBC* into an "untimed" controller communicating with suitable timers. In this case the resulting decomposition looks as shown in Fig. 9. The arrows between the component boxes indicate the communication links of the parallel composition.

The original state *purge* of the gas burner controller in Fig. 8 has been split into two substates $p1$ and $p2$ and the original state *idle* has been split into $i1$ and $i2$. Formally, this state or *phase splitting* can be justified using a transformation rule of [23, 22]. On the semantic level a linking invariant $link_{a,c}$ is needed to specify the refinement relationship between the "abstract" gas burner in Fig. 8 and the "concrete" untimed controller in Fig. 9. If we take st_a and st_c as the names for the two state observables, the linking invariant can be expressed by the following DC formula:

$$\begin{aligned} link_{a,c} \overset{\text{def}}{=} \Box\ (\ & \lceil st_a = purge \rceil \Leftrightarrow \lceil st_c = p1 \vee st_c = p2 \rceil \\ & \lceil st_a = ignite \rceil \Leftrightarrow \lceil st_c = i1 \vee st_c = i2 \rceil \\ & \lceil st_a = burn \rceil \Leftrightarrow \lceil st_c = burn \rceil \\ & \lceil st_a = idle \rceil \Leftrightarrow \lceil st_c = idle \rceil\ ) \end{aligned}$$

Thus whenever the original gas burner is in state *purge* the new untimed controller is in state $p1$ or $p2$ and analogously for *ignite* and $i1, i2$. The other states *burn* and *idle* are left unchanged.

It is understood that in the states $idle, p1, p2$ the output is $\neg gas$ and in the states $burn, i1, i2$ it is *gas*. Additionally, new outputs *set30, reset30, set1, reset1* are generated in the new states $p1, p2, i1, i2$ and used as inputs for the timers. In the graphic representation of the untimed gas burner controller the lower parts of the new states $p1, p2, i1, i2$ display these new outputs.

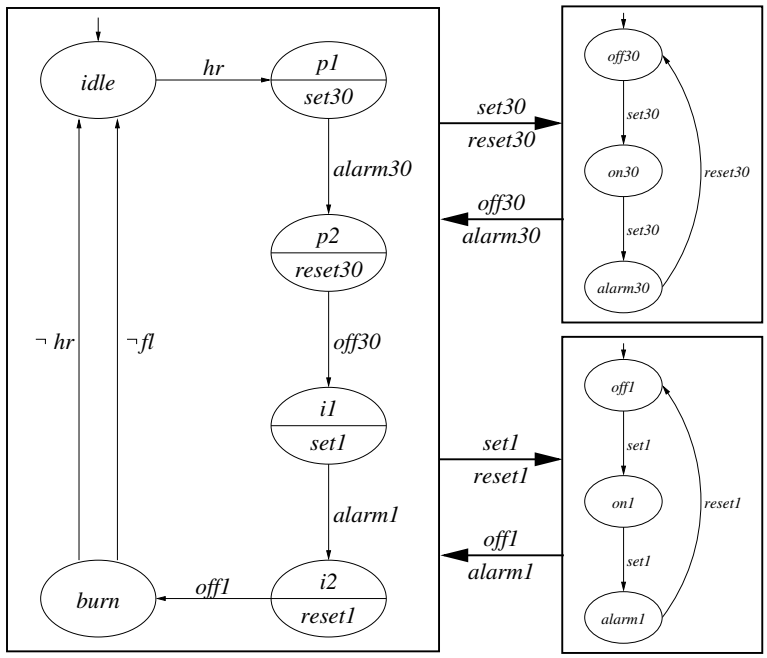


Fig. 9. Gas burner decomposition

The timers in turn directly output their states as described in Section 3. In this particular decomposition only the outputs *off30, alarm30, off1, alarm1* of the timers are needed as inputs of the gas burner controller. For a proper functioning we have to assume that the progress bounds of all transitions the gas burner controller are $\varepsilon/5$. The same is assumed for all transitions of the timer for 30 seconds except for the delayed *set30*-transition leaving state *on30* where the progress bound is $30+\varepsilon/5$ and analogously for the timer for 1 second.

Communication between gas burner controller and timers proceeds according to the following protocol. When the gas burner controller is in state $p1$ it outputs *set30*. This is an input to the timer for 30 seconds waiting in its initial state *off30*. Within a time of $\varepsilon/5$ the timer progresses to its *on30* state. By the specification of the timer, we know that within the time interval $[30, 30+\varepsilon/5]$ progress to the state *alarm30* occurs. This state information is output to the gas burner controller. Within $\varepsilon/5$ seconds the controller progresses to its state $p2$ in which it outputs *reset30* to the timer. The timer reacts within $\varepsilon/5$ seconds to this communication by returning to its initial state *off30*. This state is then output to the gas burner controller as an acknowledgement of the *reset30* communication. Only after receiving this acknowledgement does the controller proceed (in at most $\varepsilon/5$ seconds) to the subsequent state $i1$.

Altogether, we observe the following behaviour: whenever the gas burner controller is in state $p1$ then after at least 30 seconds and at most $30 + \varepsilon$ seconds the state $i1$ is entered. This corresponds to entering the original state *idle* and is thus exactly what is required.

4.2 Filter

This example is motivated by an industrial case study on tramway control [15]. For the safety of a single track segment sensors are needed to detect how many trams are in certain track segments. In particular, at most one tram is allowed to enter the critical single track segment at a time.

A technical problem is that sensors may stutter, i.e. issue more than one output signal when in reality only a single tram passes the sensor. To avoid wrong data in the drive controller for the trams a suitable *filter* is needed for each sensor. We consider here a filter FES reading input values 0, 1 and E (for

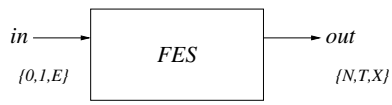


Fig. 10. Filter

error) from a sensor called ES (for *enter single track*) and transforming them into output values N (for *no tram*), T (for *tram*) and X (for *exception*). This is indicated by Fig. 10 and the corresponding observables:

$$in : Time \to \{0, 1, E\}$$
$$st : Time \to \{N, T, X\}$$
$$out = st$$

The desired real-time behaviour is shown in the timing diagram in Fig. 11. When an input 1 from the sensor ES is detected the filter FES should (after a reaction time of at most ε) output T (*tram detected*). In the subsequent 5 seconds the filter should ignore any further stuttering of inputs 0 or 1 from the sensor and stay with output T. We stipulate that after 5 seconds any stuttering of the sensor has ceased so that the filter (after a reaction time of at most ε) returns to output N. Afterwards any further input 1 will be treated as signalling a new tram approaching thus causing output T again.

There is one input though which the filter FES must not ignore. That is the input E indicating an erroneous sensor value. Then the filter should proceed (after a reaction time of at most ε) to (state and) output X. The transition diagram (again without showing the idling transitions) is given in Fig. 12.
The timing conditions formalising the desired behaviour are as follows.

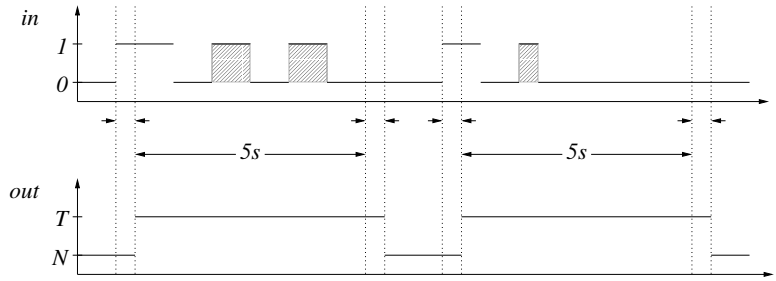


Fig. 11. Filter requirement

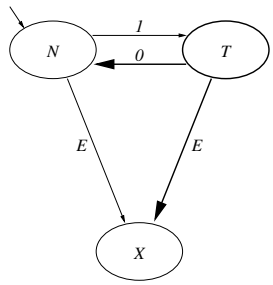


Fig. 12. Filter transition diagram

- Filtering: $\lceil \neg T \rceil;\ \lceil T \wedge (0 \vee 1) \rceil \overset{\leq 5}{\Leftrightarrow} \lceil T \rceil$
  specifies that if a tram has been detected (output T) then this output should stay for the next 5 seconds provided only 0 or 1 occur as inputs.
- Error handling: $\lceil T \wedge E \rceil \overset{\varepsilon}{\Leftrightarrow} \lceil \neg T \rceil$
  specifies that if an error E has been sensed, state and output T should be changed within ε seconds.

Again we wish to make these timing conditions more explicit by decomposing the above filter *FES* into an "untimed" controller communicating with a timer for 5 seconds. In this case the resulting decomposition looks as shown in Fig. 13 where $\varepsilon/5$ is required as the new progress bound for all transitions. Note that this decomposition is more complex than that of the gas burner *GBC* because here the timer may be reset before it elapses and the alarm sounds, viz. whenever input E is detected.

5 Decomposition: Algorithm

So far we have seen two examples for decomposing the specification of a real-time controller into an untimed controller communicating with suitable timers.

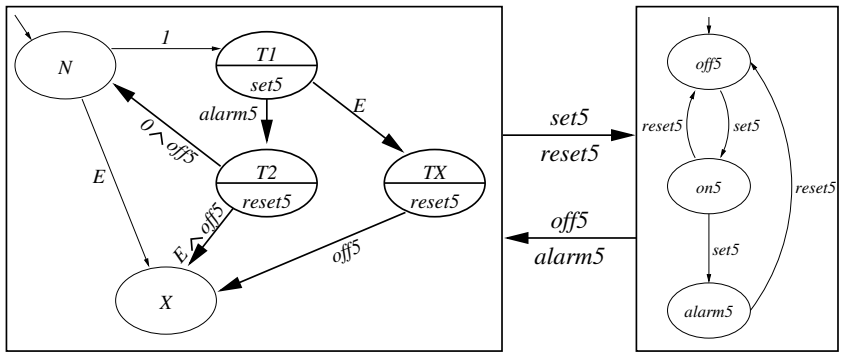


Fig. 13. Filter decomposition

The question is whether this decomposition works in general. In this section we give an affirmative answer for the case that the specification is given as a set *SPEC* of DC implementables. We stipulate that these implementables specify the desired behaviour of the observables *in* and *st* of a real-time system as in subsection 2.2.

Based on the synthesis algorithm of [6] we describe now a decomposition algorithm that realises the following transformation where $||_{asyn}$ is the parallel composition based on asynchronous communication described in subsection 2.3:

Decomposition
$SPEC$ $\Uparrow$ $(untimed_controller \,\|\|_{asyn}\, timers) \wedge link_{a,c}$
provided $SPEC$ is consistent

Note that the decomposition transformation states a refinement result, not a semantic equivalence: the parallel composition of the untimed controller with the timers *refines* the original specification *SPEC*.

The notion of *consistency* deals with the question whether the requirements stated in *SPEC* contradict each other. For example, a progress requirements

$$\lceil p \wedge a \rceil \overset{5}{\Leftrightarrow\!\!\!\rightarrow} \lceil \neg p \rceil$$

stating that under input a the system should *leave* state p within 5 seconds would be inconsistent with a bounded stability

$$\lceil \neg p \rceil ; \lceil p \wedge a \rceil \overset{\leq 6}{\Leftrightarrow\!\!\!\rightarrow} \lceil p \rceil$$

stating that under the same input the system should *stay* in state p for at least 6 seconds.

Idea of the algorithm. Assume now that *SPEC* is a set of DC implementables consisting of an

- Initialisation: $\lceil\rceil \vee \lceil q_0 \rceil;\ true$

and for each $q \in States$ of constraints of the form

- Sequencing: $\lceil q \rceil \Leftrightarrow \lceil q \vee P \rceil$
- Unbounded Stability: $\lceil \neg q \rceil;\ \lceil q \wedge A_i \rceil \Leftrightarrow \lceil q \vee Q_i \rceil\ for\ i \in I$
- Bounded Stability: $\lceil \neg q \rceil;\ \lceil q \wedge B_j \rceil \overset{\leq s'_j}{\Leftrightarrow} \lceil q \vee R_j \rceil\ for\ j \in J$
- Timed Sequencing: $\lceil q \wedge C_k \rceil \overset{t_k}{\Leftrightarrow} \lceil q \vee T_k \rceil\ for\ k \in K$
- Progress: $\lceil q \wedge D_\ell \rceil \overset{t'_\ell}{\Leftrightarrow} \lceil \neg q \rceil\ for\ \ell \in L$

Then the decomposition algorithm proceeds – for each state q – in four steps.

Step 1. Sort the set of stability time bounds $\{s'_j \mid j \in J\} = \{s_1, s_2, ..., s_n(q)\}$ such that

$$0 = s_0 < s_1 < s_2 < ... < s_n(q) < s_{n(q)+1} = \infty.$$

Step 2. Construct – by scanning the sequencing, unbounded stability, bounded stability, timed sequencing and progress constraints – a time dependent transition table for the state q:

δ_q	s_1	...	s_i	...	$s_{n(q)}$	∞
a		...	$\delta_q(a, s_i)$	...		
b		...	$\delta_q(b, s_i)$	...		
...		...		...		

An entry $\delta_q(a, s_i)$ in the table denotes the set of all successor states of q under input a during the time interval $(s_{i-1}, s_i]$ where time is measured from the moment that state q is entered. Note that progress constraints are the only form of constraints that can *remove* the state q from the set $\delta_q(a, s_i)$; all other constraints allow the possibility of staying in q and thus keep $q \in \delta_q(a, s_i)$. If removing q yields $\delta_q(a, s_i) = \varnothing$, an inconsistency of the specification *SPEC* of the form illustrated above has been discovered and the algorithm stops with an appropriate error massage.

Otherwise we continue by calculating constraints for the allowed reaction time ε to inputs of the decomposed system. This is done by comparing the required progress times with the stability times. Consider a progress constraint of the form

$$\lceil q \wedge D_\ell \rceil \overset{t'_\ell}{\Leftrightarrow} \lceil \neg q \rceil.$$

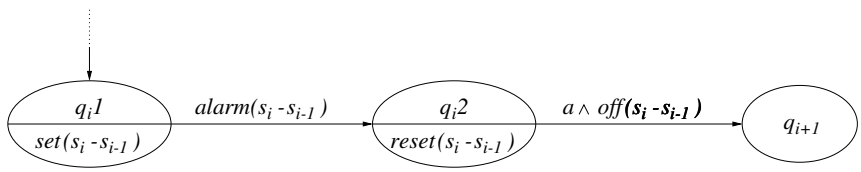
If $t'_\ell \in (s_{i-1}, s_i]$ then we generate the inequality $\varepsilon \leq t'_\ell \Leftrightarrow s_{i-1}$. Choose an ε satisfying all these inequalities as the reaction time.

Step 3. Split q into a cascade of substates $q_1, q_2, ..., q_{n(q)+1}$. As illustrated in subsection 4.1, such a splitting is formalised by a suitable linking invariant. To describe the transition relation from these substates consider some $i \in \{1, ..., n(q)+1\}$. For each input a we distinguish two cases.

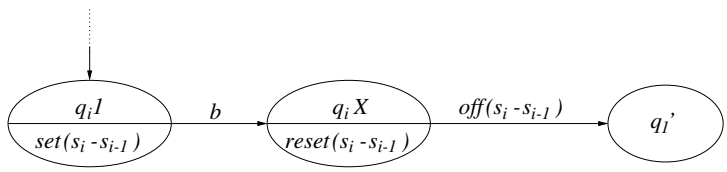
Time progress case: $q \in \delta_q(a, s_i)$
If $i \leq n(q)$, we add an a-transition from q_i to q_{i+1} and require that this transition is delayed (by a corresponding bounded stability constraint) for $s_i \Leftrightarrow s_{i-1}$ seconds. After this stability time has passed, progress is required to occur within ε seconds. Otherwise, i.e. if $i = n(q)+1$, we add an idling a-transition from $q_{n(q)+1}$ to $q_{n(q)+1}$. These transitions model time passage without leaving the original state q.

Exit case: $q \notin \delta_q(a, s_i)$
Take one state $q' \in \delta_q(a, s_i)$ and add an a-transition from q_i to q_1', the first substate of the cascade generated for q'. This transition models the exit from q to q'.

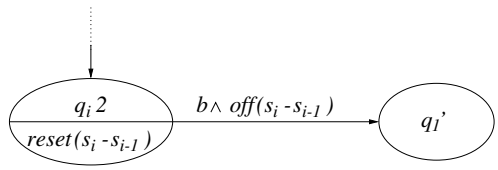
Step 4. Introduce timers and suitable communications with them. More specifically, for each of the substates q_i with $i \in \{1, ..., n(q)+1\}$ we add a new timer for $s_i \Leftrightarrow s_{i-1}$ seconds. Given q_i we perform the following state splitting to achieve the asynchronous communication with this timer. Each time progress a-transition from q_i to q_{i+1} is replaced by



Each exit a-transition from q_i to q_1' is replaced by



and for each $q_i 2$ generated by a replacement of some time progress transition the new transition



is added. Altogether the replacement looks as in Fig. 14. The state splittings

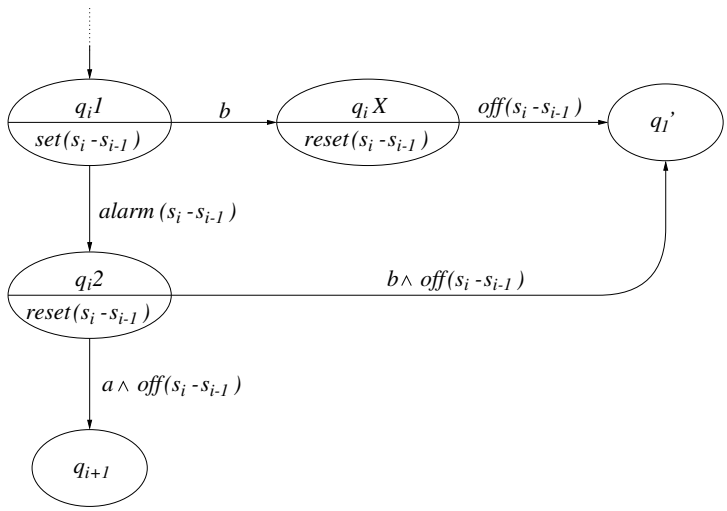


Fig. 14. Overall replacement in the asynchronous setting

shown here and in *Step 3* are formalised by a suitable linking invariant $link_{a,c}$, the one mentioned in the statement of the decomposition rule.

To guarantee the original progress constraints we take $\varepsilon/5$ as the new progress bound for all transitions. Note that the stability constraint introduced in *Step 3* is implemented by the stability of the timer for $s_i \Leftrightarrow s_{i-1}$ seconds after the input $set(s_i \Leftrightarrow s_{i-1})$ and by the corresponding delay of the input of $alarm(s_i \Leftrightarrow s_{i-1})$ for the controller component.

6 Decomposition based on Synchronous Communication

It is interesting to note that the decomposition result can also be adapted to case of *synchronous communication* via directed channels as in CSP [12] or OCCAM [13]. Semantically, timed systems based on synchronous communications can be modelled by introducing two specific observables [22], a time dependent *trace*

$$tr \in Time \to Comm^*$$

where $Comm^*$ denotes the set of all finite sequences over a set $Comm$ of possible communications, usually structured as pairs (ch, m) where ch is an element of a set $Chan$ of communication channels and m is an element of a set of messages, and a time dependent *ready set*

$$R \in Time \to \mathcal{P}(Chan).$$

Intuitively, $ch \in R$ means that the component is willing to communicate along channel ch.

If two components of a system are connected by a directed channel ch and both are ready for communication, one for output and the other for input, the synchronised communication can occur. We do not require it to occur exactly at some point of time but allow that it happens within some interval bounded by a *latency function*

$$lat : Chan(\Delta) \to \mathbb{R}_{>0}.$$

Intuitively, $lat(ch) = t$ means that if the partner components are ready for t time units to communicate along channel ch then a communication will take place. Strictly speaking, this amounts to requiring the existence of a certain *channel scheduler* in the underlying hardware on which the constructed real-time programs are to run.

In the setting of synchronous communication a timer for t seconds needs only two states, *off* and *on* (see Fig. 15). It has three communication channels of message type *signal* for pure synchronisation without value transmission, the input channels *reset* and *set*, and the output channel *alarm*. Following the CSP conventions, we use ? to indicate input and ! to indicate output.

Initially, the timer is in state *off*. On input of a communication along the channel *set*, it proceeds within a time of $lat(set)$ to state *on*. In *on* it stays for at least t seconds. After t seconds it is ready for an output along the channel *alarm*. In both states the timer is ready for an input along the channel *reset*. After that communication the timer returns to its initial state *off*.

How to achieve a decomposition based on synchronous communication ? Let us first look at the gas burner example again (see Fig. 16). When the untimed controller component of the gas burner is in state $p1$ it is ready for output along the channel *set30*. The timer for 30 seconds is already waiting for input along this channel. After a time of at most $lat(set30)$ this communication is actually occurring, thus advancing the controller component to state $p2$ and the timer to

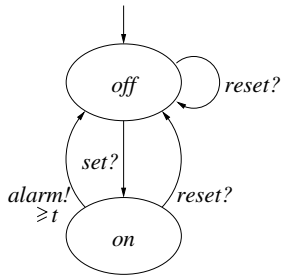


Fig. 15. A timer in a synchronous setting

state *on*. In state $p2$ the controller component is ready for input along channel *alarm30*, but the timer delays its readiness for this communication by 30 seconds. Then after at most $lat(alarm30)$ this communication occurs and advances the controller component to state $i1$ and resets the timer to its initial state *off*. Now an analogous communication behaviour with the timer for 1 second occurs. By choosing the above latencies all as $\varepsilon/2$, the required timing conditions concerning the original *purge* state, here split into $p1$ and $p2$, are met. The stability of 30 seconds is guaranteed by the delay between the communications $set30$ and $alarm30$; the progress of $30 + \varepsilon$ is guaranteed by the choice of the latencies. An analogous argument holds for the original *idle* state, here split into $i1$ and $i2$.

Note that in contrast to the asynchronous communication we need not specify a protocol for sending and acknowledging communications but exploit that each synchronous handshake communication between untimed controller and timer provides simultaneous information for both components.

The general case is more complex as the gas burner example and is dealt with in the following.

Algorithm. We reuse the first three steps of the decomposition algorithm described in Section 5. Only the final step needs to be adapted.

Step 4 – Synchronous communication. Introduce timers and suitable synchronous communications with them. More specifically, for each of the substates q_i with $i \in \{1, ..., n(q) + 1\}$ we add a new timer for $s_i \Leftrightarrow s_{i-1}$ seconds. Given a time progress a-transition from q_i to q_{i+1} and an exit b-transition from q_i to q'_1, we replace these transitions by the part shown in Fig. 17. As latency we take $\varepsilon/3$ for all communication channels with the timers. Also we need $\varepsilon/3$ as the new progress bound for all other transitions. Together this guarantees the required timing conditions.

7 Conclusion

In this paper we have presented an algorithm for decomposing a real-time system specified in a certain subset of Duration Calculus, the so-called *implementables*

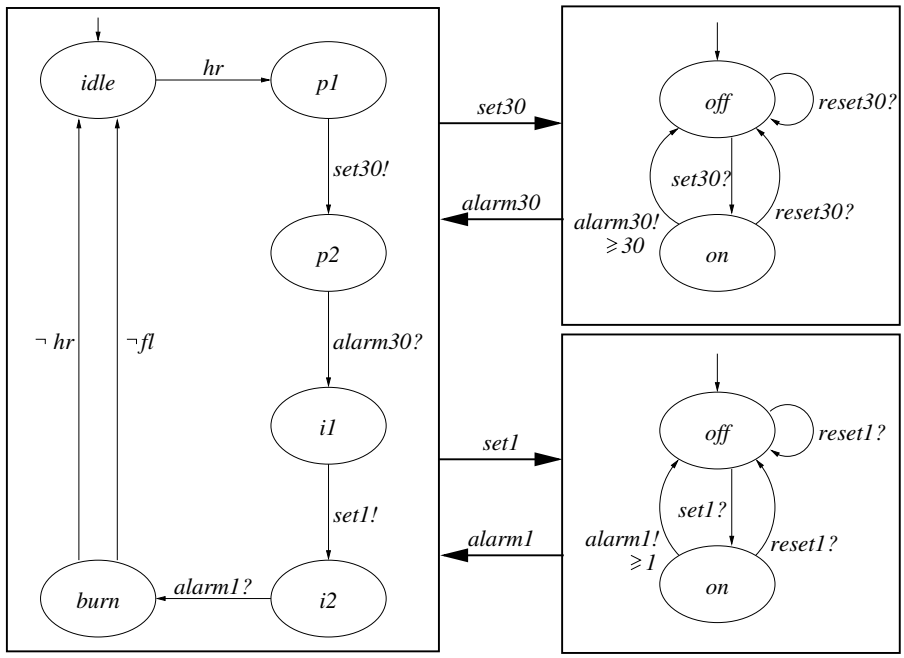


Fig. 16. The gas burner in a synchronous setting

of [21], into an untimed controller communicating in an asynchronous manner with timers. The result was established in the setting of continuous time and non-zero reaction times for the system components.

For simplicity we used here a separate timer instance for each required delay time t. In reality one will use (and reuse) programmable timers where t is transmitted with the initial *set* communication. The decomposition shown in this paper restructures a system into a possibly large untimed system communicating with small timer components. For future work it is desirable to search for more flexible decomposition techniques. We hope to obtain more detailed insights by analysing case studies of real-time systems like the production cell [18,4] and the tramway control [15]. More generally, it would be desirable to have techniques for correctly decomposing real-systems into different views like a process-oriented, a data-oriented and a time-critical view.

References

1. R. Alur, D. Dill. A theory of timed automata. Theoret. Comput. Sci. 126, 1994, 283–235
2. J. Davies. *Specification and Proof in Real-Time CSP.* Cambridge University Press, 1993.

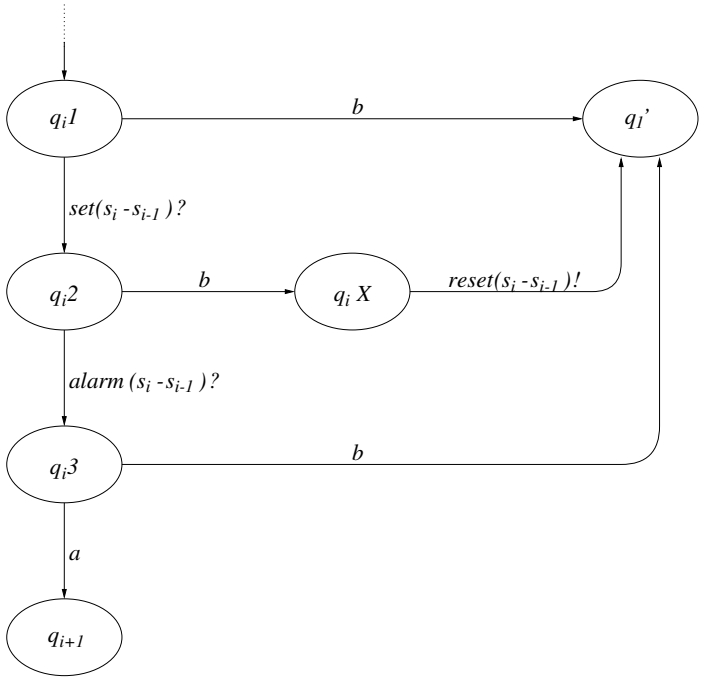


Fig. 17. Overall replacement in the synchronous setting

3. S. Dick, J. Peleska. A structural decomposition theorem for real-time CSP. Tech. Report, Univ. Bremen, in preparation, 1998.
4. H. Dierks. The production cell: A verified real-time system. In: B. Jonsson and J. Parrow (Eds.), *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*, LNCS 1135 (Springer-Verlag, 1996) 208–227.
5. H. Dierks. PLC-automata: a new class of implementable real-time automata. In: M. Bertran and T. Rus (Eds.), *Transformation-Based Reactive Systems Development*. LNCS 1231 (Springer-Verlag, 1997) 111–125.
6. H. Dierks. Synthesising controllers from real-time specifications. In: *Tenth International Symposium on System Synthesis* (IEEE CS Press, September 1997) 126–133.
7. K. Engelhardt, W.-P. de Roever, *Data Refinement* (Book draft, 1997).
8. R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (Eds.). *Hybrid Systems*. LNCS 736 (Springer-Verlag, 1993)
9. M.R. Hansen, Zhou Chaochen, Duration Calculus: Logical Foundations. Formal Aspects of Computing 9 (1997) 283-330.
10. Jifeng He, C.A.R. Hoare, M. Fränzle, M. Müller-Olm, E.-R. Olderog, M. Schenke, M.R. Hansen, A.P. Ravn, and H. Rischel. Provably correct systems. In: [16] 288–335.
11. C. Heitmeyer and D. Mandrioli (Eds.), *Formal Methods for Real-Time Computing*. Trends in Software, Vol.5, (Wiley, 1996).
12. C.A.R. Hoare, *Communicating Sequential Processes* (Prentice Hall, 1985).

13. INMOS Ltd., occam *2 Reference Manual* (Prentice Hall, 1988).
14. M. Joseph (Ed.). *Real-time Systems — Specification, Verification and Analysis* (prentice Hall, 1996).
15. B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, A. Baer. UniForM — Universal Formal Methods Workbench. In: U. Grote and G. Wolf (Eds.), *Statusseminar des BMBF Softwaretechnologie* (BMBF, Berlin, 1996) 357–377.
16. H. Langmaack, W.-P. de Roever, J. Vytopil (Eds.). *Formal Techniques in Real-Time and Fault-Tolerant Systems.* LNCS 863 (Springer-Verlag, 1994).
17. K.G. Larsen, B. Steffen, C. Weise. Countinuous modeling of real time and hybrid systems: from concepts to tools. To appear in Software Tools for Technology Transfer (STTT, Springer-Verlag).
18. C. Lewerentz, T. Lindner. *Formal Development of Reactive Systems: Case Study Production Cell.* LNCS 891 (Springer-Verlag, 1995).
19. E.-R. Olderog, A.P. Ravn, J.U. Skakkebæk. Refining system requirements to program specifications. In: [11] 107–134.
20. A.P. Ravn, H. Rischel, K.M. Hansen. Specifying and verifying requirements of real-time systems. IEEE Transactions on Software Engineering, vol. 19,1 (1993) 41–55.
21. A.P. Ravn. *Design of Embedded Real-Time Computing Systems.* Thesis for the Doctor of Technics. Technical Report ID-TR: 1995-170, Technical University of Denmark, 1995.
22. M. Schenke, E.-R. Olderog, Transformational design of real-time systems – part I: from requirements to program specifications. To appear in Acta Inform.
23. M. Schenke. *Development of Correct Real-Time Systems by Refinement.* Habilitation thesis, Univ. Oldenburg, 1997.
24. Zhou Chaochen, C.A.R. Hoare, A.P. Ravn. A calculus of durations. Information Processing Letters 40/5 1991, 269–276.

On the Combination of Synchronous Languages *

Axel Poigné, Leszek Holenderski

GMD-SET, Schloss Birlinghoven, D-53754 Sankt Augustin, Germany

1 Introduction

Synchronous languages [1, 4, 7, 9] address the specification and programming of *reactive* processes, i.e. processes which continuously respond to stimuli at a rate determined by the environment. The *synchrony hypothesis* [1] states that a process is fully responsible for the synchronization with its environment, that is:

- *event synchronization*: the process is always able to react to events of the environment at a rate determined by the environment;
- *response synchronization*: the response synchronizes properly with the environment, i.e., the time elapsed between a stimulus and the response of the process is short enough (relatively to the dynamics of the environment) so that the environment is still receptive to the response.

Furthermore, the behaviour of a process should be *reproducible* with regard to input events, or, in more technical terms, *deterministic*. Both these requirements are prerequisites for the dependable service of a process, for instance as controller in a safety-critical environment such as an automobile, an aircraft, or a power station.

Available synchronous formalisms are quite different in focus and style:

- Data flow languages such as *Lustre* [4] or *Signal* [7] are particularly suited for representing periodic behaviour as is typical for "continuous" computation of sensor/actuator data,
- state-based languages such as *Esterel* [1] are better suited for "spontaneous" control behaviour, e.g. that of a mouse or a track pad.
- Graphical languages such as *Statecharts* [5] or *Argos* [9] are useful for structuring as, e.g., representing "change of mode" of a continuous system.

Existence of several synchronous formalisms is rather an advantage than a drawback: the formalisms are complementary, addressing different aspects of use. For instance, we may distinguish several modes of operation in a control application; a start mode, normal continuous behaviour, exception mode, and a termination mode. LUSTRE is most adeaquate for modelling the normal continous mode, and, maybe, an exception mode if its behaviour is cyclic, ARGOS is well suited to model change of mode, while ESTEREL may be useful for the start and termination modes which typically involve some sequential processing. Real applications

* The work was partially funded by the Esprit LTR Action **SYRF**, "Synchronous Reactive Formalisms" (Esprit Project 22703).

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 490-514, 1998.
Springer-Verlag Berlin Heidelberg 1998

address such issues, hence a combination of formalisms should prove itself to be useful which allows the user freely to merge the different points of view. This is an issue we address in this paper.

The different styles of formalisms are reflected by the different styles of semantics and of implementation:

- Data-flow formalisms are based on a "flow" semantics where each signal is related to a trace of values

  $$v_0 \quad v_1 \quad v_2 \quad \ldots$$

  Data flow programs constrain these traces. They essentially compile into (definitional) equation systems with additional registers. This compilation process amounts to a great extent to a clock resolution calculus.
- On the other hand state-based languages typically rely on an structural operational semantics. For compilation, these are symbolically presented by *purely boolean* equation systems, with additional boolean variables inserted to drive memory operations on data, these operations being stored in an *action table*. This corresponds to a separation between control and data flows.

Combination of formalisms presumes integration of both semantics and compilation schemes. This report sets up such a semantical framework as well as a language of reactive processes on which the translation schemes of the SYNCHRONIE WORKBENCH (SWB [14]) are based. In particular, some semantic invariants are specified which prove to be beneficial for a smooth integration. Section 2 addresses the semantic issues. In Section 3 we introduce an language of synchronous automata, our presentation of reactive processes, discuss invariants, and we present in Section 4 particularly efficient translation schemes for control-based code. Section 5 deals with declarative code, while Section 6 is concerned with combination. Much of the work is based on and extends [12] and [8], as well as [11]. Our style is informal in order to present the ideas in as little space as possible; we assume some familiarity with synchronous languages.

2 The Semantic Framework

2.1 The Model

Behaviour manifests in what we are able or want to observe. We classify observations by attributing a name we refer to as a *signal*. A signal s may be *present* having a value taken from a set $\mathcal{V}$, or it may be *absent*. Let $\mathcal{S}$ be the set of all signals.

We are concerned with *linear time* only as modeled by the ordered set ω of natural numbers. A *trace* δ_s of a signal s is specified by a subset $!\delta_s \subseteq \omega$, the *frequency* of δ_s, and a *valuation* $\delta_s : !\delta \to \mathcal{V}$ to a set of values. A *system trace* δ consists of a set of signal traces $\{\delta_s | s \in S\}$. We use $\Delta^{\mathcal{S}}$ to denote the domain of system traces, and speak of *synchronous* behaviour reflecting awareness of

"global" time which allows to reason about the presence and the absence of a signal.

A system trace $\delta \in \Delta^{\mathcal{S}}$ is called a *flow* if $!\delta := \bigcup !\delta_s$ is downward closed, i.e. $m \in !\delta$ if $n \in !\delta$ and $m \leq n$. The idea is that passing of "time" is bound to an event, i.e. the presence of at least one signal. We refer to a set of flows as a *process*. Note that every set $\mathcal{T} \subseteq \Delta^{\mathcal{S}}$ of system traces determines a set $\mathcal{T}\downarrow$ of those flows which are in $\mathcal{T}$. The domain $\mathcal{P}roc$ of processes will be the semantic domain for synchronous languages.

Reflecting the synchrony hypothesis we require processes to be reactive, and deterministic in the sense below:
Let $\mathcal{I} \subseteq \mathcal{S}$ be a set of *input signals*, and let $\Upsilon \in \Delta^{\mathcal{I}}$ be a set of (admissible) input flows. We say that a process $\mathcal{P}$ is *reactive* with regard to Υ if $\mathcal{P}_{\mathcal{I}} = \Upsilon$ where $\mathcal{P}_{\mathcal{I}} = \{\{\delta_s \mid s \in \mathcal{I}\} \mid \delta \in \mathcal{P}\}$ is the projection of $\mathcal{P}$ to input signals. Further, a process $\mathcal{P}$ is *deterministic* if $|\{\delta \in \mathcal{P} \mid \delta_{\mathcal{I}n} \in \Upsilon\}| \leq 1$. In other words, a process is reactive and deterministic if, for every input flow $\delta \in \Upsilon$, there is exactly one flow $\delta' \in \mathcal{P}$ such that $\delta = \delta'_{\mathcal{I}n}$.

We avoid explicitly discussing *typing* here (and elsewhere) but assume that all the relevant entities are (well-) typed in that the values of a signal s are chosen from a specified set $\mathcal{V}_s$. In particular, we assume existence of a type `bool` of Booleans with a constants `true` and `false`, and existence of a type `unit` with the only value being `void`. In the latter case we speak of a *pure signal* which is fully specified by its frequency.

Synchronous data-flow languages impose constraints on signal traces by lifting relations on data to traces:

$$R(\delta_1, \ldots, \delta_n) \text{ iff } \forall i \in \omega.\ [(R(\delta_1(i), \ldots, \delta_n(i)) \Leftrightarrow i \in \bigcap !\delta_j)].$$

where $R \subseteq \mathcal{V}^n$. Such a lifting is called *strong* if additionally

$$\bigcap !\delta_j \subseteq \bigcup !\delta_j$$

If this isn't the case, we speak of a weak lifting. Strongness implies that all traces in a relation are of the same frequency. A relation is typically obtained by lifting a functional equation

$$\delta = f(\delta_1, \ldots, \delta_n),$$

where $f : \mathcal{V}^n \to \mathcal{V}$ is a function (we assume equality $\delta = \delta'$ to be a special case with f being the identity function).

The more interesting aspect of synchronous data-flow languages is that they provide a variety of operators for manipulating time indexes:

memorisation $\quad \delta' = \mathtt{pre}(\delta) \quad !\delta' = !\delta$

$$\delta'(i) = \begin{cases} \mathtt{init} & \text{if } i = 0 \\ \delta(\max\{m \in !\delta \mid m < i\}) & \text{else} \end{cases}$$

flèche	$\delta'' = \delta$ `->` δ'	$!\delta'' = !\delta' \cap !\delta$ $\delta''(i) = \begin{cases} \delta(i) & \text{if } i = \min !\delta \\ \delta'(i) & \text{else} \end{cases}$
downsampling	$\delta' = \delta$ `when` β	$!\delta' = !\delta \cap !\beta$ $\delta'(i) = \delta(i)$
upsampling	$\delta'' = \delta$ `default` δ'	$!\delta'' = !\delta \cup !\delta'$ $\delta''(i) = \begin{cases} \delta(i) & \text{if } i \in !\delta \\ \delta'(i) & \text{else} \end{cases}$

where β is of type `bool`, and where `init` is an initial value (of appropriate type). All operators except for the default operator are required to be strong. There is nothing particular about our choice of operators. Synchronous data flow languages such as LUSTRE [4] or SIGNAL [7] support a different choice.

Elementary declarative programs consists of set of clauses which are interpreted disjunctively, plus a declaration of of input, output, and local signals, e.g.

```
node raising_edge (x:bool)(y:bool);
let
  y = false -> x and not pre(x);
tel
```

Notation and style considerably differ according to a specific language.

2.2 The Operational Model

We complement our semantic model by an *operational model* which is as simple: Let *synchronous computation* be specified by a labeled transition system $\mathcal{P}$ of the form

$$\sigma \overset{E/E'}{\underset{\mathcal{P}}{\Longrightarrow}} \sigma'$$

where σ, σ' are states, and where $E \neq \emptyset$ indicates which signals are present when changing state. We refer to E as an *event*, and to single transition as an *instant* of time. The set E' specifies which signals are emitted by $\mathcal{P}$ at a given instant. We require that $E' \subset E$, i.e. the output event is part of the overall event. This property is referred to as *consistency*.

An event is specified by a partial function E from $\mathcal{S}$ to $\mathcal{V}$ which we present by its graph, i.e. the set $E \subseteq \mathcal{S} \times V$ such that $v = v'$ whenever $\langle s, v\rangle, \langle s, v'\rangle \in E$ (which justifies the subset notation above). Let $\mathcal{V}_\circ^{\mathcal{S}}$ denote the set of partial functions from $\mathcal{S}$ to $\mathcal{V}$.

States are of the form $\sigma \in \mathcal{V}_\circ^{\mathcal{R}}$ where $\mathcal{R}$ is a finite set of *registers* (where $\mathcal{S} \cap \mathcal{R} = \emptyset$). Registers behave similar to signals; a register $r \in \mathcal{R}$ may be *active* having a valuation $v \in \mathcal{V}$, or it may be *inactive*. The difference between signals

and registers is that signals are set for the present instant while registers are set for the next instant.

In other words, our operational model is a Mealy machine with a particular structure of events and of states. We note that every sequence

$$\sigma_0 \overset{E_0/E'_0}{\underset{\mathcal{P}}{\Longrightarrow}} \sigma_1 \overset{E_1/E'_1}{\underset{\mathcal{P}}{\Longrightarrow}} \cdots$$

of reactions specifies a flow $\delta \in \Delta^{\mathcal{S}}$ such that $n \in !\delta_s$ and $\delta_s(n) = v$ iff $\langle s, v\rangle \in E_n$.

2.3 Synchronous Automata

Favourite synchronous languages such as ESTEREL [1], or ARGOS [9], the synchronous variant of STATECHARTS [5], are closely related to this operational model. We present our own brand of very elementary language, named *synchronous automata*, for programming such machines. Synchronous automata will serve as an intermediate layer for compilation schemes.

A synchronous automaton $\mathcal{P}$ consists of a set of *actions* of the form

$$\textbf{\textit{if}}\ \phi\ \textbf{\textit{then}}\ x = \epsilon$$

where x is a name, ϵ is a *data expression*, and ϕ is a pure (signal) expression (of appropriate form). For the semantics, let an *environment* be defined to be a union $\sigma \cup E$ with $\sigma \in \mathcal{V}_\circ^{\mathcal{R}}$, and $E \in \mathcal{V}_\circ^{\mathcal{S}}$. The notation is justified because of the isomorphism $\mathcal{V}_\circ^{\mathcal{R}+\mathcal{S}} \approx \mathcal{V}_\circ^{\mathcal{R}} \times \mathcal{V}_\circ^{\mathcal{S}}$. Then

$$\langle x, \epsilon(\sigma \cup E)\rangle \in \sigma' \cup E' \text{ iff } \sigma \cup E \models \phi$$

"if the condition ϕ holds in the environment $\sigma \cup E \in \mathcal{V}_\circ^{\mathcal{R}+\mathcal{S}}$, the the register or state x is set with the value obtained by evaluating ϵ in the environment $\sigma \cup E$". A *synchronous component* is a synchronous automaton wrapped with a header:

```
syn_aut raising_edge (α,β:unit;x:bool)(y:bool);
register pre_x:bool;
let
  if α ∨ β then pre_x = x;
  if α then y = false;
  if β then y = x and not pre_x;
tel
```

recodes the declarative raising edge program. The input parameters α and β explicitly represent the time index in the declarative model, α is true at the very first instant of time, and β is true at all instants of time except for the first one. Hence $\alpha \vee \beta$ represents the whole time scale ω. With regard to the original program, the `pre`-operator on time indices has been replaced by a register which stores the previous value.

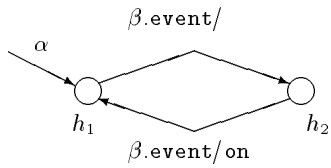
To give another example, let us consider the ESTEREL program

```
module one_bit (event:unit)(on:unit);
  loop
    await event;
    await event;
    emit on
  end
end
```

If started, after each second "event" the signal `on` is emitted. A corresponding synchronous automaton is specified by:

syn_aut *one_bit* (α,β,*event:unit*)(*on:unit*);
register h_1,h_2:*unit*;
let
  if $\alpha \vee$ (*event* $\wedge$ β $\wedge$ h_2) $\vee$ (*not event* $\wedge$ β $\wedge$ h_1) ***then*** h_1 = *void*
  if (*event* $\wedge$ β $\wedge$ h_1) $\vee$ (*not event* $\wedge$ β $\wedge$ h_2) ***then*** h_2 = *void*
  if *event* $\wedge$ β $\wedge$ h_2 ***then*** *on* = *void*
tel

The same automaton implements the ARGOS diagram below:



2.4 Reactivity and Control

For the *semantics* of synchronous automata, we require (for the time being) that all actions are consistent: two actions (***if*** ϕ ***then*** $x = \epsilon$) and (***if*** ϕ' ***then*** $x' = \epsilon'$) are *consistent at* $\sigma \cup E$ if $\epsilon(\sigma \cup E) = \epsilon'(\sigma \cup E)$ whenever $x = x'$ and $\sigma \cup E \models \phi$ and $\sigma \cup E \models \phi'$. Then a synchronous automaton determines a *transition function*

$$\mathcal{P}^{\triangleright} : \mathcal{V}_0^{\mathcal{R}+\mathcal{S}} \to \mathcal{V}_0^{\mathcal{R}}$$

and an *output function*

$$\mathcal{P}^{!} : \mathcal{V}_0^{\mathcal{R}+\mathcal{S}} \to \mathcal{V}_0^{\mathcal{S}}.$$

The reaction of an automaton at an instant may depend on the signals emitted by itself. Hence, given some set E of "inputs", the reaction should be stable in that

stability $\quad \mathcal{P}^{!}(\sigma \cup E) \subseteq E.$

holds. Moreover, we would like to distinguish between signals E which are broadcast by the environment, and those signals E' which are emitted by $\mathcal{P}$. We say that a reaction is *coherent* if each signal is either an "input" or emitted by $\mathcal{P}$:

coherence Given an "input" $E \in \mathcal{S}$, a reaction $E' \in \mathcal{S}$ is *coherent* if $E \subseteq E'$, E' is stable, and if $E' \Leftrightarrow P^!(\sigma, E \cup E') \subseteq E$.

We expect an automaton to react to every input coherently and reproducible:

reactivity For every input, there exists a unique[1] coherent reaction.

Given reactivity, the automaton P defines *reaction functions* $\tilde{P}^!$ and $\tilde{P}^\triangleright$ of the same arity as $P^!$ and $P^\triangleright$ such that $\tilde{P}^!(E) = E'$ and $\tilde{P}^\triangleright(E) = P^\triangleright(E')$ where E' is the unique coherent reaction with regard to the input E. The automaton P may be considered as presenting these reaction functions. Of course, not every automaton is reactive in that $\tilde{P}$ may fail to exist. It will be a matter of *causality analysis* [13] to weed those which are not.

So far, the assumption is that, once started, a system reacts forever. For structuring purposes we would rather have that (sub-) systems may be *active* or *inactive*, or are *in control* or *out of control*, at different stages of evolution. We relate control to a particular set of pure registers $\mathcal{C}$, the *control registers*, and stipulate that a synchronous automaton is out of control if none of these registers is active. Then the automaton should not be able to react except for the trivial reaction. The idea is captured by the *control axiom*:

control $P^\triangleright(\sigma \cup E) = \emptyset$ and $P^!(\sigma \cup E) = \emptyset$ if $\sigma \cap \mathcal{C} = \emptyset$.

Note that a program such as the raising edge program does not a priori satisfy the control axiom. There is no obvious candidate for a control register. In fact, "control" is external here, hidden in the frequency $\chi = \alpha \vee \beta$; the automaton computes only if χ is present, hence the respective parameters.

2.5 The Initial Reaction

So far, synchronous automata only specify ongoing behaviours. Some activation mechanism is needed. We assume that the reaction of a system is immediate if "switched on", hence assume a *initial reaction* (rather than an initial state as in traditional automata theory). The initial reaction should not refer to some previous state in that both the *initial transition function*

$$\mathcal{P}^\triangleright_\alpha : \mathcal{V}^{\mathcal{S}}_\circ \to \mathcal{V}^{\mathcal{R}}_\circ$$

and the *initial output function*

$$\mathcal{P}^!_\alpha : \mathcal{V}^{\mathcal{S}}_\circ \to \mathcal{V}^{\mathcal{S}}_\circ .$$

[1] The condition may be weakened to existence only. We then speak of a *non-deterministic* system.

do not depend on registers.

Syntactically, let $\mathcal{P}_\alpha \subseteq \mathcal{P}$, and $\mathcal{P}_\beta = \mathcal{P} \Leftrightarrow \mathcal{P}_\alpha$. We require that

$$\textit{initialization} \quad \mathcal{P}^{\triangleright}_\alpha(E) \cup \mathcal{P}^{!}_\alpha(E) \neq \emptyset \Rightarrow \mathcal{P}^{\triangleright}_\beta(\sigma \cup E) \cup \mathcal{P}^{!}_\beta(\sigma \cup E) = \emptyset$$

for all $\sigma \in \mathcal{V}^{\mathcal{R}}_\circ$ and $E \in \mathcal{V}^{\mathcal{S}}_\circ$; either an initial reaction takes place exclusively, or an reaction depends on a previous state. Note that $\sigma = \emptyset$ is a state which is, however, persistent due to the control axiom.

We will later use a specific pure *system signal*

α - *start*

which determines the initial reaction of a synchronous automaton. We than can rephrase initialization to

$$\textit{initialization} \quad \alpha \Rightarrow \neg \bigvee \mathcal{C}$$

3 A Language of Synchronous Automata

3.1 Some Basic Operators and Predicates

We introduce an intermediate layer of operators on synchronous automata each of which reflects semantic as well as programming concepts. A first definition of such a language has been given in [8] which carefully elaborates the choices made. Here we define a slightly revised version.

As a first step, we factor synchronous automata into a pure control part, which we refer to as *Boolean automaton* and an "action table". All registers of the Boolean automaton are control registers. The two subautomata communicate via pure signals only. Hence we adopt a control-based view, but this does not restrict generality. Let $\mathcal{P}$ from now on range over Boolean automata.

Boolean automata will be enhanced by three predicates:

$\mathcal{P}.\omega$ - *termination*
$\mathcal{P}.\tau$ - *interrupt*
$\mathcal{P}.\eta$ - *control*

which are synthesized. The predicate $\mathcal{P}.\omega$ evaluates to true if $\mathcal{P}$ terminates, $\mathcal{P}.\tau$ evaluates to true if $\mathcal{P}$ issues an interrupt, and $\mathcal{P}.\eta$ evaluates to true if $\mathcal{P}$ is in control. We expect that the invariant

$$\textit{termination} \quad \mathcal{P}.\omega \Rightarrow \neg \mathcal{P}.\eta'$$

holds, where we use the notation $\ldots'$ to refer to the value at the next instant; if $\mathcal{P}$ terminates all control registers become inactive.

Now the most elementary of our operators are

s <= ϕ $\Leftrightarrow$emit the pure signal s
h <- ϕ $\Leftrightarrow$activate the control register h
nothing $\Leftrightarrow$which does nothing, but terminates
$\mathcal{P} \vee \mathcal{Q}$ $\Leftrightarrow$the union of $\mathcal{P}$ and $\mathcal{Q}$, and
$\phi \wedge \mathcal{P}$ $\Leftrightarrow$guarding $\mathcal{P}$ by the condition ϕ,

where $\textbf{\textit{if}}\ \phi \wedge \psi\ \textbf{\textit{then}}\ x = \epsilon$ is an action of $\phi \wedge \mathcal{P}$ if $\textbf{\textit{if}}\ \psi\ \textbf{\textit{then}}\ x = \epsilon$ is an action of $\mathcal{P}$. We stipulate:

- Emittance is defined by

$$s\ \texttt{<=}\ \phi \quad :\Leftrightarrow \quad \textbf{\textit{if}}\ \phi\ \textbf{\textit{then}}\ s = \mathit{void}.$$

  It terminates instantaneously, but neither raises an interrupt, nor keeps control:

$$(s\ \texttt{<=}\ \phi).\omega = \mathit{tt}$$
$$(s\ \texttt{<=}\ \phi).\tau = \mathit{ff}$$
$$(s\ \texttt{<=}\ \phi).\eta = \mathit{ff}$$

- Activation is defined by

$$h\ \texttt{<-}\ \phi \quad :\Leftrightarrow \quad \mathit{if}\ \phi\ \mathit{then}\ h = \mathit{void}.$$

  If the register is activated then control is kept, and no termination takes place. No interrupt is raised:

$$(h\ \texttt{<=}\ \phi).\omega = \neg\mathtt{h}$$
$$(h\ \texttt{<=}\ \phi).\tau = \mathit{ff}$$
$$(h\ \texttt{<=}\ \phi).\eta = \mathtt{h}$$

- *nothing* has the empty set of actions

$$\mathit{nothing} \quad :\Leftrightarrow \quad \emptyset,$$

  and it terminates instantaneously

$$\mathit{nothing}.\omega = \mathit{tt}$$
$$\mathit{nothing}.\tau = \mathit{ff}$$
$$\mathit{nothing}.\eta = \mathit{ff}$$

- Disjunction means disjunction:

$$(\mathcal{P} \vee \mathcal{Q}).\omega = \mathcal{P}.\omega \vee \mathcal{Q}.\omega$$
$$(\mathcal{P} \vee \mathcal{Q}).\tau = \mathcal{P}.\tau \vee \mathcal{Q}.\tau$$
$$(\mathcal{P} \vee \mathcal{Q}).\eta = \mathcal{P}.\eta \vee \mathcal{Q}.\eta$$

- Guarding raises an interrupt signal, and restricts termination, but does not affect control[2]:

$$(\phi \wedge \mathcal{P}).\omega = \phi \wedge \mathcal{P}.\omega$$
$$(\phi \wedge \mathcal{P}).\tau = \phi$$
$$(\phi \wedge \mathcal{P}).\eta = \mathcal{P}.\eta$$

[2] This abstracts a mechanism introduced by Reinhard Budde for compilation of *synchronous Eifel* [3]

Further, we have already introduced a number of operators implicitly:

$\mathcal{P}_\alpha$ - the initial reaction
$\mathcal{P}_\beta$ - the ongoing reaction
$\mathcal{P}^\triangleright$ - the transition function
$\mathcal{P}^!$ - the output function

This completes the set of basic operators.

A number of familiar operators are easily specified in terms of these base operators. We give a couple of examples using our syntactical convention that the system signal α specifies the first reaction.

$$emit\ s = s\ \texttt{<=}\ \alpha \tag{1}$$
$$halt^h = h\ \texttt{<-}\ \alpha \vee h \tag{2}$$
$$start\ \mathcal{P}\ at\ \phi = (\phi \wedge \mathcal{P}_\alpha) \vee \mathcal{P}_\beta \tag{3}$$
$$if\ \phi\ then\ \mathcal{P}\ else\ \mathcal{Q}\ fi = (start\ \mathcal{P}\ at\ \phi) \vee (start\ \mathcal{Q}\ at\ \neg\phi) \tag{4}$$
$$if\ \phi\ then\ \mathcal{P}\ fi = if\ \phi\ then\ \mathcal{P}\ else\ nothing\ fi \tag{5}$$
$$\mathcal{P}\ ;\ \mathcal{Q} = \mathcal{P} \vee (start\ \mathcal{Q}\ at\ \mathcal{P}.\omega) \tag{6}$$
$$loop\ \mathcal{P}\ end = \mathcal{P} \vee (start\ \mathcal{P}\ at\ \mathcal{P}.\omega) \tag{7}$$
$$terminate\ \mathcal{P}\ if\ \phi = \mathcal{P}^! \vee (\neg\phi \wedge \mathcal{P}^\triangleright) \tag{8}$$
$$terminate\ \mathcal{P}\ if\ next\ \phi = \mathcal{P}^! \vee \mathcal{P}^\triangleright_\alpha \vee (\neg\phi \wedge \mathcal{P}^\triangleright_\beta) \tag{9}$$
$$cancel\ \mathcal{P}\ if\ \phi = \neg\phi \wedge \mathcal{P} \tag{10}$$
$$cancel\ \mathcal{P}\ if\ next\ \phi = \mathcal{P}_\alpha \vee (\neg\phi \wedge \mathcal{P}_\beta) \tag{11}$$
$$await\ \phi = cancel\ halt^h\ \ if\ \phi \tag{12}$$
$$await\ next\ \phi = cancel\ halt^h\ \ if\ next\ \phi \tag{13}$$
$$do\ \mathcal{P}\ when\ \phi = (\phi \wedge \mathcal{P}) \vee (\neg\phi \wedge KEEP) \tag{14}$$
$$do\ \mathcal{P}\ when\ next\ \phi = \mathcal{P}_\alpha \vee (\phi \wedge \mathcal{P}_\beta) \vee (\neg\phi \wedge KEEP) \tag{15}$$

where $KEEP = \{h\ \texttt{<-}\ h \mid h \in \mathcal{C}\}$. In words:

1. The signal s is emitted in the first instant only.
2. The control register h is activated in the first instant, and then kept activated.
3. $\mathcal{P}$ starts only if the condition ϕ holds.
4. If ϕ holds then $\mathcal{P}$ is executed, otherwise $\mathcal{Q}$.
5. obvious.
6. $\mathcal{P}$ computes first. If $\mathcal{P}$ terminates, $\mathcal{Q}$starts to compute.
7. $\mathcal{P}$ is immediately reinitialised if it terminates.
8. $\mathcal{P}$ looses control if ϕ holds (*weak preemption*)
9. As above but preemption does not take place in the first instant.
10. As (8), but signals are not emitted either (*strong preemption*)
11. As above but preemption does not take place in the first instant.

12. Whenever the control register h gets control, control will stay with h till the condition ϕ holds. If ϕ holds in the first instant, h does not get control.
13. As above, but h gets control in the first instant whether or not the condition holds.
14. $\mathcal{P}$ is down sampled in that $\mathcal{P}$ computes only if the condition ϕ holds, otherwise the present status of control registers is kept. Note that the process is out of control instantaneously if the condition ϕ holds in the first instant, but does not terminate.
15. Here $\mathcal{P}$ is started in the first instant whatever ϕ says.

With regard to preemption it may be worthwhile to observe that the diagram

signals at first instant	*signals at later instants*
registers at first instant	*registers at later instants*

displays all the preemption strategies of our model. Overall there are sixteen different strategies starting with no preemption at all up to preemption of all signals and registers. The latter is the strategy specified by the operator *cancel* $\mathcal{P}$ *if* ϕ. A modification is to cancel only at later instants which covers the two squares on the right, and which more or less corresponds to the *do ... watching* mechanism of ESTEREL. Preemption by termination does not affect signals, hence covers only the two lower squares, being a mild variant of the *trap*-statement of ESTEREL. If applied only in later instants, termination covers exactly the lower square on the right. The latter corresponds to the preemption mechanisms used in ARGOS.

3.2 Concerning Compositionality

Our set of base operators has several defects with regard to compositionality, meaning that our semantical requirements / invariants are not preserved.

Most notably, disjunction does not preserve reactivity: e.g.

$$(a \mathrel{<=} b) \vee (b \mathrel{<=} a)$$

is not well behaved though its components are. This is well known and inherent, and there is no way to for a compositional analysis of reactivity. Hence we abandon any hope of a compositional solution but use global causality analysis as everybody else does.

However, disjunction does not preserve the control axiom either: e.g.

$$(h \mathrel{<=} tt) \vee (a \mathrel{<=} tt)$$

terminates though it keeps control. In fact, disjunction is only a very useful auxiliary operator, and the basis of

$\mathcal{P} \otimes \mathcal{Q}$ - parallel composition

where $\mathcal{P} \otimes \mathcal{Q}$ is equivalent to $\mathcal{P} \vee \mathcal{Q}$ except that we have a new termination condition:

$$(\mathcal{P} \otimes \mathcal{Q}).\omega = (\mathcal{P}.\omega \wedge \mathcal{Q}.\omega) \vee (\mathcal{P}.\omega \wedge \mathcal{P}.\eta \wedge \neg\mathcal{Q}.\eta) \vee (\mathcal{Q}.\omega \wedge \mathcal{Q}.\eta \wedge \neg\mathcal{P}.\eta)$$

The parallel composition of $\mathcal{P}$ and $\mathcal{Q}$ terminates if both automata terminate at the same instant ($\mathcal{P}.\omega \wedge \mathcal{Q}.\omega$), or if $\mathcal{Q}$ has already terminated and $\mathcal{P}$ terminates ($\mathcal{P}.\omega \wedge \mathcal{P}.\eta \wedge \neg\mathcal{Q}.\eta$), and vice versa. As a point of fine tuning, observe that, in the latter cases, $\mathcal{P}$ must be in control (i.e. $\mathcal{P}.\eta$ holds); otherwise, e.g., $\mathcal{P}$ may terminate in the first instant, hence the parallel computation as well since $\mathcal{Q}.\eta$ must be false because of the initialization axiom. However, $\mathcal{Q}$ may gain control as in $(h \texttt{ <= } tt) \vee (a \texttt{ <= } tt)$.

As a kind of dual, we have

$\mathcal{P} \oplus \mathcal{Q}$ - choice

where $\mathcal{P} \oplus \mathcal{Q}$ is equivalent to $\mathcal{P} \vee \mathcal{Q}$ except that we require that

$$\mathcal{P}^{\triangleright}_{\alpha}(E) \cap \mathcal{Q}^{\triangleright}_{\alpha}(E) = \emptyset;$$

only one of $\mathcal{P}$ or $\mathcal{Q}$ can obtain control. We can now state a first "compositionality" result:

Proposition 1. *The operators* s `<=` ϕ, h `<-` ϕ, `nothing`, $\mathcal{P} \otimes \mathcal{Q}$, $\mathcal{P} \oplus \mathcal{Q}$, *and* $\phi \wedge \mathcal{P}$ *preserve the control axiom.*

Having in mind this proposition, the resulting strategy should be to replace the ill-behaved $\mathcal{P} \vee \mathcal{Q}$ by the well-behaved $\mathcal{P} \otimes \mathcal{Q}$ or $\mathcal{P} \oplus \mathcal{Q}$ exploiting

Lemma 2. $\mathcal{P}_{\alpha} \vee \mathcal{P}_{\beta} = \mathcal{P}_{\alpha} \otimes \mathcal{P}_{\beta}$

Inspection of the derived operators above proves that we could have used

$$\begin{aligned}
\mathit{start}\ \mathcal{P}\ \mathit{at}\ \phi &= (\phi \wedge \mathcal{P}_{\alpha}) \otimes \mathcal{P}_{\beta} \\
\mathit{if}\ \phi\ \mathit{then}\ \mathcal{P}\ \mathit{else}\ \mathcal{Q}\ \mathit{fi} &= (\phi \wedge \mathcal{P}_{\alpha}) \oplus \mathcal{P}_{\beta} \vee (\neg\phi \wedge \mathcal{Q}_{\alpha}) \vee \mathcal{Q}_{\eta} \\
\mathcal{P}\ ;\ \mathcal{Q} &= \mathcal{P} \oplus (\mathit{start}\ \mathcal{Q}\ \mathit{at}\ \mathcal{P}.\omega) \\
\mathit{loop}\ \mathcal{P}\ \mathit{end} &= (\mathcal{P} \otimes (\mathit{start}\ \mathcal{P}\ \mathit{at}\ \mathcal{P}.\omega) \\
\mathit{terminate}\ \mathcal{P}\ \mathit{if}\ \phi &= \mathcal{P}^{!} \otimes (\neg\phi \wedge \mathcal{P}^{\triangleright}) \\
\mathit{cancel}\ \mathcal{P}\ \mathit{if}\ \phi &= \neg\phi \wedge \mathcal{P} \\
\mathit{terminate}\ \mathcal{P}\ \mathit{if\ next}\ \phi &= \mathcal{P}^{!} \oplus \mathcal{P}^{\triangleright}_{\alpha} \otimes (\neg\phi \wedge \mathcal{P}^{\triangleright}_{\beta}) \\
\mathit{cancel}\ \mathcal{P}\ \mathit{if\ next}\ \phi &= \mathcal{P}_{\alpha} \otimes (\neg\phi \wedge \mathcal{P}_{\beta}) \\
\mathit{do}\ \mathcal{P}\ \mathit{when}\ \phi &= (\phi \wedge \mathcal{P}) \oplus (\neg\phi \wedge \mathit{KEEP}) \\
\mathit{do}\ \mathcal{P}\ \mathit{when\ next}\ \phi &= \mathcal{P}_{\alpha} \otimes (\phi \wedge \mathcal{P}_{\beta}) \oplus (\neg\phi \wedge \mathit{KEEP})
\end{aligned}$$

rather than the original definitions.

4 Adding Efficiency

4.1 System Signals

Implementation of a concrete languages has one other prerogative beside semantic transparency: efficiency of the resultant code in terms of time and space. The operators on synchronous automata may obviously fail to produce such code. However, concrete languages use the operators in rather specific ways. If we are able to discern these specific patterns, and to implement them efficiently, we may get the best of both worlds, both semantic transparency and efficiency of the generated code.

We use particular pure signals, so-called *system signals*, to represent semantic concepts in a convenient fashion. One of these signals has already been introduced, the *start* signal α. The idea of α is to specify the notion of the first instant, hence α corresponds to the operator $\mathcal{P}_\alpha$: let $\mathcal{P}$ be a synchronous automaton. Let us define

$$\mathcal{P}_\alpha := \mathcal{P}[ff/\mathcal{C}] \quad \text{and} \quad \mathcal{P}_\beta := \mathcal{P}[ff/\alpha]$$

where $\mathcal{P}[ff/\alpha]$ states that we substitute ff for α on the right hand side of actions in $\mathcal{P}$. Similarly, $\mathcal{P}[ff/\mathcal{C}]$ states that ff is substituted for each of the control registers. We give an example rather than a formal definition:

$$\begin{aligned} h_1 &<\text{-}\ \alpha \vee (event \wedge \beta \wedge h_2) \vee (\neg event \wedge h_1) \\ h_2 &<\Leftrightarrow (event \wedge \beta \wedge\ h_1) \vee (\neg event \wedge h_2) \\ on &<\text{=}\ event \wedge\ h_2, \end{aligned}$$

where $\mathtt{h}_1$ and $\mathtt{h}_2$ are control registers. Then $\mathcal{P}_\alpha$ is defined by

$$h_1 <\text{-}\ \alpha,$$

and $\mathcal{P}_\beta$ by

$$\begin{aligned} h_1 &<\text{-}\ (event \wedge \beta \wedge h_2) \vee (\neg event \wedge h_1) \\ h_2 &<\text{-}\ (event \wedge \beta \wedge\ h_1) \vee (\neg event \wedge h_2) \\ on &<\text{=}\ event \wedge\ h_2. \end{aligned}$$

In order to make the start system semantically well behaved we require that

$$(\alpha) \qquad \mathcal{P} \cong \mathcal{P}_\alpha \otimes \mathcal{P}_\beta.$$

The example, quite deliberately, suggests existence of a system signal β, we refer to as *run* signal; if β is not present, no computation will take place but the status of control variables is retained. The run signal provides for a simple implementation of the *when next*-operator, or dually the *suspend*-operator:

$$\mathcal{P}\ \ when\ next\ \phi = \mathcal{P}[\phi \wedge \beta/\beta].$$

The run signal behaves semantically correct if

$$(\beta) \qquad \mathcal{P} \cong \mathcal{P}_\alpha \otimes (\beta \wedge \mathcal{P}_\beta \oplus \neg\beta \wedge KEEP).$$

Next, we have a system signal τ, the *preempt* signal, which, if present, deactivates all control registers for the next instant. Semantically we require that

$$(\tau) \qquad \mathcal{P} \cong \mathcal{P}^{\shortmid} \oplus \neg\tau \wedge \mathcal{P}^{\triangleright}.$$

These are all the system signals we shall use for efficient translation schemes of control based programs.

4.2 Starring

Substitution is an "expensive" operation due to possible code multiplication. In order to be more efficient we can use "lazy" substitution, i.e. we introduce a *local* signal, bind the respective term to this signal, and substitute the signal instead of the term:

$$\mathcal{P}[\phi/x] \quad \text{becomes} \quad \mathcal{P}[\gamma/x] \vee (\gamma \ \texttt{<=}\ \phi)$$

This operation is constant in size, however not quite equivalent in our setup. For example, consider $\mathcal{P}[\alpha \vee h/\alpha]$ where $\mathcal{P} = (a \ \texttt{<=}\ \alpha)$, and where h is a control register. Clearly $\mathcal{P}[\alpha \vee h/\alpha] = (a \ \texttt{<=}\ \alpha \vee h)$, and

$$\mathcal{P}[\alpha \vee h/\alpha]_\alpha = (a \ \texttt{<=}\ \alpha) \quad \text{and} \quad \mathcal{P}[\alpha \vee h/\alpha]_\beta = (a \ \texttt{<=}\ h)$$

On the other hand we have

$$(\mathcal{P}[\gamma/\alpha] \oplus (\gamma \ \texttt{<=}\ \alpha \vee h))_\alpha = (\gamma \ \texttt{<=}\ \alpha) \vee (a \ \texttt{<=}\ \gamma)$$
$$(\mathcal{P}[\gamma/\alpha] \oplus (\gamma \ \texttt{<=}\ \alpha \vee h))_\beta = (\gamma \ \texttt{<=}\ h) \vee (a \ \texttt{<=}\ \gamma)$$

Obviously, the operators of our algebra of concrete synchronous automata are not compositional with regard to local variables.

We resolve the problem by adding a new operation, *starring*, which renames all local signals. Let $\mathcal{L}(\mathcal{P})$ be the set of local signals of $\mathcal{P}$. Then

$$\mathcal{P}^* := \mathcal{P}[s^*//s \mid s \in \mathcal{L}(\mathcal{P})]$$

where $\mathcal{P}[s^*//s \mid s \in \mathcal{L}(\mathcal{P})]$ states that every local signal $s \in \mathcal{L}(\mathcal{P})$ is renamed to s^* everywhere in $\mathcal{P}$ (this is different from substitution which affects only the ϕ's in $s \ \texttt{<=}\ \phi$ and $h \ \texttt{<-}\ \phi$). Again an example should be sufficient to grasp the idea:

$$((\gamma \ \texttt{<=}\ \alpha) \vee (a \ \texttt{<=}\ \gamma))^* = (\gamma^* \ \texttt{<=}\ \alpha) \vee (a \ \texttt{<=}\ \gamma^*),$$

where γ is a local variable. Thus we should in general redefine $\mathcal{P}_\alpha$ to

$$\mathcal{P}_\alpha = \mathcal{P}[ff/\mathcal{C}]^*.$$

Then lazy substitution is compositional, with our trivial example hopefully being enough of a witness to substatiate the claim. These observations have been used in [12] where we give a translation of ESTEREL and prove its correctness.

4.3 Reincarnation

The staring operator resolves another subtlety of synchronous languages, which is related to the loop construct: reincarnation [1]. For example, if control is with *await* c in

```
loop signal a in if a then emit b; await c; emit a end end
```

then presence of c will imply emittance of a, reinitialisation of the loop, evaluation of the *if*-statement, and finally control will again be with the *await*-statement. The question is whether b will be emitted or not. It should not: in a block structured language, leaving a block will forget all bindings. Entering the block a new incarnation of bound variables, here a, is generated. Due to coherence this new incarnation cannot be present, hence b is not emitted. Note that we have two incarnations of a existing at the same instant, one incarnation being present, the other not.

With a being local, starring generates two copies, a and a^*, the latter covers the reentry of the loop. Applying the definitions we roughly obtain the automaton

$$\begin{aligned} b &\texttt{ <= } a^* \wedge (\alpha \vee c \wedge h) \\ h &\texttt{ <- } \alpha \vee (c \wedge h) \vee (\neg c \wedge h) \\ a &\texttt{ <= } c \wedge h \end{aligned}$$

which behaves perfectly well.

4.4 Complexity

The *splitting* of $\mathcal{P}$ into $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ may cause quadratic growth. Assume we have the action

$$\mathcal{P} \quad \equiv \quad (a \texttt{ <= } b \wedge (\alpha \vee h_1))$$

with h being a control register. Then splitting generates two copies of b,

$$\begin{aligned} \mathcal{P}_\alpha &\quad \equiv \quad (a \texttt{ <= } b \wedge \alpha) \\ \mathcal{P}_\beta &\quad \equiv \quad (a \texttt{ <= } b \wedge h_1) \end{aligned}$$

Now we may need to substitute $(\mathcal{P}_\alpha \otimes \mathcal{P}_\beta)[\gamma/\alpha] \otimes \{\gamma \texttt{ <= } c \wedge (\alpha \vee h_2)\}$, and to split again which leaves us with three copies of b,

$$\begin{aligned} (\mathcal{P}_\alpha \otimes \mathcal{P}_\beta)[\gamma/\alpha]_\alpha &\quad \equiv \quad (a \texttt{ <= } b \wedge \gamma*) \vee (\gamma^* \texttt{ <= } b \wedge \alpha) \\ (\mathcal{P}_\alpha \otimes \mathcal{P}_\beta)[\gamma/\alpha]_\beta &\quad \equiv \quad (a \texttt{ <= } b \wedge h_1) \vee (a \texttt{ <= } b \wedge \gamma) \vee (\gamma \texttt{ <= } b \wedge h_2). \end{aligned}$$

In general, we get a geometric sum $\sum_{i=1}^{n} i = n^2$ with n being the number of nested splittings.

Further, splitting is expensive in terms of translation time since every subexpression needs to be touched. Hence, splitting should be avoided whenever this is feasible.

4.5 An Imperative Core

With these definition, the basic operators of an imperative synchronous language can be implemented as follows, omitting the predicates $\mathcal{P}.\omega, \mathcal{P}.\beta, \mathcal{P}.\eta$ the definition of which are not changed:

$$\begin{aligned}
start\ P\ at\ \phi\ ^{\alpha\ \beta\ \tau} &= P^{\gamma\ \beta\ \tau} \vee (\gamma\ <=\ \phi)\\
loop\ P\ end^{\alpha\ \beta\ \tau} &= P_{\alpha}^{\alpha\ \beta\ \tau} \vee P^{\gamma\ \beta\ \tau} \vee (\gamma\ <=\ P^{\alpha\ \beta\ \tau}.\omega)\\
terminate\ P\ if\ \phi\ ^{\alpha\ \beta\ \tau} &= P^{\alpha\ \beta\ \tau'} \vee (\tau'\ <=\ \tau \vee \phi)\\
terminate\ P\ if\ next\ \phi\ ^{\alpha\ \beta\ \tau} &= P^{\alpha\ \beta\ \tau'} \vee (\tau'\ <=\ \tau \vee (\phi \wedge P^{\alpha\ \beta\ \tau}.\eta))\\
do\ P\ when\ next\ \phi\ ^{\alpha\ \beta\ \tau} &= P^{\alpha\ \beta'\ \tau} \vee (\beta'\ <=\ \beta \wedge \phi)
\end{aligned}$$

All the signals named with Greek letters are local except for α, β, τ. We have

Proposition 3. *All these operators satisfy the control axiom, the initialization axiom, and they satisfy the* $(\alpha), (\beta), (\tau)$ *axiom.*

The proof works very much along the lines of that given in [12] except that we have axiomatized some of invariants in this proof. Most cases are in fact, straightforward but tedious.

All the other operators are in fact derived:

$$\begin{aligned}
do\ \mathcal{P}\ when\ \phi &= start\ (do\ \mathcal{P}\ when\ next\ \phi)\ at\ \phi\\
cancel\ \mathcal{P}\ if\ \phi &= cancel\ (do\ \mathcal{P}\ \ when\ \neg\phi)\ if\ \phi\\
cancel\ \mathcal{P}\ if\ next\ \phi &= terminate\ (do\ \mathcal{P}\ \ when\ next\ \neg\phi)\ if\ next\ \phi
\end{aligned}$$

We have reduced the number of splitting to one case, the *loop*-construct, but the more intuitive definition

$$loop\ P\ end = P^{\gamma\ \beta\ \tau} \vee (\gamma\ <=\ \alpha \vee P^{\alpha\ \beta\ \tau}.\omega)$$

may be incorrect semantically. In an inductive proof, the initialisation axiom is crucial for avoiding the splitting operator. However, this may fail to hold in case of the "intuitive " definition of the loop; if we reenter a loop some control register of $\mathcal{P}$ must have been active in contrast to the initialisation axiom. Hence proper use of the start symbol cannot be guaranteed. Our definition circumvents the problem because there are no interferences between $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$, which means that the initialisation axiom holds (by brute force so to speak). Splitting is, however, in many cases unnecessary, and it is a matter of efficiency to anticipate such cases.

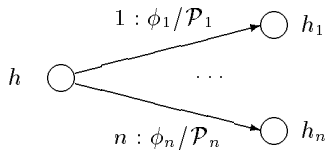
The operators discussed are, modulo syntactic sugar, those of ESTEREL V5 except for traps which need an additional attribute not covered here (but in [12]). We omit the discussion of traps not only because of the additional space needed but as well as we believe that traps are pragmatically unsound in that few users control the inherent priority scheme.

4.6 State-based Control

As an exercise we give the translation of a state-based language such as ARGOS or STATECHARTS in terms of the operators above. Let each state h correspond to a control register h. We collect all the out-going transitions from h to some h_i which we assume to be triggered if a condition ϕ_i holds. While changing state a synchronous automaton $\mathcal{P}_i$ will be executed. We use the textual notation

$$trans\ h\ :\ \phi_1/P_1;\ldots;\phi_n/\mathcal{P}_n\ end$$

for



The numbering specifies priorities in the graphical presentation. Then, for each state h, we have

$$trans\ h\ :\ \phi_1/\mathcal{P}_1;\ldots;\phi_n/\mathcal{P}_n\ end^{\alpha\ \beta\ \tau} = HALT \otimes TRAN$$

where

$$\begin{aligned} HALT &\equiv (start\ (cancel\ halt^h\ \ at\ \tau_h)\ \ at\ \alpha_h) \vee\ \ (\tau_h\ \ \texttt{<=}\ \tau \vee TRAN.\tau) \\ TRAN &\equiv start\ (if\ \phi_1\ \ then\ \mathcal{P}_1; emit\ \alpha_{h_1}\ \ end; \ldots \\ &\qquad\qquad ; \phi_n \wedge (\mathcal{P}_n; emit\ \alpha_{h_1}))\ \ at\ h \end{aligned}$$

For each state h, α_h is a local signal: if α_h is present the control register h is activated for the next instant. Only then $TRAN$ is initialized. If one of the conditions ϕ_i holds the register h will be preempted, and the $\mathcal{P}_i$ will be executed.

For the hierarchical structure one may add

$$trans\ h = \mathcal{P}\ \ :\ \ \phi_1/\mathcal{P}_1;\ldots;\phi_n/\mathcal{P}_n\ \ end$$

to state that $\mathcal{P}$ refines the state h, and redefine $HALT$ to:

$$\begin{aligned} HALT \equiv\ & (start\ (cancel\ halt^h \otimes (do\ \mathcal{P}\ \ when\ \ next\ h)\ \ at\ \tau_h)\ \ at\ \alpha_h) \\ & \vee\ \ (\tau_h\ \ \texttt{<=}\ \tau \vee TRAN_\eta) \end{aligned}$$

4.7 Data

Dealing with data one has to be more specific about the "action table'. We consider here only a very simple kind of data action, *assignment to a memory cell*:

$$(\boldsymbol{if}\ \chi\ \ \boldsymbol{then}\ x := \epsilon) \equiv (\boldsymbol{if}\ \chi\ \ \boldsymbol{then}\ x = \epsilon) \vee (\boldsymbol{if}\ \neg\chi\ \ \boldsymbol{then}\ x = x)$$

Here χ is a pure *trigger* signal, x is a data register, and ϵ is a data expression (of suitable type). At every instant, the data expression is computed and its values assigned to the memory cell, provided that the trigger signal is present. Otherwise the old value is kept. Hence the assignment notation. Emittance of a value is then implemented by

$$emit\ s(\epsilon) = (s\ \texttt{<=}\ \alpha) \vee (a\ \texttt{<=}\ \alpha) \vee (\textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ ?s := \epsilon)$$

where a is a pure (non-local !) signal. We use a "dual-rail" implementation in that every *valued signal* s is presented by a pure signal and a memory register $?s$. The notation $?s$ refers to the value of the register if used in an expression. The declaration of a valued signal is of the form

$$signal\ x\ (:type)\ in\ \mathcal{P}\ end.$$

The signal is pure if the type information is missing.

Since control may depend on values of the language we allow to use $?b$ as a pure atomic expression (overloading notation). This completes the simple protocol of how a Boolean automaton and an action table communicate.

5 Declarative Code - Dealing with Frequencies

5.1 Generating Synchronous Automata

Declarative statements define constraint on flows (cf. Section 2.1). We will concentrate on two such statements, *declarations*

$$\textit{flow}\ x\ (:type)\ \textit{when}\ \mathcal{E}\ \textit{in}\ \mathcal{P}\ \textit{end}$$

and *equations*

$$x = \mathcal{E}$$

We assume that the declaration specifies the frequency $!x$ of x which, as we freely admit, is a restricted view related to LUSTRE. $\mathcal{P}$ is a synchronous automaton, and $\mathcal{E}$ is a *valued synchronous automaton*, i.e. a pair $\mathcal{E} = \langle \epsilon, \mathcal{E} \rangle$ with ϵ being an data expression, and $\mathcal{E}$ being a synchronous automaton (overloading notation). We require that $\mathcal{E}_\eta = \textit{ff}$, i.e. the automaton related to an expression terminates instantaneously. The value of $\mathcal{E}$ is computed at the frequency $!\mathcal{E}$ which is a synthesized predicate.

Let, for every pure signal s, $!s$ determine its "base frequency" such that $!s \prec s$ where

$$s \prec s' \quad \text{iff, for all events } E, \text{ if } s \in E \text{ then } s' \in E$$

defines a preorder on signals; if s is part of an event then s' is part of it as well.

Then declarations translate to (forgetting about types)

$$\textit{flow}\ x\ \textit{when}\ \mathcal{E}\ \textit{in}\ \mathcal{P}\ \textit{end} \equiv \mathcal{E} \otimes \mathcal{P} \vee (!x\ \texttt{<=}\ !\mathcal{E} \wedge ?b) \vee (\textbf{\textit{if}}\ !\mathcal{E}\ \textbf{\textit{then}}\ ?b := \epsilon)$$

where

$$(\mathit{flow}\ldots).\omega = \mathcal{P}.\omega$$
$$(\mathit{flow}\ldots).\tau = \mathcal{P}.\tau$$
$$(\mathit{flow}\ldots).\eta = \mathcal{P}.\eta$$

and equations to

$$(x = \mathcal{E}) \equiv \mathcal{E} \vee (\chi \ \texttt{<=}\ !x \wedge\ !\mathcal{E}) \vee (\textbf{\textit{if}}\ \chi\ \textbf{\textit{then}}\ ?x := \epsilon)$$

where

$$(x = \mathcal{E}).\omega = \mathit{tt}$$
$$(x = \mathcal{E}).\tau = \mathit{ff}$$
$$(x = \mathcal{E}).\eta = \mathit{ff}.$$

Data expressions generate valued synchronous automata in a straightforward way:

$$op(\mathcal{E}_1, \ldots, \mathcal{E}_n) \equiv \langle op(\epsilon_1, \ldots, \epsilon_n), \mathcal{E}_1 \otimes \ldots \otimes \mathcal{E}_n \rangle$$
$$!op(\mathcal{E}_1, \ldots, \mathcal{E}_n) = \bigwedge\, !\mathcal{E}_j$$

Flow variables translate to

$$x = \langle ?x, \mathit{stop} \rangle.$$

All declarative synchronous languages share the concept of memorization which is an operator on expressions, and which is implemented by

$$\mathit{pre}(\mathcal{E}) \equiv \langle m, \mathcal{E} \vee (\textbf{\textit{if}}\ !\mathcal{E}\ \textbf{\textit{then}}\ m\ \texttt{:=}\ \epsilon) \rangle$$
$$!\mathit{pre}(\mathcal{E}) = !\mathcal{E}.$$

m is a new register. A complementary initialization operator has many incarnations, LUSTRE uses $\mathcal{E}_1$ `->` $\mathcal{E}_2$ which translates to

$$\begin{aligned}(\mathcal{E}_1\ \texttt{->}\ \mathcal{E}_1) \equiv \langle \epsilon, \mathcal{E}_1 \otimes \mathcal{E}_2 \vee \\ \{\chi_\alpha\ \texttt{<=}\ !\mathcal{E}_1 \wedge \neg h, \\ \textbf{\textit{if}}\ \chi_\alpha\ \textbf{\textit{then}}\ \epsilon := \epsilon_1, \\ \chi_\beta\ \texttt{<=}\ !\mathcal{E}_2 \wedge h, \\ \textbf{\textit{if}}\ \chi_\beta\ \textbf{\textit{then}}\ \epsilon := \epsilon_2\}\rangle\end{aligned}$$
$$!(\mathcal{E}_1\ \texttt{->}\ \mathcal{E}_2) = !\mathcal{E}_1 \wedge !\mathcal{E}_2$$

The pure register h allows to distinguish the first instant the frequency $!\mathcal{E}_1$ is present from later instants. The assumption is that h is inactive when starting the computation. However, h is not a control register.

Downsampling is implemented by

$$\begin{aligned}(\mathcal{E}_1 \ \mathit{when}\ \mathcal{E}_2) \equiv \langle \epsilon, (\mathcal{E}_1 \ \mathit{when}\ \chi) \vee\ & \mathcal{E}_2 \\ \vee\ & (\mathit{if}\ !\mathcal{E}_1\ \mathit{then}\ b := \epsilon_2) \vee\ (\chi\ \texttt{<=}\ !\mathcal{E}_1 \wedge b) \\ \vee\ & (\mathit{if}\ \chi\ \mathit{then}\ \epsilon := \epsilon_1)\rangle \\ !(\mathcal{E}_1 \ \mathit{when}\ \mathcal{E}_2) = \chi & \\ !\chi = !\mathcal{E} &\end{aligned}$$

At the frequency of $\mathcal{E}_1$, the value is changed if $\mathcal{E}_2$ computes to $\sharp$. The new value is that of $\mathcal{E}_1$ at this instant. The upsampling operator in LUSTRE the *current*-operator:

$$\begin{aligned}\mathit{current}(\mathcal{E}) &\equiv \mathcal{E} \\ !\mathit{current}(\mathcal{E}) &= !!\mathcal{E}\end{aligned}$$

Changes only depend on $\mathcal{E}$, otherwise the value is latched, but on the faster frequency.

5.2 Checking Frequencies

While being an acceptable implementation the above fails to satisfy strongness of equality (cf. Section 2.1) for which one needs that

$$!x \prec !\mathcal{E} \quad \text{and} \quad !\mathcal{E} \prec !x.$$

Frequency analysis is applied to reject programs which do not satisfy such frequency requirements, similar to causality analysis which rejects causally incorrect programs. Frequency analysis, of course, is a problem of comparing boolean flows, ergo maps to the satisfiability problem of Boolean expression which is NP-hard. The problem gets even worse due to the presence of memorization. To reduce complexity, only an approximation $\chi \prec\!\!\prec \chi'$ of $\chi \prec \chi'$ is considered. The approximations may be more or less sophisticated. SIGNAL offers an elaborate "clock" calculus, while LUSTRE promote a more down-to-earth approach which will be sketched.

In LUSTRE the frequency of a flow is determined by its declaration

```
flow x (: type) when e
```

Let us bind `e` to x by $\Delta(x) = \texttt{e} :: \Delta(\texttt{e})$ to obtain a stack of expressions, where

$$\begin{aligned}\Delta(\texttt{op}()) &= [\,] \\ \Delta(\texttt{op}(\texttt{e}_1, \ldots, \texttt{e}_n)) &= \Delta(\texttt{e}_1), && \text{provided that } \Delta(\texttt{e}_1) = \ldots = \Delta(\texttt{e}_n) \\ \Delta(\texttt{pre}(\texttt{e})) &= \Delta(\texttt{e}) \\ \Delta(\texttt{e}_1\ \texttt{->}\ \texttt{e}_2) &= \Delta(\texttt{e}_1), && \text{provided that } \Delta(\texttt{e}_1) = \Delta(\texttt{e}_2) \\ \Delta(\texttt{e}_1\ \texttt{when}\ \texttt{e}_2) &= \texttt{e}_2 :: \Delta(\texttt{e}_1), && \text{provided that } \Delta(\texttt{e}_1) = \Delta(\texttt{e}_2) \\ \Delta(\texttt{current}(\texttt{e})) &= \mathit{tl}(\Delta(\texttt{e}))\end{aligned}$$

and let

$$!\mathsf{e} \lessdot\!\!\prec !\mathsf{e}' \Leftrightarrow \Delta(\mathsf{e}) = \Delta(\mathsf{e}')$$

lifting the frequency operator ! to the syntactic level.

Lemma 4. *Let $\mathcal{E}$ and $\mathcal{E}'$ be the valued synchronous automata obtained by translating the expressions* e *and* e*'. Then*

$$!\mathcal{E} \prec !\mathcal{E}' \text{ if } !\mathsf{e} \lessdot\!\!\prec \mathsf{e}'.$$

The frequency analysis can be refined if flow variables are substituted by the expressions defining their frequency.

5.3 A More Liberal Policy

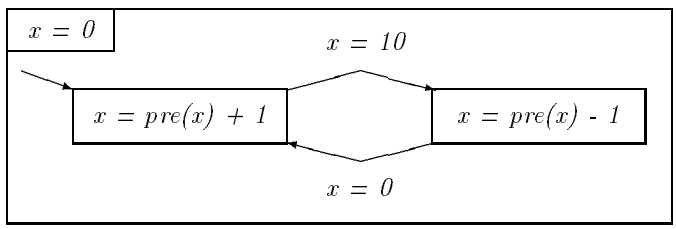
The requirement that equalities of flows are strong may be too rigid a requirement. A flow definition such as

$$x = 1 \ when \ \phi$$
$$x = 2 \ when \ \neg\phi$$

is well defined since at each instant of time only one definition applies. In terms of the flow model (cf. Section 2.1) this means that each such equation generates a flow $!x_j$ indexed by the respective equation and that

$$!x = \bigcup !x_j$$

specifies the frequency of x with the proviso that the frequencies $!x_j$ are pairwise disjoint. An application of this idea can be for instance found in [10] where different flow equation operate in different states of an ARGOS automaton:



6 Finally - the Combination

6.1 Adding Control to Equations

The synchronous automata generated by the declarative code does not provide any mechanism to preempt and to (re-) initialize a computation. In more technical terms, it does not satisfy the control axiom which is pivotal for controlling computations. As a brute force solution, we just add a control register to each equation

$$(x = \epsilon)^{\alpha\ \beta\ \tau} = \mathcal{E} \otimes halt^h \vee (\chi \mathrel{<=} !x \wedge !\epsilon \wedge (\alpha \vee \beta \wedge h)) \vee (\textbf{\textit{if}}\ \chi\ \textbf{\textit{then}}\ x := \epsilon)$$

Whenever started, the equation is evaluated until the control register is preempted. This is inefficient in terms of the number of control registers added, but effective. In a more efficient translation, equations share such a control register, in a pure declarative program only one such register is needed, the one relates to the base frequency. One should note though that now *equations and imperative statements may be freely mixed*, e.g. a list $\langle eq_1, \ldots, eq_n \rangle$ of equation translates to $eq_1^{\alpha\ \beta\ \tau} \otimes \ldots \otimes eq_1^{\alpha\ \beta\ \tau}$.

6.2 A (Re-)View of Data

We distinguish three kinds of variable entities: *valued signals*, for short *signals*, *flows*, and *program variables*, for short *variables*. The latter have not been discussed yet. There are subtle differences which are probably best explained with regard to our implementation model.

If a valued signal s is emitted several times at the same instance, non-determinism may arise in that different values may be assigned to $?s$. This non-determinism can be resolved:

- by adding an associative *combine* operator, e.g. addition in case of integers, which operates on all the values $v_1, \ldots, v_n$ emitted at the same instant in that the assignment $?s := v_1 + \ldots + v_n$ takes place, or
- by *scheduling* the emittances.

The strategy of ESTEREL is to have :

- *valued signals* which may have a combine operator, but otherwise non-deterministic behaviour is (should be) rejected, and
- *program variables* which are scheduled (according to the program structure), which are never present, but always has a value (which is the standard notion of a program variable).

E.g. the ESTEREL code

```
x := 1 ; x := x + 1
```

with `x` being a program variable is evaluated at the same instant, scheduled sequentially and yields a result 2 for `x`. The statement

```
x := 1 || x := x + 1
```

is rejected because of non-determinism. Similarly,

```
emit s(1) || emit s(?s + 1)
```

will be rejected if not, e.g., addition is a combine operator for `s`. The scheduling strategy of valued signals as well as that of flows is that all "writes" at any instant should precede all "reads". This rules out a construction such as

```
emit s(1) ; emit s(?s + 1).
```

Our semantic modeling does not cover scheduling which can be extended by adding scheduling information on the level of actions, at the expense of definitions being more cumbersome. Causality analysis has to take care of this flow of data as well as of the flow of control.

The difference between flows and valued signals is more subtle. Of course, flows are ''type-checked" in that frequencies are constrained while this does not apply to valued signals. However, this is not the essential difference since we may assume that valued signals run at (some relative) base frequency. The more serious problem is due to the fact that declarative code usually assumes that every variable has a unique equational definition, otherwise, interferences of definitions may occur: consider the equation

$$x = 0 \texttt{ -> } 1 + \texttt{pre}(x)$$

which counts the number of instants. If a second equation

$$x = 1$$

is computed in parallel the behaviour is either non-deterministic, and will be rejected which would be the same for valued signals. Applying a combine operator, e.g. +, would, however, result in unacceptable behaviour from the declarative point of view in that uniqueness of definition (at a instant) is violated. Hence, a combine operator should not be related to flows.

All the properties are summarized in the table

x	signal	flow	variable
presence	x		
value	$?x$	x	x
clock		$\surd$	
combine	$\surd$		
schedule	r $\prec$ w	r $\prec$ w	$\surd$

where x is a name of the respective entity.

What are the consequences?

- Either we consider a unified data model as we did in [6] introducing the language *LEA* as a combination of Lustre, Esterel, and Argos. based on a unified data model,
- or we accept the differences of the data models but are liberal in the way the concepts interoperate as we do in this paper using coercions.

There are various possible coercions, and it is a matter of a language designer to make these explicit or not:

- a valued signal or a program variable x can be coerced into a flow by providing some frequency, e.g. by overloading the downsampling operator: `x when e`.

– a flow x cannot turned into a valued signal because uniqueness of definition may be lost due to a combine operator. We propose to overload the notation in that the pair $!x$, $?x$ is of kind *signal*; $!x$ refers to the clock of x, and $?x$ to the value. However, to preserve uniqueness of definition we translate to

$$\langle \epsilon, \textbf{\textit{if}}\ !x\ \textbf{\textit{then}}\ \epsilon := ?x \rangle$$

where ϵ is a new memory cell.

The hinge between components of different nature are the declarations of input and output parameters. Their kind should be specified by the keywords `signal`, `flow`, and `var`, e.g.

```
node raising_edge (flow x:bool)(signal y:bool);
flow z:bool;
let
   z = false -> x and not pre(x)
 ||
   if !z then emit y(?z)
tel
```

Then a component call, e.g.,

```
raising_edge (x' when true) (y')
```

is well defined where `x'` and `y'` are signals. We propose

```
node raising_edge (flow x:bool)(signal_of_flow y:bool);
let
   y = false -> x and not pre(x)
tel
```

as a shorthand notation to the same effect.

7 Conclusion

We have presented a unified view of declarative and control-based synchronous programming based on very few, semantically meaningful operators on synchronous automata. We claim no originality with regard to the notion of synchronous automata which are closely related to Berry's hardware interpretation [2], though our more algebraic approach, first described in [12] and later extended in [8] (which we never bothered to publish, but were we tried to give, for ourselves, an account of the ideas underlying the SYNCHRONY WORKBENCH) may be mildly interesting, even novel. Of course, the real implementation uses some more shortcuts to increase efficiency of the generated code. The shortcuts do not affect correctness because they are always based on well understood assumptions. Anyway, we start from a semantically well-defined basic scheme which already proved to be reasonably efficient if implemented as is, and which has proved to be extremely versatile. The latter is, in fact, the rationale of the SYNCHRONY WORKBENCH: to have a generic framework for synchronous programming.

References

1. Berry, G., Gonthier, G., The synchronous programming language Esterel: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
2. Berry, G, A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciencies, Indian Academy of Sciences*, 17:95–130, 1992.
3. Budde, R., Sylla, K.H., Objekt-orientierte Echtzeitanwendungen auf Grundlage perfekter Synchronisation, *Object-Spektrum*, Feb. 1995
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D., The synchronous data flow programming language Lustre, *Proceedings of the IEEE*, 79(9):1305–1321, Sept. 1991.
5. Harel, D., STATECHARTS: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8(3)231-274,1987
6. Holenderski, L., Poigné, A., The Multi-Paradigm Synchronous Programming Language *LEA*, submitted, 97
7. Le Guernic, P., Gautier, T., , Le Borgne, M., , Le Maire, C., Programming Real-time Applications with SIGNAL, *Proceedings of the IEEE*, 79(9), Sept. 1991
8. Maffeis, O., Poigné, A., Synchronous Automata for Reactive, Real-time, and Embedded Systems, Arbeitspapiere der GMD 964, Forschungszentrum Informationstechnik GmbH, Jan. 19965. (`http://set.gmd.de/EES/papers/SAforRRES/SAforRRES.html`)
9. Maraninchi,F., Operational and compositional semantics of synchronous automaton compositions, In *Proceedings of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, Springer-Verlag, 550–564, Aug. 1992.
10. Maranchini, F, Y.Rémond, Y., Mode-Automata: About Modes and States in Reactive Systems, Research Report, Verimag, 1997
11. Poigné, A., Maffeis, O. ,Morley, M., Holenderski, L., Budde, R., The Synchronous Approach to Designing Reactive Systems, In: *Formal Methods in System Design*, Kluwer, 1998 (an earlier version can be found at `http://set.gmd.de/EES/papers/SP.ps.gz`)
12. Poigné, A., Holenderski, L., Boolean automata for implementing pure Esterel, Arbeitspapiere der GMD 964, Forschungszentrum Informationstechnik GmbH, 1–47, Dec. 1995. (`http://set.gmd.de/EES/papers/E2BA.ps.gz`)
13. Shiple, T.R., Berry, G., Toutai, H., Constructive Analysis of Cyclic Circuits, In : *Proceedings of the European Design and Testing Conference*, IEEE Computer Society, March 1996
14. The *Synchrony Workbench*, `http://set.gmd.de/EES/synchrony/swb.html`

Compositional Verification of Randomized Distributed Algorithms

Roberto Segala

Dipartimento di Scienze dell'Informazione, Università di Bologna
segala@cs.unibo.it

Abstract. We study compositionality issues for the analysis of randomized distributed algorithms. We identify three forms of compositionality that we call process compositionality, property compositionality, and feature compositionality. Process and property compositionality are widely known in the literature, while feature compositionality, although used extensively, does not appear to be emphasized as much. We show how feature compositionality is important for the analysis of randomized systems.

1 Introduction

It is widely recognized that compositionality is an essential feature to enable the scalability of a formal method. That is, it appears to be practically unfeasible to analyze large systems without being able to decompose them into several subcomponents that can be analyzed separately. Compositionality has received considerable attention in the literature; this volume contains several references to the related results. In this paper we study compositionality issues for concurrent systems that contain probability. More specifically, we focus on the analysis of randomized distributed algorithms. We present our study based on the probabilistic model of [38] and on a non-trivial case study [33] where the randomized consensus algorithm of Aspnes and Herlihy [2] is analyzed.

The study of randomization within concurrent systems is particularly complicated due to the interaction of *nondeterminism*, a typical feature of the theory of concurrency, and *probability*, the result of a random choice. The difficulty of randomization is well known in the literature since we can find claims like "intuition often fails to grasp the full intricacy of the algorithm" [31], or "proofs of correctness for probabilistic distributed systems are extremely slippery" [24]. The claims above are further supported by the recent discovery of some problems on known randomized algorithms (e.g., [36, 21, 1]).

In our study of randomized algorithms we have identified three forms of compositionality that we think are important.

- Compositionality of processes.

  This is the typical use of the term "compositionality". That is, we study the properties of a system by subdividing it into several components, studying the

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 515-540
Springer-Verlag Berlin Heidelberg 1998

properties of each component separately, and then combining the properties of the subcomponents to yield the final result. The properties of the components are sufficiently abstract to hide most of the low level details of the components.

- Compositionality of properties.

  Rather than decomposing a system into several subcomponents, we decompose a property into several simpler properties. Several logics for reasoning about concurrent systems use this form of compositionality.

- Compositionality of features.

  A model for a concurrent system usually includes several paradigms that interfere with each other, e.g., real-time, continuous behavior, probability. The picture is complicated further by the presence of nondeterminism. Rather than studying the system as a whole, we study each paradigm (feature) separately and then combine the results. In this way it is possible to study each feature using its own tools. Being able to separate features simplifies considerably the analysis of an algorithm and reduces the chances of error. In this paper we use *coin lemmas* to separate probability from nondeterminism.

We start by introducing a formal model for the description of randomized distributed algorithms [38]. To enable feature compositionality the model is an extension of *Labeled Transition Systems* (LTSs) [20], since there is an extensive literature on the analysis of LTSs that can be adapted to the probabilistic case. In our extension of LTSs we use the synchronization mechanism of CSP [18], where processes synchronize on common actions and evolve independently on others. Our choice of the CSP synchronization style derives from the fact that it allows us to model easily distributed algorithms. We call our probabilistic LTSs *Probabilistic Automata*. Some important properties of probabilistic automata are the following.

- An ordinary LTS is a special case of a probabilistic automaton.
- The main properties that enable compositional reasoning, e.g., projections of executions, are preserved.

Once probabilistic automata are defined, we introduce a generic notion of a *complexity measure* and the related notion of *expected complexity* of an algorithm. We show how it is possible to lift a property of complexity measures to a property of expected complexities as an example of feature compositionality.

We introduce *progress statements* [25, 38], a probabilistic generalization of the *leadsto* operator of UNITY [6], to illustrate how a complex property can be decomposed into simpler properties (property compositionality) and to illustrate a technique to derive expected complexity bounds for an algorithm (feature compositionality).

Finally, we introduce *coin lemmas* [25, 38] to illustrate our main technique to separate probability from nondeterminism. Coin lemmas are a formal expression of the intuition that a randomized algorithm behaves correctly whenever some specific

random draws give some specific results. Coin lemmas provide us with a technique to reduce the analysis of a randomized system to the analysis of an ordinary LTS.

As our main running example we use a large case study [33] where the randomized consensus algorithm of Aspnes and Herlihy [2] is shown to terminate within polynomial time. The algorithm of Aspnes and Herlihy is particularly interesting for its high non-triviality. Its nondeterministic behavior is very complicate, and its probabilistic behavior is based on the theory of Random Walks [13]. In this case study all the forms of compositionality that we introduce are used.

The rest of the paper is organized as follows. Section 2 introduces the model [38] that we use for the analysis of randomized distributed algorithms; Section 3 introduces complexity measures [38, 32] and shows how to study the expected complexity of an algorithm; Section 4 introduces progress statements [38, 25, 32] for the study of the partial progress of an algorithm; Section 5 describes the algorithm of Aspnes and Herlihy, our running example; Section 6 shows how progress statements and projections can be used to reason compositionally about an algorithm; Section 7 describes the main probabilistic component of the algorithm of Aspnes and Herlihy, and Section 8 shows how to use coin lemmas to reason compositionally about the probabilistic behavior of an algorithm; Section 9 gives a hint on how to use refinements [27] to reason about randomized algorithms; Section 10 analyzes the time complexity of the algorithm of Aspnes and Herlihy and gives examples of how to reason compositionally using complexity measures; finally, Section 11 gives references to related work, and Section 12 gives some concluding remarks.

2 Probabilistic Automata

In this section we introduce probabilistic automata by enriching the probabilistic automata of [38] with an input/output distinction. The input/output distinction is useful to define some meaningful fairness conditions; however, the properties that we describe are valid even without such distinction.

2.1 Probability Spaces

A *probability space* $\mathcal{P}$ is a triplet $(\Omega, \mathcal{F}, P)$ where Ω is a set, $\mathcal{F}$ is a collection of subsets of Ω that is closed under complement and countable union and such that $\Omega \in \mathcal{F}$, also called a *σ-field*, and P is a function from $\mathcal{F}$ to $[0, 1]$ such that $P[\Omega] = 1$ and such that for any collection $\{C_i\}_i$ of at most countably many pairwise disjoint elements of $\mathcal{F}$, $P[\cup_i C_i] = \sum_i P[C_i]$.

A probability space $(\Omega, \mathcal{F}, P)$ is *discrete* if $\mathcal{F} = 2^\Omega$ and for each $C \subseteq \Omega$, $P[C] = \sum_{x \in C} P[\{x\}]$. For any arbitrary set X, let $Probs(X)$ denote the set of discrete probability spaces $(\Omega, \mathcal{F}, P)$ where $\Omega \subseteq X$, and such that all the elements of Ω have a non-zero probability.

2.2 Probabilistic Automata

An *I/O automaton* A consists of five components:

- a set $States(A)$ of states;
- a non-empty set $Start(A) \subseteq States(A)$ of start states;
- an action signature $Sig(A) = (in(A), out(A), int(A))$, where $in(A)$, $out(A)$ and $int(A)$ are disjoint sets of input, output, and internal actions, respectively;
- a transition relation $Trans(A) \subseteq States(A) \times Actions(A) \times States(A)$, where $Actions(A)$ denotes the set $in(A) \cup out(A) \cup int(A)$, such that for each state s of $States(A)$ and each input action a of $in(A)$ there is a state s' such that $(s, a, s') \in Trans(A)$;
- a task partition $Tasks(A)$, which is an equivalence relation on $int(A) \cup out(A)$ that has at most countably many equivalence classes. The elements of $Trans(A)$ are called *transitions*, and A is said to be *input enabled*. An equivalence class of $Tasks(A)$ is called a *task* of A.

A *probabilistic I/O automaton* M differs from an I/O automaton in its transition relation. That is, $Trans(M) \subseteq States(M) \times Actions(M) \times Probs(States(M))$. In the rest of the paper we refer to (probabilistic) I/O automata as (probabilistic) automata. Observe that an automaton is a special case of a probabilistic automaton.

Probabilistic automata are partially captured by the reactive model of [16] in the sense that the reactive model assumes some form of nondeterminism between different actions. However, the reactive model does not allow nondeterministic choices between transitions involving the same action. By restricting simple probabilistic automata to have finitely many states, we obtain objects with a structure similar to that of the Concurrent Labeled Markov Chains of [17]; however, in our model we do not need to distinguish between nondeterministic and probabilistic states. In our model nondeterminism is obtained by means of the structure of the transition relation. This allows us to retain most of the traditional notation that is used for automata.

2.3 Executions

A state s of M is said to *enable* a transition if there is a transition $(s, a, \mathcal{P})$ in $Trans(M)$. An action a is said to be enabled from a state s of M if s enables a transition with action a.

An *execution fragment* of M is a sequence α of alternating states and actions of M starting with a state, and, if α is finite ending with a state, $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$, such that for each $i \geq 0$ there exists a transition $(s_i, a_{i+1}, \mathcal{P})$ of M such that $s_{i+1} \in \Omega$. Denote by $fstate(\alpha)$ the first state of α and, if α is finite, denote by $lstate(\alpha)$ the last state of α. An *execution* is an execution fragment whose first state is a start state.

An execution fragment α is said to be *fair* iff the following conditions hold for every task T of M:

1. if α is finite then no action from T is enabled in $lstate(\alpha)$;
2. if α is infinite, then either actions from T occur infinitely many times in α, or α contains infinitely many occurrences of states from which no action from T is enabled.

A state s of M is *reachable* if there exists a finite execution of M that ends in s. A finite execution fragment $\alpha_1 = s_0 a_1 s_1 \cdots a_n s_n$ of M and an execution fragment $\alpha_2 = s_n a_{n+1} s_{n+1} \cdots$ of M can be *concatenated*. The concatenation, written $\alpha_1 \frown \alpha_2$, is the execution fragment $s_0 a_1 s_1 \cdots a_n s_n a_{n+1} s_{n+1} \cdots$. An execution fragment α_1 of M is a *prefix* of an execution fragment α_2 of M, written $\alpha_1 \leq \alpha_2$, iff either $\alpha_1 = \alpha_2$ or α_1 is finite and there exists an execution fragment α_1' of M such that $\alpha_2 = \alpha_1 \frown \alpha_1'$.

2.4 Probabilistic Executions

An execution fragment of M is the result of resolving both the probabilistic and the nondeterministic choices of M. If only the nondeterministic choices are resolved, then we obtain a structure similar to a cycle-free Markov chain, which we call a *probabilistic execution fragment* of M. From the point of view of the study of algorithms, the nondeterminism is resolved by an *adversary* that chooses a transition to schedule based on the past history of the system. A probabilistic execution is the result of the action of some adversary. A probabilistic execution can be thought of as the result of unfolding the transition relation of a probabilistic automaton and then choosing one transition for each state of the unfolding. It has a structure similar to the structure of a probabilistic automaton, where the states are finite execution fragments of M.

Formally, a *probabilistic execution fragment* H of a probabilistic automaton M consists of four components.

- a set of states $States(H) \subseteq frag^*(M)$; let q range over the states of H;
- a signature $Sig(H) = Sig(M)$;
- a singleton set $Start(H) \subseteq States(M)$;
- a transition relation $Trans(H) \subseteq States(H) \times Probs((Actions(H) \times States(H)) \cup \{\delta\})$ such that for each transition $(q, \mathcal{P})$ of H there is a family of transitions of M $\{(lstate(q), a_i, \mathcal{P}_i)\}_{i \geq 0}$ and a family of probabilities $\{p_i\}_{i \geq 0}$ satisfying the following properties: $\sum_{i \geq 0} p_i \leq 1$, $P[\delta] = 1 - \sum_{i \geq 0} p_i$, and for each action a and state s, $P[(a, qas)] = \sum_{i | a_i = a} p_i P_i[s]$.

Furthermore, each state of H is reachable, where reachability is defined analogously to the notion of reachability for probabilistic automata after defining an execution of a probabilistic execution fragment in the obvious way. A *probabilistic execution* H of a probabilistic automaton M is a probabilistic execution fragment of M whose start state is a state of $Start(M)$.

A probabilistic execution is like a probabilistic automaton, except that within a transition it is possible to choose probabilistically over actions as well. Furthermore, a transition may contain a special symbol δ, which corresponds to not scheduling any

transition. In particular, it is possible that from a state q a transition is scheduled only with some probability $p < 1$. In such a case the probability of δ is $1 - p$.

It is possible to define a probability space $\mathcal{P}_H = (\Omega_H, \mathcal{F}_H, P_H)$ associated with H. In particular Ω_H is a set of execution fragments of M (the limit closure of the states of H), $\mathcal{F}_H$ is the smallest σ-field that contains the set of *cones* C_q, consisting of those elements of Ω_H having q as a prefix (let q denote a state of H), and the probability measure P_H is the unique extension of the probability measure defined on cones as follows: $P_H[C_q]$ is the product of the probabilities of each transition of H leading to q. Standard measure theory guarantees that $\mathcal{P}_H$ is well defined. Furthermore, $\mathcal{P}_H$ is sufficiently rich to describe properties like single or multiple occurrences of an action, reachability properties, fairness properties. See [38] for more details.

An *event* E of H is an element of $\mathcal{F}_H$. An event E is called *finitely satisfiable* if it can be expressed as a union of cones. We have chosen the term finitely satisfiable since it is possible to determine that an execution α is in E by looking at a finite prefix of α. A finitely satisfiable event can be represented by a set Θ of incomparable states of H. The event denoted by Θ is $\cup_{q \in \Theta} C_q$. We abuse notation by writing $P_H[\Theta]$ for $P_H[\cup_{q \in \Theta} C_q]$. We call a set of incomparable states of H a *cut* of H, and we say that a cut Θ is *full* if $P_H[\Theta] = 1$.

An important event of $\mathcal{P}_H$ is the set of fair executions of Ω_H. We define a probabilistic execution fragment H to be fair if the set of fair execution fragments has probability 1 in $\mathcal{P}_H$.

2.5 Parallel Composition

Probabilistic automata can be composed in parallel. Due to the reactive structure of probabilistic automata, the definition of parallel composition is simple. The states of the composition are the cross product of the states of the components. The composed probabilistic automata synchronize on their common actions and evolve independently on the others. Whenever a synchronization occurs, the state that is reached is obtained by choosing a state independently for each of the probabilistic automata involved.

Formally, two probabilistic automata M_1 and M_2 are *compatible* iff $int(M_1) \cap acts(M_2) = \emptyset$ and $acts(M_1) \cap int(M_2) = \emptyset$. The *parallel composition* of two compatible probabilistic automata M_1 and M_2, denoted by $M_1 \,||\, M_2$, is the probabilistic automaton M such that

1. $States(M) = States(M_1) \times States(M_2)$.
2. $Start(M) = Start(M_1) \times Start(M_2)$.
3. $in(M) = (in(M_1) \cup in(M_2)) - (out(M_1) \cup out(M_2))$,
   $int(M) = int(M_1) \cup int(M_2)$,
   $out(M) = out(M_1) \cup out(M_2)$,
4. $((s_1, s_2), a, \mathcal{P}) \in Trans(M)$ iff $\mathcal{P} = \mathcal{P}_1 \otimes \mathcal{P}_2$ where
   (a) if $a \in Actions(M_1)$ then $(s_1, a, \mathcal{P}_1) \in Trans(M_1)$, else $\mathcal{P}_1 = \mathcal{U}(s_1)$, and
   (b) if $a \in Actions(M_2)$ then $(s_2, a, \mathcal{P}_2) \in Trans(M_2)$, else $\mathcal{P}_2 = \mathcal{U}(s_2)$,

where $\mathcal{U}(s)$ denotes a probability distribution over a single state s.

In a parallel composition the notion of *projection* is one of the main tools to support compositional reasoning. A projection of an execution fragment α onto a component within a parallel composition is the contribution of the component to obtain α. Formally, let M be $M_1 \parallel M_2$, the parallel composition of M_1 and M_2, and let α be an execution fragment of M. The projection of α onto M_i, denoted by $\alpha \lceil M_i$, is the sequence obtained from α by replacing each state with its i^{th} component and by removing all actions that are not actions of M_i together with their following state. It is the case that $\alpha \lceil M_i$ is an execution fragment of M_i [26].

The notion of projection can be extended to probabilistic executions (cf. Section 4.3 of [38]). Here we do not present the formal definition of projection; rather, we describe the properties of projections that are needed for our analysis, and we refer the reader to [38] for a more detailed description. Given a probabilistic execution fragment H of M, it is possible to define an object $H \lceil M_i$, which is a probabilistic execution fragment of M_i that informally represents the contribution of M_i to H. The states of $H \lceil M_i$ are the projections onto M_i of the states of H. The most important fact is that the probability space associated with $H \lceil M_i$ is the *image space under projection* (cf. Proposition 1) of the probability space associated with H. This property allows us to prove probabilistic properties of H based on probabilistic properties of $H \lceil M_i$ (process compositionality).

Proposition1 [33]. *Let M be $M_1 \parallel M_2$, and let H be a probabilistic execution fragment of M. Let $i \in \{1,2\}$. Then $\Omega_{H \lceil M_i} = \{\alpha \lceil M_i \mid \alpha \in \Omega_H\}$, and for each $\Theta \in \mathcal{F}_{H \lceil M_i}$, $P_{H \lceil M_i}[\Theta] = P_H[\{\alpha \in \Omega_H \mid \alpha \lceil M_i \in \Theta\}]$.* □

3 Complexity Measures

A *complexity function* is a function from execution fragments of M to $\Re^{\geq 0}$. A *complexity measure* is a complexity function ϕ such that, for each pair α_1 and α_2 of execution fragments that can be concatenated, $max(\phi(\alpha_1), \phi(\alpha_2)) \leq \phi(\alpha_1 \frown \alpha_2) \leq \phi(\alpha_1) + \phi(\alpha_2)$.

Informally, a complexity measure is a function that determines the complexity of an execution fragment, where by complexity of an execution fragment we mean something proportional to the amount of work that is neessary to carry out the related operations. A complexity measure satisfies two natural requirements: the complexity of two tasks performed sequentially should not exceed the complexity of performing the two tasks separately and should be at least as large as the complexity of the more complex task; it should not be possible to accomplish more by working less. Examples of complexity measures are the total number of operations performed in a protocol, and the number of operations of some specific type performed in a protocol.

Consider a probabilistic execution fragment H of M and a finitely satisfiable event Θ of $\mathcal{F}_H$. The elements of Θ represent the points where the property denoted

by Θ is satisfied. Let ϕ be a complexity function. Then it is possible to define the expected complexity ϕ to reach Θ in H as

$$E_\phi[H,\Theta] \triangleq \begin{cases} \sum_{q\in\Theta} \phi(q)P_H[C_q] & \text{if } P_H[\Theta] = 1 \\ \infty & \text{otherwise.} \end{cases}$$

The expected complexity $E_\phi[H,\Theta]$ expresses the average amount of work necessary before satisfying the property expressed by Θ. If the probability of Θ is not 1, then the average work could be potentially infinite.

Below we present three compositionality results for complexity measures. Proposition 2 is an instance of feture compositionality, Proposition 3 is an instance of property compositionality, and Proposition 4 is an instance of process compositionality. We give an informal explanation of each result.

If several complexity measures are related by a linear inequality, then their expected values over a full cut are related by the same linear inequality. This result follows from the observation that the function that expresses the complexity of the elements of a full cut is a random variable [13].

Proposition 2. *Let Θ be a full cut of a probabilistic execution fragment H. Let ϕ, ϕ_1, ϕ_2 be complexity functions, and c_1, c_2 two constants such that, for each $\alpha \in \Theta$, $\phi(\alpha) \leq c_1\phi_1(\alpha) + c_2\phi_2(\alpha)$. Then $E_\phi[H,\Theta] \leq c_1E_{\phi_1}[H,\Theta] + c_2E_{\phi_2}[H,\Theta]$.* □

Suppose that within a computation it is possible to identify several phases, each one with its own complexity, and suppose that the complexity associated with each phase remains 0 until the phase starts. Suppose that the expected complexity of each phase is bounded by some constant c. If we know that the expected number of phases that start is bounded by k, then the expected complexity of the system is bounded by ck. In the statement below ϕ_i denotes the complexity associated with phase i and ϕ denotes the number of phases that have started.

Proposition 3. *Let M be a probabilistic automaton. Let $\phi_1, \phi_2, \phi_3, \ldots$ be a countable collection of complexity functions for M, and let ϕ' be a complexity function defined as $\phi'(\alpha) = \sum_{i>0} \phi_i(\alpha)$. Let c be a constant, and suppose that for each fair probabilistic execution fragment H of M, each full cut Θ of H, and each $i > 0$, $E_{\phi_i}[H,\Theta] \leq c$.*

Let H be a probabilistic fair execution fragment of M, and let ϕ be a complexity measure for M. For each $i > 0$, let Θ_i be the set of minimal states q of H such that $\phi(q) \geq i$. Suppose that for each $q \in \Theta_i$, $\phi_i(q) = 0$, and that for each state q of H and each $i > \phi(q)$, $\phi_i(q) = 0$.

Then, for each full cut Θ of H, $E_{\phi'}[H,\Theta] \leq cE_\phi[H,\Theta]$. □

Finally, to verify properties modularly it is useful to derive complexity properties of complex systems based on complexity properties of their components. Proposition 4 below is an example of how to lift an expected complexity bound from a component to the whole composition.

Proposition 4. *Let M be $M_1 \| M_2$, and let $i \in \{1,2\}$. Let ϕ be a complexity function for M, and let ϕ_i be a complexity function for M_i. Suppose that for each finite execution fragment α of M, $\phi(\alpha) = \phi_i(\alpha \lceil M_i)$. Let c be a constant. Suppose that for each probabilistic execution fragment H of M_i and each full cut Θ of H, $E_{\phi_i}[H, \Theta] \leq c$. Then, for each probabilistic execution fragment H of M and each full cut Θ of H, $E_{\phi}[H, \Theta] \leq c$.* □

4 Progress Statements

A progress statement is a predicate that can be used to state reachability properties. It is a probabilistic extension of the leadsto operator of UNITY [6]. The notation for a progress statement is

$$U \xrightarrow[p]{\phi \leq c} U',$$

where U and U' are sets of states, p is a probability, ϕ is a complexity measure, and c is non-negative real number. It states that, no matter how the nondeterminism is resolved, the probability of reaching a state of U' from a state of U within ϕ-complexity c is at least p. In this paper we require fairness for the resolution of nondeterminism; however, the results that we describe below hold also for more general schemas of resolution of the nondeterminism [38].

Given a probabilistic execution fragment H of a probabilistic automaton M, let $e_{U',\phi(c)}(H)$ denote the set of executions α of Ω_H with a prefix α' such that $\phi(\alpha') \leq c$ and $lstate(\alpha') \in U'$. We say that the predicate $U \xrightarrow[p]{\phi \leq c} U'$ is true for M iff for each fair probabilistic execution fragment H of M that starts from a state of U, $P_H[e_{U',\phi(c)}(H)] \geq p$.

Progress statements can be decomposed into simpler statements to be proved separately. Some examples of decompositions are provided by the proposition below.

Proposition 5. *Let M be a probabilistic automaton, $U, U', U'', U''' \subseteq States(M)$, and ϕ be a complexity measure. Then,*

1. *if $U \xrightarrow[p]{\phi \leq c} U'$ and $U' \xrightarrow[p']{\phi \leq c'} U''$, then $U \xrightarrow[pp']{\phi \leq c+c'} U''$;*
2. *if $U \xrightarrow[p]{\phi \leq c} U'$, then $U \cup U'' \xrightarrow[p]{\phi \leq c} U' \cup U''$;*
3. *if $U \xrightarrow[p]{\phi \leq c} U'$ and $U'' \xrightarrow[p']{\phi \leq c'} U'''$, then $U \cup U'' \xrightarrow[min(p,p')]{\phi \leq max(c,c')} U' \cup U'''$.* □

Progress statements can also be used to derive upper bounds on the expected complexity to reach a set of states. Denote by $U \Rightarrow U \, unless \, U'$ the predicate that is true for M iff for every execution fragment sas' of M, $s \in U - U' \Rightarrow s' \in U \cup U'$. Informally, $U \Rightarrow U \, unless \, U'$ means that, once a state from U is reached, M remains in U unless U' is reached. For each probabilistic execution fragment H of M, let $\Theta_{U'}(H)$ denote the set of minimal states of H where a state from U' is reached. The following theorem provides a way of computing the expected ϕ for reaching U'.

Proposition 6 [38]. *Let M be a probabilistic automaton and ϕ be a complexity measure for M. Suppose that for each execution fragment of M of the form sas', $\phi(sas') \leq 1$, that is, each transition of M increases ϕ by at most 1. Let U and U' be sets of states of M. Let H be a probabilistic execution fragment of M that starts from a state of U, and suppose that for each state q of H such that $lstate(q) \in U - U'$ some transition is scheduled with probability 1. Suppose also that $U \xrightarrow[p]{\phi \leq c} U'$ and $U \Rightarrow U$ unless U'. Then, $E_\phi[H, \Theta_{U'}(H)] \leq (c+1)/p$.* □

5 Example: The Algorithm of Aspnes and Herlihy

The algorithm of Aspnes and Herlihy [2] is a randomized algorithm that solves the *consensus* problem within expected polynomial time. The problem consists of letting n processes agree on some value in the set $\{0, 1\}$ so that the properties of *validity*, *agreement*, and *wait-free termination* are satisfied. Validity states that the value chosen by each process should be a value that was proposed in the past by some process; agreement states that no two processes choose different values; wait-free termination states that all non-failed processes eventually decide. Processes may fail by stopping, and the interaction between processes is asynchronous. In other words, it is not possible to distinguish between a slow process and a failed process. It is shown in [14] that there is no algorithm that can solve the consensus problem. Aspnes and Herlihy have shown that by relaxing the wait-free termination property so that the probability of termination is 1 the consensus problem can be solved within expected polynomial time.

The algorithm of Aspnes and Herlihy proceeds in rounds. Every process maintains a variable with two fields, *value* and *round*, that contain the process' current preferred value ($0, 1$ or $\perp$) and current round (a non-negative integer), respectively. We say that a process is at round r if its *round* field is equal to r. The variables (*value*, *round*) are multiple-reader single-writer. Each process starts with its *round* field initialized to 0 and its *value* field initialized to $\perp$.

After receiving the initial value to agree on, each process i executes the following loop. It first reads the (*value*, *round*) variables of all other processes in its local memory. We say that process i is a *leader* if according to its readings its own round is greater than or equal to the rounds of all other processes. We also say that a process i *observed* that another process j is a leader if according to i's readings the round of j is greater than or equal to the rounds of all other processes. If process i at round r discovers that it is a leader, and that according to its readings all processes that are at rounds r and $r-1$ have the same value as i, then i breaks out of the loop and decides on its value. Otherwise, if all processes that i observed to be leaders have the same value v , then i sets its value to v, increments its round and proceeds to the next iteration of the loop. In the remaining case, (leaders that i observed do not agree), i sets its value to $\perp$ and scans again the other processes. If once again the leaders observed by i do not agree, then i determines its new preferred value for the next round by invoking a coin flipping protocol. There is a separate coin flipping protocol for each round.

We represent the main part of the algorithm as an automaton AP (Agreement Protocol) and the coin flipping protocols as probabilistic automata CF_r (Coin Flipper), one for each round r (cf. Figure 1). The coin flipper receives and handles

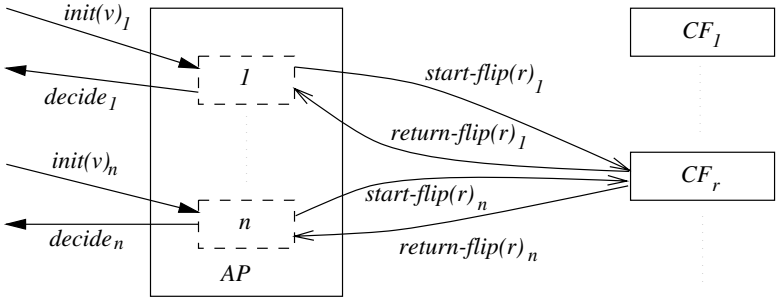


Fig. 1. The decomposition of the algorithm of Aspnes and Herlihy.

requests from each process. We say that a request from a process i is received on *port* i, and we say that a port i is non-failing if process i does not fail. With this decomposition we can analyze several properties just on AP using ordinary techniques for non-probabilistic systems. Indeed, in this section we deal with AP only, and we leave the coin flippers unspecified.

The formal definition of AP is given in Table 1 using the precondition/effect notation that is typical of I/O automata [26]. Beside the shared variables $value(i)$ and $round(i)$, each process has a program counter pc, two arrays *values* and *rounds* containing the scans of the other processes, a set variable *obs* saying what processes have been observed, a variable *start* holding the initial preferred value, and two variables *decided*, and *stopped* stating whether the process has decided or failed. We explain some of the relevant predicates: *obs-leader*(j) is true if i observes that j is a leader; *obs-agree*(r, v) is true if the observations of all the processes whose round is at least r agree on v; *obs-leader-agree*(v) is true if i observes that the leaders agree on a value v; *obs-leader-value* is the value of one of the leaders observed by i. We say that a process is *active* if it is attempting to agree on a value. An active process becomes inactive either by deciding a value or by failing.

6 Compositionality Using Projections and Progress Statements

The validity and agreement properties of the algorithm of Aspnes and Herlihy do not depend on any probabilistic assumption. These are *safety* properties and can be studied solely on AP by means of ordinary invariants. Informally, the invariant for validity states that no process will ever prefer a value different from its initial value if all processes have the same initial value, while the invariant for agreement states that if a process i that is at round r is "about to decide" on some value v, then every

Actions and transitions of process i.

input $init(v)_i$
Eff: $start \leftarrow v$

output $start(v)_i$
Pre: $pc = init \wedge start = v \neq \bot$
Eff: $value(i) \leftarrow v$
$round(i) \leftarrow 1$
$obs \leftarrow \emptyset$
$pc \leftarrow read1$

output $read1(k)_i$
Pre: $pc = read1$
$k \notin obs$
Eff: $values[k] \leftarrow value(k)$
$rounds[k] \leftarrow round(k)$
$obs \leftarrow obs \cup \{k\}$
if $obs = \{1, \ldots, n\}$ then $pc \leftarrow check1$

output $check1_i$
Pre: $pc = check1$
Eff: if $\exists_{v \in \{0,1\}}\, obs\text{-}agree(rounds[i] - 1, v) \wedge$
$obs\text{-}leader(i)$ then
$pc \leftarrow decide$
elseif $\exists_{v \in \{0,1\}}\, obs\text{-}leader\text{-}agree(v)$ then
$value(i) \leftarrow obs\text{-}leader\text{-}value$
$round(i) \leftarrow rounds[i] + 1$
$obs \leftarrow \emptyset$
$pc \leftarrow read1$
else
$value(i) \leftarrow \bot$
$obs \leftarrow \emptyset$
$pc \leftarrow read2$

output $decide(v)_i$
Pre: $pc = decide \wedge values[i] = v$
Eff: $decided \leftarrow true$
$pc \leftarrow nil$

output $read2(k)_i$
Pre: $pc = read2$
$k \notin obs$
Eff: $values[k] \leftarrow value(k)$
$rounds[k] \leftarrow round(k)$
$obs \leftarrow obs \cup \{k\}$
if $obs = \{1, \ldots, n\}$ then
$pc \leftarrow check2$

output $check2_i$
Pre: $pc = check2$
Eff: if $\exists_{v \in \{0,1\}}\, obs\text{-}leader\text{-}agree(v)$ then
$value(i) \leftarrow obs\text{-}leader\text{-}value$
$round(i) \leftarrow rounds[i] + 1$
$obs \leftarrow \emptyset$
$pc \leftarrow read1$
else
$pc \leftarrow flip$

output $start\text{-}flip(r)_i$
Pre: $pc = flip$
$round(i) = r$
Eff: $pc \leftarrow wait$

input $return\text{-}flip(v, r)_i$
Eff: if $pc = wait \wedge round(i) = r$ then
$value(i) \leftarrow v$
$round(i) \leftarrow rounds[i] + 1$
$obs \leftarrow \emptyset$
$pc \leftarrow read1$

input $stop_i$
Eff: $stopped \leftarrow true$
$pc \leftarrow nil$

Tasks: The locally controlled actions of process i form a single task.

Table 1. The actions and transition relation of AP.

process that is at round r or higher has its value equal to v. Since the verification of validity and agreement is reduced to the analysis of ordinary nondeterministic systems, it is not our interest here to pursue such direction and we refer the reader to [33] for further details.

For the wait-free termination property we prove that the algorithms terminates within an expected constant number of rounds and we relate later the round complexity to the time complexity. That is, we prove the following.

Theorem 7. *The algorithm of Aspnes and Herlihy terminates within a constant expected number of rounds.* □

We use progress statements to derive our result. Define the following sets of states.

$\mathcal{R}$ the set of reachable states of AH such that there is an active process;
$\mathcal{D}$ the set of reachable states of AH such that there is no active process.

Let $\phi_{MaxRound}$ be a complexity measure that counts the number of new rounds visited within an execution fragment, i.e., $\phi_{MaxRound}(\alpha) = lstate(\alpha).max\text{-}round - fstate(\alpha).rmax$, where $s.max\text{-}round$ denotes the highest round number of the processes in state s. Our objective is to show that the progress statement

$$\mathcal{R} \xrightarrow[p]{\phi_{MaxRound} \leq 3} \mathcal{D} \tag{1}$$

is valid for a number p that is independent of n, the number of precesses. Then, using Proposition 6, we can derive from (1) that a state of $\mathcal{D}$ is reached within expected $4/p$ rounds, that is, within a constant expected number of rounds. Note that fairness implies that from every state of $\mathcal{R} - \mathcal{D}$ the probability of scheduling a transition is 1, thus satisfying the condition for the applicability of Proposition 6.

The advantage of using the progress statement (1) is that we are left with a property that can be verified by analyzing a finite number of rounds. However, the analysis of Statement (1) is still too complex. The informal argument to prove Statement (1) argues that either a decision is reached, or eventually some process moves to a new fresh round. Once a new round is reached, we know that no coin has been flipped yet at that round. Furthermore, if all the coins flipped at the new round give the same result, then a decision is reached within two other rounds. In order to reflect the informal argument, we decompose Statement (1) into two parts (property compositionality) and use Proposition 5 to combine them. For $v \in \{0, 1\}$, define the following set of states.

$\mathcal{F}_v$ the set of states of $\mathcal{R}$ where there exists a round r and a process l such that $round(l) = r$, $value(l) = v$, $obs_l = \emptyset$, and for all processes $j \neq l$, $round(j) < r$.

Then,

$$\mathcal{R} \xrightarrow[1]{\phi_{MaxRound} \leq 1} \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{D} \tag{2}$$

and

$$\mathcal{F}_v \xrightarrow[p]{\phi_{MaxRound} \leq 2} \mathcal{D}. \tag{3}$$

In order to show the validity of Statement (2) and (3) we identify two properties of AP and two properties of CF that can be composed together to yield the final result (process compositionality). The properties of AP are the following:

D1 If AH is in a state s of $\mathcal{R}$ and all invocations to the coin flippers on non-failing ports get a response, then a state from $\mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{D}$ is reached within one round.

D2 If AH is in a state s of $\mathcal{F}_v$, all invocations to the coin flippers on non-failing ports get a response, and all invocations to $CF_{s.max\text{-}round}$ get only response v, then a state from $\mathcal{D}$ is reached within two rounds.

The properties of each CF_r are the following.

C1 For each fair probabilistic execution fragment of CF_r that starts with a reachable state of CF_r, the probability that each invocation on a non-failing port gets a response is 1.

C2 For each fair probabilistic execution of CF_r, and each value $v \in \{0, 1\}$, the probability that all invocations on a non-failing port get response v is at least p, $0 < p \leq 1$.

Properties **D1** and **D2** are properties of an ordinary nondeterministic system and can be analyzed by means of existing techniques (invariants and liveness arguments). Therefore, we do not deal with their analysis in this paper. Properties **C1** and **C2** are properties of the coin flippers. Their proofs involve the analysis of a random walk [13] and are postponed to the following sections. We emphasize that, by decomposing the system into AP and CF, the main probabilistic analysis of the algorithm is done on the coin flippers only.

Our final objective for this section is to show how properties **D1**, **D2**, **C1**, and **C2** can be composed to yield Statements (2) and (3). To this purpose we use the results about the projections of a probabilistic execution (cf. Proposition 1).

Proposition 8. *Assuming that properties* **C1** *and* **D1** *are valid, Statement (2) is valid.*

Proof. Let H be a probabilistic execution fragment of AH that starts from a state of $\mathcal{R}$. Let Θ be the set of executions of Ω_H where each invocation to any coin flipper on a non-failing port gets a response. By the definition of projection, the executions of $\Theta \lceil AP$ satisfy the premise of **D1**, and thus in each execution of Θ a state from $\mathcal{F}_1 \cup \mathcal{F}_0 \cup \mathcal{D}$ is reached within one round. Thus, it is sufficient to show that $P_H[\Theta] = 1$. Let, for each $i \geq 1$, Θ_i be the set of executions of Ω_H where each invocation to CF_i on a non-failing port gets a response. Then $\Theta = \cap_{i \geq 1} \Theta_i$. Observe that, by definition, Θ_i is the inverse image under projection of the set of executions

of $\Omega_{H\lceil CF_i}$ where each invocation on a non-failing port gets a response. From **C1**, for each i, $P_{H\lceil CF_i}[\Theta_i\lceil CF_i] = 1$, and thus, by Proposition 1, $P_H[\Theta_i] = 1$. Therefore, $P_H[\Theta] = 1$ since, from probability theory, any countable intersection of probability 1 events has probability 1.

Proposition9. *Assuming that properties* **D1**, **D2**, **C1** *and* **C2** *are valid, Statement (3) is valid.*

Proof. Let H be a probabilistic execution fragment of AH that starts from a state s_0 of $\mathcal{F}_v$, and let $r = s_0.max\text{-}round$. Let Θ be the set of executions of Ω_H where each invocation to any coin flipper on a non-failing port gets a response and where each response of CF_r has value v. By the definition of projection, the executions of $\Theta\lceil AP$ satisfy the premise of **D2**, and thus, by **D2**, in each execution of Θ a state from $\mathcal{D}$ is reached within two rounds. Thus, it is sufficient to show that $P_H[\Theta] \geq p$. Let, for each $i \geq 1$, Θ_i be the set of executions of Ω_H where each invocation to CF_i on a non-failing port gets a response. Furthermore, let Θ'_r be the set of executions of Ω_H where no response of CF_r has value $\bar{v}$. Then, $\Theta = (\cap_{i\geq 1}\Theta_i) \cap \Theta'_r$. Observe that, by definition, Θ_i is the inverse image under projection of the set of executions of $\Omega_{H\lceil CF_i}$ where each invocation on a non-failing port gets a response, and Θ'_r is the inverse image under projection of the set of executions of $\Omega_{H\lceil CF_r}$ where each response has value v. From **C1**, for each i, $P_{H\lceil CF_i}[\Theta_i\lceil CF_i] = 1$, and thus, by Proposition 1, $P_H[\Theta_i] = 1$. Since $s_0 \in \mathcal{F}_v$ and $r = s_0.max\text{-}round$, $H\lceil CF_r$ is a probabilistic execution of CF_r (the start state of $H\lceil CF_r$ is a start state of CF_r), and thus property **C2** can be applied. From **C2**, $P_{H\lceil CF_r}[\Theta'_r\lceil CF_r] \geq p$, and thus, by Proposition 1, $P_H[\Theta'_r] \geq p$. Therefore, $P_H[\Theta] \geq p$ since any countable intersection of probability 1 events has probability 1 and the intersection of a probability 1 event with an event with probability p has probability at least p.

7 Example: a Coin Flipping Protocol

The algorithm of Aspnes and Herlihy relies on a coin flipper that satisfies the properties **C1** and **C2** mentioned earlier. The algorithm for the coin flipper is given in terms of n processes that interact through a centralized multiple-write single-read counter. Each of the n processes works as follows: once a request for a flip is received, it reads the value of the counter to check whether it is beyond one of the barriers Kn or $-Kn$, where K is a fixed constant. If the counter is above Kn, then the process returns value 1; if the counter is below $-Kn$, then the process returns 0; otherwise, the process first flips a fair coin to decide whether to increment or decrement the value of the counter, and then starts again.

If after each coin flip we look at the difference between the heads and tails obtained so far, we observe that this number increases/decreases with probability 1/2 at each step. This process is called a *random walk*. By looking at the structure of the coin flipping protocol, we observe that the value of the shared counter may differ by at most n from the current value of the difference between heads and tails.

Therefore, as soon as the difference between heads and tails goes beyond one of the barriers $\pm(K+1)n$, all the processes return a value. From random walk theory this property holds with probability 1. Furthermore, if the barrier $(K+1)n$ is reached before the barrier $-(K-1)n$, then all the processes return 1. From random walk theory this property holds with probability $(K-1)/2K$, our p in property **C2**. A symmetric argument holds for the case where all processes return 0.

So far we have argued informally that the coin flipping protocol works correctly since it behaves like a random walk. However, how can we be sure that there is really a random walk going on in the algorithm? Is there any way that the nondeterminism can affect the randomized process we have identified? For example, how can we guarantee that the scheduler cannot prevent a process from flipping whenever one of the barriers $\pm Kn$ is too close? Indeed, the scheduler can prevent processes from flipping; fortunately, this situation occurs only if all the processes that are flipping fail. The main problem here is that the above argument did not appear in our informal analysis, and the absence of such an argument could be the source of errors in general (cf. [36] for an example).

Our approach to this problem is to provide general theorems that separate the probabilistic argument (properties of the random walk) from the nondeterministic argument (how the scheduler can affect the random walk), so that each problem can be analyzed in its own field. We call such results *coin lemmas* [38, 25, 32], which are an instance of feature compositionality.

8 Compositionality Using Coin Lemmas

A useful technique to prove the validity of a probabilistic property for a probabilistic automaton M is the following [32]:

1. choose a set of random draws that may occur within a probabilistic execution of M, and choose some of the possible outcomes;
2. show that, no matter how the nondeterminism is resolved, the chosen random draws give the chosen outcomes with some minimum probability p;
3. show that whenever the chosen random draws give the chosen outcome, a state from U' is reached within c units of complexity ϕ.

The first two steps can be carried out using the so-called *coin lemmas* [25, 32, 38], which provide rules to map a stochastic process onto a probabilistic execution and lower bounds on the probability of the mapped events based on the properties of the given stochastic process; the third step concerns non-probabilistic properties and can be carried out by means of any known technique for non-probabilistic systems. Coin lemmas are essentially a way of reducing the analysis of a probabilistic property to the analysis of an ordinary nondeterministic property. We refer the reader to [38] for several examples of coin lemmas. Here we illustrate a coin lemma for symmetric random walks.

8.1 A Coin Lemma for Random Walks

Roughly speaking, a random walk is a process that describes the moves of a particle on the real line, where at each time the particle moves in one direction with probability p and in the opposite direction with probability $(1-p)$. In this section we present a coin lemma for symmetric random walks. That is, we present a rule for choosing events within a probabilistic execution fragment that are guaranteed to have properties similar to the properties of random walks where $p = 1/2$. A more general result is given in [33].

Let M be a probabilistic automaton and let $Acts = \{flip_1, \ldots, flip_n\}$ be a subset of $Actions(M)$. Let $\mathbf{S} = \{(U_1^h, U_1^t), (U_2^h, U_2^t), \ldots, (U_n^h, U_n^t)\}$ be a set of pairs where for each $i, 1 \leq i \leq n$, U_i^h, U_i^t are disjoint subsets of $States(M)$ such that for every transition $(s, flip_i, \mathcal{P})$ with an action $flip_i$, $\Omega \subseteq U_i^h \cup U_i^t$, and $P[U_i^h] = P[U_i^t] = 1/2$. The actions from $Acts$ represent coin flips, and the sets of states U_i^h and U_i^t represent the two possible outcomes. Given a finite execution fragment α of M, let $Diff_{Acts,\mathbf{S}}(\alpha)$ denote the difference between the heads and the tails that occur in H. Let z, B, and T be natural numbers, and let $B < T$. The value of z denotes the starting point of the particle, while B and T denote *barriers* in the real line. For each finite execution fragment α, let $z + Diff(\alpha)$ denote the position of the particle after the occurrence of α. For each probabilistic execution fragment H of M, let $\mathbf{Top}[B, T, z](H)$ be the set of executions α of Ω_H such that either the particle reaches the top barrier T before the bottom barrier B, or the total number of "flips" is finite and the particle reaches neither barrier. Define the symmetric event $\mathbf{Bot}[B, T, z](H)$, which is the same as **Top** except that the bottom barrier B should be reached before the top barrier T. Finally, define the event $\mathbf{Either}[B, T, z](H)$ as $\mathbf{Top}[B, T, z](H) \cup \mathbf{Bot}[B, T, z](H)$, which excludes those executions of M where infinitely many "flips" occur and the particle reaches neither barrier.

Proposition 10. *Let H be a probabilistic execution fragment of M, and let $B \leq z \leq T$. Then*

1. $P_H[\mathbf{Top}[B, T, z](H)] \geq (z - B)/(T - B)$.
2. $P_H[\mathbf{Bot}[B, T, z](H)] \geq (T - z)/(T - B)$.
3. $P_H[\mathbf{Either}[B, T, z](H)] = 1$. □

Therefore, we know lower bounds on the probability of the events expressed by **Top**, **Bot**, and **Either**, which are closely connected with our informal argument of correctness for the coin flipping protocol. Note that the events contain executions where finitely many coins are flipped and no barrier is reached. During the analysis of the events (with no probability involved) these executions appear, and therefore we are forced to analyze the case where the scheduler prevents the protocol from reaching one of the barriers.

We conclude with a result about the expected complexity of a random walk. Let $\phi_{Acts}(\alpha)$ be the complexity measure that counts the number of actions from $Acts$

that occur in α. Define $\phi_{Acts,B,T,z}$ to be the truncation of ϕ_{Acts} at the point where one of the barriers B and T is reached. Then we can prove an upper bound on the number of expected flip actions that occur before reaching one of the barriers.

Proposition 11. *Let H be a probabilistic execution fragment of M, and let Θ be a full cut of H. Let $B \leq z \leq T$. Then, $E_{\phi_{Acts,B,T,z}}[H, \Theta] \leq -z^2 + (B+T)z - BT$.* □

8.2 Analysis of the Coin Flipping Protocol

We build a coin flipping protocol that satisfies **C1** and **C2** with $p = (K-1)/2K$. The protocol is based on random walks. We define the protocol by letting a probabilistic automaton DCN_r (Distributed CoiN) interact with a non-probabilistic counter CT_r (CounTer), that is, $CF_r = DCN_r \parallel CT_r$. In this Section, DCN_r is distributed while CT_r is composed of n processes that receive requests from DCN_r and read/update a single shared variable. In Section 9 we discuss how to decentralize CT_r. Since the protocols for DCN_r and CT_r are the same for any round r, we drop the subscript r from our notation. In DCN each process flips a fair coin to decide whether to increment or decrement the shared counter. Then the process reads the current value of the shared counter by invoking CT, and if the value read is beyond the barrier $-Kn$ $(+Kn)$, where K is a fixed constant, then the process returns 0 (1). The specification of CT states that an increment or decrement operation always completes unless the corresponding process fails, while a read operation is guaranteed to complete only if increments and decrements eventually cease.

For the analysis of the coin flipping protocol we start with part 3. Let *Acts* be $\{flip_1, \ldots, flip_n\}$, and let **S** be $\{(U_1^i, U_1^d), (U_2^i, U_2^d), \ldots, (U_n^i, U_n^d)\}$, where U_j^i is the set of states of CF where process j has just flipped *inc* ($fpc_j = inc$), and U_j^d is the set of states of CF where process j has just flipped *dec* ($fpc_j = dec$). Given a finite execution fragment α of CF, let $\phi_{inc}(\alpha)$ be the number of coin flips in α that give *inc*, and let $\phi_{dec}(\alpha)$ be the number of coin flips in α that give *dec*.

Lemma 12. *Let α be a fair execution of CF, such that $\alpha \in$ **Either**$[-(K+1)n, (K+1)n, 0](H)$ for some probabilistic execution H of CF. Then in α each invocation on a non-failing port gets a response.* □

Lemma 13. *Let α be a fair execution of CF, such that $\alpha \in$ **Top**$[-(K-1)n, (K+1)n, 0](H)$ for some probabilistic execution H of CF. Then in α every invocation on a non-failing port gets response 1.* □

The proofs of Lemmas 12 and 13 follow from simple invariant properties and do not involve probability. The main idea is that the value of the shared counter remains beyond Kn $(-Kn)$ once the barrier $(K+1)n$ $(-(K+1)n)$ is reached. A symmetric argument is valid for **Bottom**$[-(K-1)n, (K+1)n, 0](H)$.

At this point properties **C1** and **C2** can be proved by simple applications of the coin lemma for random walks.

Proposition 14. *The coin flipper CF satisfies* **C1**. *That is, for each fair probabilistic execution fragment of CF that starts with a reachable state of CF, with probability* 1 *each invocation on a non-failing port gets an answer.*

Proof. Let H be a fair probabilistic execution fragment of CF that starts with a reachable state s of CF, and let α be a finite execution of CF such that $lstate(\alpha) = s$. Let $z = \phi_{inc}(\alpha) - \phi_{dec}(\alpha)$. If α' is an execution of the event $\mathbf{Either}[-(K+1)n, (K+1)n, z](H)$, then $\alpha \frown \alpha'$ is an execution of $\mathbf{Either}[-(K-1)n, (K+1)n, 0](H')$ for some fair probabilistic execution H' of CF, and by Lemma 12, every invocation to CF in $\alpha \frown \alpha'$ gets a response, and therefore every invocation to CF in α' gets a response. By Theorem 10, $P_H[\mathbf{Either}[-(K+1)n, (K+1)n, z](H)] = 1$.

Proposition 15. *The coin flipper CF satisfies* **C2** *with* $p = (K+1)/2K$. *That is, fixed* $v \in \{0, 1\}$, *for each fair probabilistic execution of CF, with probability at least* $(K-1)/2K$ *each invocation to CF on a non-failing port returns value* v.

Proof. Assume that $v = 1$; the case for $v = 0$ is symmetric. Let H be a fair probabilistic execution of CF. If α is an execution of $\mathbf{Top}[-(K-1)n, (K+1)n, 0](H)$, then, by Lemma 13, every invocation to CF in α gets response 1. By Theorem 10, $P_H[\mathbf{Top}[-(K-1)n, (K+1)n, 0](H)] \geq (K-1)/2K$.

9 Compositionality Using Refinements

A well known compositional verification technique for ordinary automata is based on the notions of *forward/backward simulation* [27] and *bisimulation* [29]. These notions can be extended to probabilistic automata as well [22, 39]. The simulation method is sound for the notion of *trace inclusion* [27], which can also be used as a notion of implementation. The same is true for probabilistic automata [37, 38], and the algorithm of Aspnes and Herlihy provides again a significative example of how probabilistic simulation relations enable compositional reasoning.

In order for the algorithm of Aspnes and Herlihy to be really wait-free, the counter CT must be distributed among all the processes of a system. The distributed implementation of CT, which we denote by DCT (Distributed CounTer), is presented in [2]. We are not interested in the details of the implementation here since there is no probability involved. It is possible to verify that DCT implements CT by exhibiting a *refinement mapping* [27] from DCT to CT. This part of the proof is simple and does not involve probability. Then we use the fact that an ordinary refinement is a special case of a probabilistic refinement, and the fact that the existence of refinements is preserved by parallel composition to lift to the whole algorithm the refinement from DCT to CT, thus showing that DCT can replace CT in AH.

We emphasize that in the analysis above there is no probability involved. The decomposition of the coin flipping protocol into two parts, the processes that do flip coins and the shared counter, allows us to use probabilistic arguments only in those places where probability is really involved.

10 Time Analysis by Lifting Complexity Measures

In this section we derive an upper bound on the time to reach $\mathcal{D}$ once all processes have some minimum speed. We achieve this result by studying the expected number of *inc* and *dec* events (increments and decrements of the shared counter) that occur within the coin flippers and then converting the new expected bound into a time bound. This is done by studying several properties that express relationships between different complexity measures, and then lifting the results to expectations, again an example of feature compositionality. Again we omit all the details that do not rely on probability and we refer the interested reader to [33].

We change slightly our formal model to handle time. Specifically, we add a component *.now* to the states of all our probabilistic I/O automata, and we add the set of positive real numbers to the input actions of all our probabilistic I/O automata. The *.now* component is a nonnegative real number and describes the current time of an automaton. At the beginning (i.e., in the start states) the current time is 0, and thus the *.now* component is 0. The occurrence of an action d, where d is a positive real number, increments the *.now* component by d and leaves the rest of the state unchanged. Thus, the occurrence of an action d models the fact that d time units are elapsing. The amount of time elapsed since the beginning of an execution is recorded in the *.now* component. Since time-passage actions must synchronize in a parallel composition context, parallel composition ensures that the *.now* components of the components are always equal. Thus, we can abuse notation and talk about the *.now* component of the composition of two automata while we refer to the *.now* component of one of the components. We define a new complexity measure $\phi_t(\alpha)$ as the difference between the *.now* components of the last and first states of α. Informally, ϕ_t measures the time that elapses during an execution. We say that an execution fragment α of a probabilistic automaton M is *well-timed* if each task does not remain enabled for more than one time unit without being performed.

We give some preliminary definitions. Let, for each $r > 0$, DCF_r (Distributed Coin Flipper) denote $DCN_r \,\|\, DCT_r$. Let DAH (Distributed Aspnes-Herlihy) denote $AP \,\|\, (\|_{r \geq 1} DCF_r)$. For an execution fragment α of DCF_r or of DAH, let $\phi_{flip,r}(\alpha)$ be the number of *flip* events of DCF_r that occur in α, and let $\phi_{id,r}(\alpha)$ be the number of *inc* and *dec* events of DCF_r that occur in α. For each execution fragment α of DAH let $\phi_{id}(\alpha)$ be the number of *inc* and *dec* events that occur in α.

We start with some non-probabilistic properties about the new complexity measures. The first result, Lemma 16, provides a linear upper bound on the time it takes for DAH to span a given number of rounds and to flip a given number of coins under the assumption of well-timedness. The next two results state basic properties of the coin flipping protocols. That is, once a barrier $\pm(K+1)n$ is reached, there are at most n other *flip* events, and within any execution fragment of DCF_r the difference between the *inc*, *dec* events and the *flip* events is at most n.

Lemma 16. *Let α be a well-timed execution fragment of DAH, and suppose that all the states of α, with the possible exception of lstate(α) are active, that is, are states of $\mathcal{R}$. Let $R = fstate(\alpha).max\text{-}round$. Then, $\phi_t(\alpha) \leq d_1 n^2 (\phi_{MaxRound}(\alpha) + R) + d_2 n \phi_{id}(\alpha) + d_3 n^2$ for some constants d_1, d_2, and d_3.* □

Lemma 17. *Let $\alpha = \alpha_1 \frown \alpha_2$ be a finite execution of DCF_r, and suppose that $|Diff_{Acts,\mathbf{S}}(\alpha_1)| \geq (K+1)n$. Then $\phi_{flip,r}(\alpha_2) \leq n$.* □

Lemma 18. *Let α be a finite execution fragment of DCF_r. Then, $\phi_{id,r}(\alpha) \leq \phi_{flip,r}(\alpha) + n$.* □

We now deal with probabilistic properties. First, based on our results on random walks and on Lemma 17, we show in Lemma 19 an upper bound on the expected number of coin flips performed by a coin flipper. Then, in Lemma 20 we use Lemma 18 and our results about linear combinations of complexity measures to derive an upper bound on the expected number of increment and decrement operations performed by a coin flipper, and we use our compositionality result about complexity measures to show that the bound is preserved by parallel composition. Finally, in Lemma 21 we use our result about phases of computations to combine Theorem 7 with Lemma 20 and derive an upper bound on the expected number of *inc* and *dec* events performed by the algorithm.

Lemma 19. *Let H be a probabilistic execution fragment of DCF_r that starts from a reachable state, and let Θ be a full cut of H. Then $E_{\phi_{flip,r}}[H,\Theta] \leq (K+1)^2n^2+n$.* □

Proof. Let s be the start state of H, and let α be a finite execution of DCF_r with $s = lstate(\alpha)$. Let $z = \phi_{inc}(\alpha) - \phi_{dec}(\alpha)$. If $|z| \geq (K+1)n$, then, by Lemma 17, for each $q \in \Theta$, $\phi_{flip,r}(q) \leq n$, and thus $E_{\phi_{flip,r}}[H,\Theta] \leq n$. If $|z| < (K+1)n$, then, by Proposition 11, $E_{\phi_{Acts,-(K+1)n,(K+1)n,z}}[H,\Theta] \leq -z^2 + (K+1)^2n^2 \leq (K+1)^2n^2$, that is, the event denoted by Θ is satisfied within expected $(K+1)^2n^2$ *flip* events, truncating the count whenever an absorbing barrier $\pm(K+1)n$ is reached. Once an absorbing barrier is reached, by Lemma 17 there are at most n other flip events. Thus, for each state q of H, $\phi_{flip,r}(q) \leq \phi_{Acts,-(K+1)n,(K+1)n,z}(q) + n$. By Proposition 2, $E_{\phi_{flip,r}}[H,\Theta] \leq (K+1)^2n^2 + n$.

Lemma 20. *Let H be a probabilistic execution fragment of DAH that starts from a reachable state, and let Θ be a full cut of H. Then $E_{\phi_{id,r}}[H,\Theta] \leq (K+1)^2n^2 + 2n$.* □

Proof. By Lemma 18, for each execution fragment of α of CF_r, $\phi_{id,r}(\alpha) \leq \phi_{flip,r}(\alpha) + n$. By Proposition 2, $E_{\phi_{id,r}}[H,\Theta] \leq E_{\phi_{flip,r}}[H,\Theta] + n$. By Lemma 19, $E_{\phi_{flip,r}}[H,\Theta] \leq (K+1)^2n^2 + n$. Thus, $E_{\phi_{id,r}}[H,\Theta] \leq (K+1)^2n^2 + 2n$.

Lemma 21. *Let H be a probabilistic fair execution fragment of DAH with start state s, and let $R = s.max\text{-}round$. Suppose that s is reachable. Let Θ denote the set of minimal states of H where a state from $\mathcal{D}$ is reached. Then $E_{\phi_{id}}[H,\Theta] = O(Rn^2)$.* □

Proof. If $R = 0$, then $\Theta = \{s\}$, and thus $E_{\phi_{id}}[H,\Theta] = 0 = O(Rn^2)$. For the rest of the proof assume that $R > 0$. Given a state q of H, we know that $\phi_{id}(q) =$

$\phi_{id,1}(q) + \cdots + \phi_{id,R}(q) + \phi'(q)$, where $\phi'(q) = \sum_{r>0} \phi_{id,r+R}(q)$. For each $r > 0$, let Θ_r be the set of minimal states q of H such that $\phi_{MaxRound}(q) \geq r$. Then, for each $q \in \Theta_r$, $\phi_{id,r+R}(q) = 0$, and for each state q of H and each $r > \phi_{MaxRound}(q)$, $\phi_{id,r+R}(q) = 0$ (CF_{r+R} does not start until some process reaches round $r+R$). Furthermore, by Lemma 20 and Proposition 4, there is a constant $c = (K+1)^2 n^2 + 2n$ such that for each probabilistic execution fragment H' of M, each full cut Θ' of H', and each $i > 0$, $E_{\phi_{id,i}}[H', \Theta'] \leq c$. Therefore, we are in the conditions to apply Proposition 3: each round is a phase, and the numbers of *inc* and *dec* events that occur within each round are the complexity measures for their corresponding round. Function $\phi_{MaxRound}$ is the measure of how many phases are started. By Proposition 3, $E_{\phi'}[H, \Theta] \leq cE_{\phi_{MaxRound}}[H, \Theta]$. By Theorem 7, $E_{\phi_{MaxRound}}[H, \Theta]$ is bound by a constant (independent of n). Therefore, $E_{\phi'}[H, \Theta] = O(n^2)$. Finally, since for each i, H, and Θ, $E_{\phi_{id,i}}[H, \Theta] = O(n^2)$, by Proposition 2, $E_{\phi_{id}}[H, \Theta] = O(Rn^2) + O(n^2) = O(Rn^2)$.

The main result is just a pasting together of the results obtained so far. An immediate consequence on the algorithm of Aspnes and Herlihy is that, if we know that some initialized process does not fail and that the maximum round is 1, then a decision is reached within expected cubic time.

Theorem 22. *Let H be a probabilistic fair, well-timed execution fragment of DAH with a reachable start state s, and let $R = s.max\text{-}round$. Let Θ denote the set of minimal states of H where a state from $\mathcal{D}$ is reached. Then $E_{\phi_t}[H, \Theta] = O(Rn^3)$.*

Proof. By Lemma 16 and Proposition 2, $E_{\phi_t}[H, \Theta] \leq d_1 n^2 E_{\phi_{MaxRound}}[H, \Theta] + d_1 n^2 R + d_2 n E_{\phi_{id}}[H, \Theta] + d_3 n^2$. Thus, by Theorem 7 and Lemma 21, $E_{\phi_t}[H, \Theta] = O(Rn^3)$. □

11 Related Work

There is an extensive literature on the description and analysis of randomized systems. Objects with the same structure as probabilistic automata were introduced already by Rabin [34], even though with different motivations and objectives.

From the modeling point of view there are several results in process algebras [23, 15, 42, 3, 44, 8, 7, 9, 45, 10, 46, 17, 40], where algebras like CCS [28], CSP [18], and ACP [5] are enriched with probability. Most of the algebras above do not deal with nondeterminism and can be classified into *reactive*, *generative*, and *stratified* according to [16]. Our probabilistic automata are an extension of the reactive model of [16], while our probabilistic executions are an example of a generative process. The algebras of [17, 45] do include nondeterminism. The algebra of [45] is used to study a theory of testing for probabilistic systems; the algebra of [17] is based on the alternating model of [43] and is used mainly to illustrate a new model checking algorithm for probabilistic systems. In the alternating model there is a strict alternation between states that enable only nondeterministic transitions and states that enable a single probabilistic transitions. In our model we avoid the alternation, thus obtaining a structure which is closer to ordinary automata.

Other techniques for the analysis of randomized algorithms are studied in [30, 31, 35, 47, 12, 41]. Most of the work concentrates on properties that hold with probability 1 and that can be verified simply by looking at the topology of a system. In [30] a notion of extreme fairness is introduced, later generalized in [47] under the name of α-fairness. The set of α-fair executions of a system have probability 1; thus, the correctness of a system can be verified just by looking at its α-fair executions. The problem with the study of properties that hold with probability 1 is that it is not easy to study the expected complexity of a system.

In [41] a reasoning in the style of weak preconditions is extended to probabilistic systems using objects called predicate transformers. The method, though, seems to be applicable only to small systems. It would be useful to investigate how methods like those of [41] can be integrated with our way of reasoning about systems. In [12] a different approach to the analysis of a randomized algorithm is presented. An algorithm is viewed as a game between a player called *scheduler*, which tries to degrade performance, and a player called *luck*, which fixes the outcome of some coins trying to improve performance. We say that luck has a winning strategy with k moves if luck can make the algorithm work against any scheduler by fixing the value of at most k coins. In such case it is possible to show that the algorithm works with probability at least $1/2^k$. This approach can be seen as an instance of coin lemmas, where the game is the rule to map the process of flipping k coins onto a probabilistic execution.

12 Concluding Remarks

We have shown how different forms of compositionality can be included into a model for randomized distributed computation and can be used for the analysis of nontrivial randomized distributed algorithms. We have identified three forms of compositionality: process compositionality, property compositionality, and feature compositionality. Process and property compositionality are just two new names for forms of compositionality that are widely known in the literature; feature compositionality, although typically used in mathematics, is a form of compositionality that is not usually considered as compositionality. We have shown how feature compositionality plays a crucial role in simplifying the analysis of a randomized system.

An obvious question is whether we have been lucky in our case study and whether the main idea of separating probability from nondeterminism really works. Although we cannot claim that it is always possible to obtain a clean separation between probability and nondeterminism, our experience with the analysis of randomized algorithms [25, 32, 1, 38] gives us a reasonable confidence that a separation can be obtained. The original choice of studying the algorithm of Aspnes and Herlihy [33] was guided mainly by the idea of looking for an algorithm where such separation appeared to be difficult to achieve. Of course, the main problem in the analysis of a system is to understand how to decompose the system, which is still nontrivial.

Another question concerns the generality of the model. In the interaction between two systems we always assume that the probabilistic choices of each component

are independent. In other words, it is not possible for a process to condition the probability distribution associated with the transitions of another process (our model is not generative). This restriction limits considerably the field of application of our theory. We understand that such limitations exist, and it would be desirable to overcome them. Unfortunately, we do not know of any way of extending the CSP synchronization style to a model that is not reactive, and on the other hand we find such synchronization mechanism very useful for the analysis of distributed algorithms.

Acknowledgments. I would like to thank the organizers of COMPOS'97 for inviting me to the symposium and for the wonderful exchange environment they have provided.

References

1. S. Aggarwal. Time optimal self-stabilizing spanning tree algorithms. Technical Report MIT/LCS/TR-632, MIT Laboratory for Computer Science, 1994. Master's thesis.
2. J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460, September 1990.
3. J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: ACP with generative probabilities. In Cleaveland [11], pages 472–485.
4. J.C.M. Baeten and J.W. Klop, editors. *Proceedings of CONCUR 90,* Amsterdam, volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
5. J.C.M. Baeten and W.P Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.
6. K.M. Chandi and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
7. I. Christoff. Testing equivalences and fully abstract models for probabilistic processes. In Baeten and Klop [4], pages 126–140.
8. I. Christoff. *Testing Equivalences for Probabilistic Processes*. PhD thesis, Department of Computer Science, Uppsala University, 1990.
9. L. Christoff. *Specification and Verification Methods for Probabilistic Processes*. PhD thesis, Department of Computer Science, Uppsala University, 1993.
10. R. Cleaveland, S.A. Smolka, and A. Zwarico. Testing preorders for probabilistic processes (extended abstract). In *Proceedings* 19^{th} *ICALP,* Madrid, volume 623 of *Lecture Notes in Computer Science*, pages 708–719. Springer-Verlag, 1992.
11. W.R. Cleaveland, editor. *Proceedings of CONCUR 92,* Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
12. S. Dolev, A. Israeli, and S. Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, April 1997.
13. W. Feller. *An Introduction to Probability Theory and its Applications. Volume 1.* Jokn Wiley & Sons, Inc., 1950.
14. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with a family of faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
15. A. Giacalone, C.C Jou, and S.A. Smolka. Algebraic reasoning for probabilistic concurrent systems. In *Proceedings of the Working Conference on Programming Concepts and Methods (IFIP TC2),* Sea of Galilee, Israel, 1990.
16. R.J. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1996.

17. H. Hansson. *Time and Probability in Formal Design of Distributed Systems*, volume 1 of *Real-Time Safety Critical Systems*. Elsevier, 1994.
18. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.
19. B. Jonsson and J. Parrow, editors. *Proceedings of CONCUR 94,* Uppsala, Sweden, volume 836 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
20. R. Keller. Formal verification of parallel programs. *Communications of the ACM*, 7(19):561–572, 1976.
21. E. Kushilevitz and M. Rabin. Randomized mutual exclusion algorithms revisited. In *Proceedings of the* 11^{th} *Annual ACM Symposium on Principles of Distributed Computing,* Quebec, Canada, pages 275–284, 1992.
22. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Conference Record of the* 16^{th} *ACM Symposium on Principles of Programming Languages,* Austin, Texas, pages 344–352, 1989.
23. K.G. Larsen and A. Skou. Compositional verification of probabilistic processes. In Cleaveland [11], pages 456–471.
24. D. Lehmann and M. Rabin. On the advantage of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the* 8^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 133–138, January 1981.
25. N.A. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings of the* 13^{th} *Annual ACM Symposium on Principles of Distributed Computing,* Los Angeles, CA, pages 314–323, 1994.
26. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, Canada, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.
27. Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.
28. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
29. D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, 5^{th} *GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.
30. A. Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the* 15^{th} *Annual ACM Symposium on Theory of Computing, Boston, Massachusetts*, May 1983.
31. A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.
32. A. Pogosyants and R. Segala. Formal verification of timed properties of randomized distributed algorithms. In *Proceedings of the* 14^{th} *Annual ACM Symposium on Principles of Distributed Computing,* Ottawa, Ontario, Canada, pages 174–183, August 1995.
33. A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. Technical Memo MIT/LCS/TM-555, MIT Laboratory for Computer Science, 1997.
34. M.O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.
35. J.R. Rao. Reasoning about probabilistic algorithms. In *Proceedings of the* 9^{th} *Annual ACM Symposium on Principles of Distributed Computing,* Quebec, Canada, August 1990.
36. I. Saias. Proving probabilistic correctness: the case of Rabin's algorithm for mutual exclusion. In *Proceedings of the* 11^{th} *Annual ACM Symposium on Principles of Dis-*

tributed Computing, Quebec, Canada, August 1992.
37. R. Segala. A compositional trace-based semantics for probabilistic automata. In I. Lee and S.A. Smolka, editors, *Proceedings of CONCUR 95,* Philadelphia, PA, USA, volume 962 of *Lecture Notes in Computer Science*, pages 234–248. Springer-Verlag, 1995.
38. R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science, 1995. Also appears as technical report MIT/LCS/TR-676.
39. R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
40. K. Seidel. Probabilistic communicating processes. Technical Report PRG-102, Ph.D. Thesis, Programming Research Group, Oxford University Computing Laboratory, 1992.
41. K. Seidel, C. Morgan, and A. McIver. An introduction to probabilistic predicate transformers. Technical Report PRG-TR-6-96, Programming Research Group, Oxford University Computing Laboratory, 1996.
42. C. Tofts. A synchronous calculus of relative frequencies. In Baeten and Klop [4].
43. M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings of 26th IEEE Symposium on Foundations of Computer Science*, pages 327–338, Portland, OR, 1985.
44. S.H. Wu, S. Smolka, and E.W. Stark. Composition and behaviors of probabilistic I/O automata. In Jonsson and Parrow [19].
45. W. Yi and K.G. Larsen. Testing probabilistic and nondeterministic processes. In *Protocol Specification, Testing and Verification XII*, pages 47–61, 1992.
46. S. Yuen, R. Cleaveland, Z. Dayar, and S. Smolka. Fully abstract characterizations of testing preorders for probabilistic processes. In Jonsson and Parrow [19].
47. L. Zuck. *Past Temporal Logic.* PhD thesis, The Weizman Institute of Science, 1986.

Lazy Compositional Verification*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract. Existing methodologies for the verification of concurrent systems are effective for reasoning about global properties of small systems. For large systems, these approaches become expensive both in terms of computational and human effort. A *compositional* verification methodology can reduce the verification effort by allowing global system properties to be derived from local component properties. For this to work, each component must be viewed as an open system interacting with a well-behaved environment. Much of the emphasis in compositional verification has been on the *assume-guarantee* paradigm where component properties are verified contingent on properties that are assumed of the environment. We highlight an alternate paradigm called *lazy composition* where the component properties are proved by composing the component with an abstract environment. We present the main ideas underlying lazy composition along with illustrative examples, and contrast it with the assume-guarantee approach. The main advantage of lazy composition is that the proof that one component meets the expectations of the other components, can be delayed till sufficient detail has been added to the design.

1 Introduction

In the last two decades, there has been considerable progress in the verification of concurrent, reactive systems. Much of the research has been devoted to the development of formalisms such as temporal logics [Eme90, Lam94, MP92, CM88] and

* Supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931 and CCR-9300444. Based on earlier work [Sha93b] funded by Naval Research Laboratory (NRL) under contract N00015-92-C-2177. Connie Heitmeyer, Ralph Jeffords, and Pierre Collette gave useful feeback on the work cited above. John Rushby, Sam Owre, and Nikolaj Bjørner provided detailed comments on drafts of this paper. Presentations of earlier versions of this work at the meetings of IFIP Working Group 2.3 and at COMPOS'97 yielded valuable insights and criticisms. Martín Abadi and Leslie Lamport prompt and helpful in their responses to various technical queries and with feedback on earlier drafts.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 541-564, 1998.
Springer-Verlag Berlin Heidelberg 1998

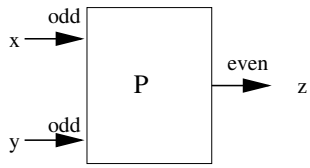


Fig. 1. Even number generator

process algebras [Hoa85, Mil80], and verification methods [Bar85, dBdRR90, dBdRR94, Sha93a] based on deduction [Eme90, Lam94, MP92, CM88] and model checking [CES86, Kur93, Hol91]. While these techniques are effective on small examples—mutual exclusion, basic cache consistency algorithms, and simple communication protocols—the difficult problem of scaling these techniques up to large and realistic systems has remained largely unsolved.

Large-scale concurrent systems are usually defined by composing together a number of components or subsystems. The typical verification methods are non-compositional and require a global examination of the entire system. In the *deductive* approach to verification, this means that a property such as an invariant has to be verified with respect to each transition of all of the components in the system. Verification approaches based on *model checking* also fail to scale up gracefully since the global state space that has to be explored can grow exponentially in the number of components [GL94]. The purpose of a *compositional* verification approach is therefore to shift the burden of verification from the global level to the local, component level so that global properties are established by composing together independently verified component properties.

To motivate compositional verification, we can consider a very simple example of an adder component P shown in Figure 1 that adds two input numbers x and y and places the output in z. Here x, y, and z can be program variables, signals, or latches depending on the chosen model of computation. The system containing P as a component might require its output z to be an even number, but obviously P cannot unconditionally guarantee this property of the output z. It might be reasonable to assume that the environment always provides odd number inputs at x and y, so that with this assumption it is easy to show that the output numbers at z are always even. Only local reasoning in terms of P is needed to establish that z is always even when given odd number inputs at x and y.

If, as is shown in Figure 2, P is now composed with another component Q that generates the inputs at x and y, then to preserve the property that only even numbers are output at z, Q must be shown to output only odd numbers at x and y. However, the demonstration that Q provides only odd numbers as outputs at x and y might require assumptions on the inputs taken by Q, where z itself might be such an input. If in showing that Q produces odd outputs at x and y, one has to assume that the z input is always even, then we have an obvious

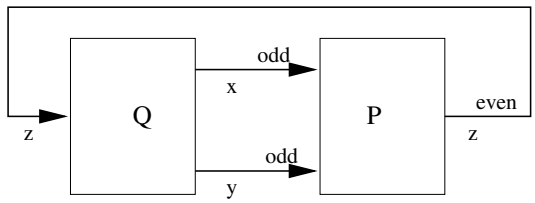


Fig. 2. Odd and even number generators

circularity and nothing can be concluded about the oddness or evenness of x, y, and z. If this circularity can somehow be broken, we then have a form of well-founded mutual recursion between P and Q that admits a proof by simultaneous induction that x and y are always odd and z is always even. The circularity can be broken by noting that that a z output for P is even as long as the preceding x and y inputs are odd, and the x and y outputs for Q are odd as long as the preceding z input is even.

The assume-guarantee paradigm is the best studied approach to compositional verification [AL93, AL95, AP93, CMP94, Col93, Hoo91, Jon83, MC81, PJ91, Pnu84, Sta85, XCC94, XdRH97, Zwi89]. In this approach, a property of a component is stated as a pair (A, C) consisting of a guarantee property C that the component will satisfy provided the environment to the component satisfies the assumption property A. The interpretation of (A, C) has to be carefully defined to be non-circular. Informally, a component P satisfies (A, C) if the environment to P violates A before the component fails to satisfy C. When two or more components, P_1 satisfying (A_1, C_1) and P_2 satisfying (A_2, C_2), are composed into a larger component $P_1 \| P_2$, the assumption A together with property C_1 of component P_1 must be used to show that P_1 does not violate assumption A_2, and correspondingly, A and C_2 must be used to show that P_2 does not violate A_2. Discharging these proof obligations allows one to conclude that the composite component $P_1 \| P_2$ has a similar property (A, C) where C follows from A, C_1, and C_2. The assume-guarantee technique as described informally still suffers from the earlier circularity. The formal details of the assume-guarantee technique are deferred to Section 2. The assume-guarantee approach has been more widely studied than actually used. The primary difficulty in applying this approach for compositional verification is that it requires component guarantee properties to be strong enough to entail any potential environment constraints. It is obviously not easy to anticipate all the potential constraints that might be placed on a component by the other components in a system.

The *lazy composition* approach advocated in this paper builds on conventional techniques while avoiding the difficulties associated with the assume-guarantee approach [Sha93b]. Lazy composition works at the level of the *specification* of component behavior. In lazy composition, a property C of a component specified as P is actually proved of the system $P \| E$ obtained by composing P

with an abstract environment specification E that captures the expected behavior of the environment. When the component specification P is composed with another component specification Q, then C might no longer be a property of the specification $P\|Q$ since Q might not satisfy the constraint E. However, C is a property of the composition $P\|(Q \wedge E)$ obtained by strengthening Q to additionally satisfy E. This allows local properties such as C to be used as global properties of the specification of a larger system. If in fact the combined specification $P\|(Q \wedge E)$ can be simplified to $P\|Q$, then clearly the constraint E is redundant and can be eliminated. However, it is not imperative that (properties guaranteed by) Q already imply E as is the case with the assume-guarantee technique. While the assumed environment specification has eventually to be shown to hold of the other components in the system, this proof obligation can be discharged lazily as the system design is being refined. The demonstration that $P\|(Q \wedge E)$ is refined by $P\|Q$ uses inductive reasoning on computations so that any possible circularity between assumptions E and guarantees C is avoided. Thus lazy composition allows global properties to be proved by local component-wise reasoning combined with a one-time demonstration that each component satisfies the accumulated constraints imposed by the other components. There are several other tradeoffs between lazy composition and assume-guarantee reasoning that are discussed in Section 3.

The lazy composition approach is quite general and can be applied to a wide variety of synchronous and asynchronous computational models, but this paper considers only one such model, namely, asynchronous transition systems with interleaving composition.

We first present some background on compositional verification in Section 2. Lazy composition is introduced in Section 3. Some examples illustrating the use of lazy composition in verifying safety properties are presented in Section 4. The elimination of environment constraints by means of refinement proofs is described in Section 5. The verification of liveness properties using lazy composition is given in Section 6. A comparison between lazy composition and other compositional approaches is given in Section 7.

2 Background

The presentation in this paper is entirely at the semantic level where we are dealing with states, predicates (sets) and relations on states, computations as infinite sequences of states, and properties as sets of computations. We will also speak of sets of sequences and properties interchangeably.

Asynchronous Transition Systems. In its simplest form, an *asynchronous transition system* is a triple $\langle \Sigma; I, N \rangle$ of a state type Σ, an initial set of states I, and a reflexive (stuttering-closed) *next-state* relation N that defines the possible *atomic* actions of the system. Seen as a *closed system*, i.e., one with no interac-

tion with an outside environment,[2] a valid *computation* of such a system consists of an infinite sequence of states σ whose initial state $\sigma(0)$ is in I, i.e., $I(\sigma(0))$ holds, and N holds of each pair of adjacent states, i.e., for all i, $N(\sigma(i), \sigma(i+1))$. A property is a set of infinite state sequences. If P is an asynchronous transition system, the set of its computations in the closed interpretation is represented as $[\![P]\!]$. The transition system P *has a property* A, in symbols, $[\![P]\!] \models A$, iff the set of computations $[\![P]\!]$ is a subset of the set of sequences corresponding to the property A. We write $\models A$ when the property A is valid, i.e., contains all the infinite sequences. Properties (sets of infinite sequences) can be combined with connectives $\neg A$ (complement), $A \vee B$ (union), $A \wedge B$ (intersection), and $A \supset B$ which is defined as $\neg A \vee B$. One transition system P *refines* another transition system Q when $\models [\![P]\!] \supset [\![Q]\!]$. In typical usage below, a transition system will be given as $\langle I, N \rangle$ leaving the state type Σ implicit.

Safety Properties. A safety property informally asserts that nothing bad happens during a computation. Let $\sigma[i]$ represent the finite prefix consisting of the first i states $\sigma(0) \ldots \sigma(i-1)$ of σ. A safety property [AS85] is one that excludes an infinite sequence σ exactly when it excludes all extensions $\sigma[i] \circ \rho$ of some finite prefix $\sigma[i]$ of σ. This means that safety properties are falsified by some finite prefix of a sequence. For any property A, there is a property A^S (the *safety closure* of A) which is the strongest safety property containing A defined as $\{\sigma \mid \forall i : \exists \rho : \sigma[i] \circ \rho \in A\}$. The property (set) A^S is clearly a safety property. If A is a safety property, we say that $\sigma[n] \in A$ when $\sigma[n] \circ \rho \in A$ for some ρ.

Liveness Properties. Liveness properties assert that something good eventually happens during the computation. Such properties hold of some infinite extension of any finite sequence α, i.e, they can always be satisfied by an appropriately chosen sequence of states. A liveness property can exclude an infinite sequence σ but must contain some extension of $\sigma[i]$ for each i. Given a property A, let A^L (the *liveness closure* of A) be $A \vee \neg A^S$, where $\neg A^S$ represents the complement of A^S. Then A^L is a liveness property because if for some α there is no ρ such that $\alpha \circ \rho \in A^L$, then since $A \subseteq A^L$, $\forall \rho : \alpha \circ \rho \notin A$, but then $\forall \rho : \alpha \circ \rho \notin A^S$. This is a contradiction since every infinite sequence must be in A^L or A^S. Thus every property A can be expressed as the conjunction of a safety property A^S and a liveness property A^L [Sch87].

Stuttering Invariance. A set of sequences A is *stuttering invariant* if whenever $\sigma[i+1] \circ \rho \in A$ then $\sigma[i+1] \circ \sigma(i) \circ \rho \in A$. In words, if A contains a sequence, then it contains all variants of this sequence obtained by stuttering individual states in the sequence finitely often. Stuttering arises naturally when there is a notion of an observation of a transition system so that some of the transitions have no observable effect. Stuttering invariance is often imposed as a constraint

[2] The closed interpretation here means that each transition of a valid computation satisfies the next-state relation N leaving no room for any environment transitions other than those already specified by N.

on the allowable properties so that the resulting transition system can always be implemented using internal unobservable state components.

Published explanations of assume-guarantee proof techniques often implicitly rely on stuttering invariance without explicitly mentioning it. Stuttering invariance is needed to argue that if we are given safety properties A and B such that $\sigma[i] \in A$ and $\sigma[i] \in B$, then $\sigma[i] \in A \wedge B$. Such a result is valid if A and B are stuttering invariant properties. To see how the result can fail to hold, let A consist of the strictly increasing sequences of even numbers and B consist of the strictly increasing sequences of prime numbers. Both A and B are safety properties that are not stuttering invariant. The singleton prefix $\langle 2 \rangle$ is in both A and B but $A \wedge B$ is empty.[3]

Expressing Properties. The above notions of computation and property are typical of the use of linear-time temporal logics for stating and proving properties of closed systems. Examples of such logics include

- Manna and Pnueli's LTL [MP92] with the temporal operators $\bigcirc$ (next-time), $\Box$ (always), and $\Diamond$ (eventually). Properties expressed in LTL that use the $\bigcirc$ operator are not necessarily stuttering invariant.
- Chandy and Misra's Unity [CM88] with operators **invariant**, **stable**, **unless**, **until**, and **leadsto** which are applied to state predicates so that temporal formulas are not nested. Unity properties are stuttering invariant.
- Lamport's temporal logic of actions [Lam94] which drops the next-time operator from linear-time temporal logic but allows temporal operators to range over *actions*, i.e., binary relations over states. TLA is designed to admit only stuttering invariant properties.

In the examples below, we restrict ourselves to some simple operators for defining properties. If p is a predicate on states, then

1. **invariant** p holds of σ iff $\forall i : p(\sigma(i))$. This is a safety property.
2. **eventually** p holds of σ iff $\exists i : p(\sigma(i))$. This is a liveness property for any satisfiable predicate p since any finite sequence can be extended to one in which p eventually holds.

For a given transition system $\langle I, N \rangle$, the invariance of p can be proved using induction by showing that for all states s in Σ, $\vdash I(s) \supset p(s)$, and for all states s and s', and $\vdash p(s) \wedge N(s, s') \supset p(s')$.

Components as Open Systems. The next step is to extend the model to open systems so that components can be independently specified and composed to form larger systems. If Σ is the set of global states of the large system, then a component i can be given as a triple $\langle \Sigma; I_i, N_i \rangle$. However, we can no longer take

[3] A weaker requirement than stuttering invariance suffices for the soundness of the assume-guarantee proof reasoning methods. A safety property A must include the infinite sequence $\sigma[i+1] \circ \sigma(i)^\omega$ obtained by infinitely stuttering the last state of any nonempty finite prefix $\sigma[i+1]$ in A.

the closed interpretation since a computation must include the actions taken by other components. In the *open system* interpretation, a computation is an infinite sequence of states whose initial state is in I_i and each pair of adjacent states is either related by N_i or is an arbitrary environment transition. The open system interpretation is much too liberal and does not admit any interesting properties since there are no constraints on the environment actions. This can be partially overcome by placing weak constraints on the environment actions, e.g., the values of the local variables of a component must be left unchanged by its environment. With some constraint on the environment actions, one can actually verify reasonably interesting local properties of a component. For example, in TLA [Lam94], the next-state relation of a component is written as $[N]_f$ which holds of a pair of states s, s' when $N(s,s') \vee f(s') = f(s)$. The state function f typically projects out the local variables of the component so that the environment transitions must not affect the values of these variables. In Lynch and Tuttle's I/O automata [LT87], a component is an input-enabled automaton with its own local state so that any component properties established with respect to this interpretation remain globally valid even in composition with other components.

Even with such restrictions on the environment behavior, the open system interpretation is somewhat weak since many properties of a component can only be proved by assuming a stronger degree of cooperation from the environment. We have already seen the example of the adder component of Figure 1 which can be shown to always output even numbers when given odd number inputs by its environment.

The Owicki–Gries Method. The Owicki–Gries method [OG76] is the first attempt at a component-wise decomposition of the verification problem. In this method, one proves a global invariant of the composition $P_1 \| P_2$ by showing it to be a local invariant of one of the components, say P_1, and a stable predicate, i.e., one that is never falsified, of the other component P_2. In other words, one component establishes the invariant and the other component does not falsify it. This method is not really compositional since it requires global reasoning on all the actions of each component in order to establish an invariant. The Owicki–Gries method was originally proposed in the framework of a proof-outline logic where program components are annotated with assertions. Such program-based proof methods can be quite restrictive when compared to the use of high-level behavioral specifications as given by asynchronous transition systems.

Compositional Verification Using the Assume-Guarantee Approach. The assume-guarantee approach originally proposed by Jones [Jon83] and Misra and Chandy [MC81] is perhaps the most widely studied compositional verification technique for concurrent systems. The presentation of this approach given below is adapted from Abadi, Lamport, and Plotkin [AL93, AL95, AP93] and Collette [Col94]. An assume-guarantee specification of a a component property is given as a pair (A, C) consisting of an assumption property A and a guarantee property C. To capture (A, C) is defined as $A \xrightarrow{+} C$ (A *secures* C) which is the subset

of $A \supset C$ defined as $\{\sigma \in A \supset C \mid \forall i : \sigma[i] \in A^S \supset \sigma[i+1] \in C^S\}$. Thus $A \stackrel{+}{\longrightarrow\!\triangleright} C$ rules out unrealizable implementations of $A \supset C$ that exhibit computations where C^S fails before the failure of A^S can be detected by the component. Similarly, $A \longrightarrow\!\triangleright C$ (A *maintains* C) is the set of σ in $A \supset C$ such that for all i, $\sigma[i] \in A \supset \sigma[i] \in C$. Note that $A \stackrel{+}{\longrightarrow\!\triangleright} C \equiv (A \supset C) \wedge (A^S \stackrel{+}{\longrightarrow\!\triangleright} C^S)$, and $A \longrightarrow\!\triangleright C \equiv (A \supset C) \wedge (A^S \longrightarrow\!\triangleright C^S)$.

Composition of components $P_1 \| P_2$ is defined so that $[\![P_1 \| P_2]\!]$ is the intersection of $[\![P_1]\!]$ and $[\![P_2]\!]$. Since P_1 and P_2 are specified to allow environment transitions, the composition of P_1 and P_2 includes all the interleavings of P_1 and P_2 actions, but also contains computations with simultaneous P_1 and P_2 actions.[4]

The main compositionality rule in the assume-guarantee method [AL95] is stated in Theorem 1.

Theorem 1.

$$\frac{\begin{array}{l} P_i \models A_i \stackrel{+}{\longrightarrow\!\triangleright} C_i, \text{ for } i = 1,2 \\ \models A^S \wedge C_1^S \wedge C_2^S \supset A_1 \wedge A_2 \\ \models A \stackrel{+}{\longrightarrow\!\triangleright} (C_1 \wedge C_2 \longrightarrow\!\triangleright C) \end{array}}{P_1 \| P_2 \models A \stackrel{+}{\longrightarrow\!\triangleright} C.}$$

In words, in order to show that the composition $P_1 \| P_2$ has property $A \stackrel{+}{\longrightarrow\!\triangleright} C$, it suffices to establish the following premises of the compositionality rule:

1. Each P_i has property $A_i \stackrel{+}{\longrightarrow\!\triangleright} C_i$.
2. The individual environment constraints A_1 and A_2 must be satisfied by the conjunction of the safety parts of the joint environment constraint A and the guarantee properties C_1 and C_2.
3. The joint commitment C must be maintained by the individual commitments C_1 and C_2 when secured by the environment assumption A.

The formal details justifying the assume-guarantee rule are fairly elaborate, but we can briefly convey some of the intuition by sketching the soundness argument. It is sufficient to focus our attention on infinite sequences σ such that $\sigma \in (A_1 \stackrel{+}{\longrightarrow\!\triangleright} C_1) \wedge (A_2 \stackrel{+}{\longrightarrow\!\triangleright} C_2)$. To show $\sigma \in A \stackrel{+}{\longrightarrow\!\triangleright} C$, we need to prove both $\sigma \in A \supset C$ and $\sigma \in A^S \stackrel{+}{\longrightarrow\!\triangleright} C^S$. The argument proceeds in three steps:

- $\sigma \in (A^S \stackrel{+}{\longrightarrow\!\triangleright} C_1^S \wedge C_2^S)$.
  That is, for any n, $\sigma[n] \in A^S$ implies $\sigma[n+1] \in C_1^S \wedge C_2^S$. This can be proved by induction on n using premises 1 and 2 while noting that the stuttering invariance of A^S, C_1^S, and C_2^S is used in this argument.

[4] To obtain a strict interleaving of P_1 and P_2 actions, such joint actions can be excluded by asserting that the variables written by P_1 and P_2 must be disjoint and never simultaneously updated. Another approach is to label each transition with the agent associated with it, and to have a disjoint set of agents associated with components P_1 and P_2.

- $\sigma \in A^S \xrightarrow{+\triangleright} C^S$.
  By premise 3, for any n, $\sigma[n] \in A^S$ implies $\sigma[n+1] \in C_1^S \wedge C_2^S \longrightarrow\!\triangleright\, C^S$. From step 1, we therefore have $\sigma[n+1] \in C^S$.
- $\sigma \in A \supset C$. For $\sigma \in A$, since $A \supset A^S$, we have by $A^S \xrightarrow{+\triangleright} C_1^S \wedge C_2^S$ that $\sigma \in C_1^S \wedge C_2^S$. By premise 2, this yields $\sigma \in A_1 \wedge A_2$. By premise 1 and the definition of $\xrightarrow{+\triangleright}$, we have that $\sigma \in C_1 \wedge C_2$. We can then apply premise 3 with the definitions of the connectives $\xrightarrow{+\triangleright}$ and $\longrightarrow\!\triangleright$ to obtain $\sigma \in C$.

There are some approaches to modular verification based on model checking that employ a weak form of assume-guarantee reasoning. In the work of Grumberg and Long [GL94], assume-guarantee properties (A, C) are treated as implications $A \supset C$ and not $A \xrightarrow{+\triangleright} C$. Note that the use of implication for assume-guarantee reasoning is not valid in general, and is sound only for a restricted form of Theorem 1 where the cycle of dependencies between A_1, C_2, A_2, and C_1 has been broken. If A and C are just linear-time temporal logic (LTL) formulas, then LTL model checking can be used to verify $A \supset C$ of component P since this implication is also in LTL. If C is a CTL or CTL* formula, then the situation is more complicated since the implication $A \supset C$ is not a well-formed CTL or CTL* state formula, and furthermore, it does not capture the intended meaning of A as an assumption [Jos90] which is that C must hold on the computation tree whose paths have been pruned according to A. Then, A can be chosen as a $\forall$CTL formula that characterizes the subtree of the computation tree that meets the assumption. For the case of $\forall$CTL assumptions and synchronous Moore machine composition, Grumberg and Long give a way of compiling the assumption A into a tableau automaton A^T so that $P \| A^T \models C$ iff $P \models A \supset C$. Kupferman and Vardi [KV96] analyze the complexity of various linear and branching-time variants of modular model checking. Alur and Henzinger [AH96] give an assume-guarantee rule for proving language containment in the context of the synchronous composition of a form of Mealy machines called *reactive modules*.

3 Lazy Composition

Lazy composition differs from the assume-guarantee approach in several respects.

1. *Components are not treated as blackboxes.* Compositional verification merely requires that properties be proved locally at the component level. It does not require that components be treated as blackboxes for this purpose. The assume-guarantee approach requires the assumptions to be discharged solely by means of the guarantee properties of a component. The actual implementation of the component is never used for discharging proof obligations. This means that the guarantee properties must either somehow anticipate the possible constraints imposed by other components, or they must contain implementation details. Lazy composition on the other hand does not take a blackbox view of components and allows the behavioral specification to be

used for discharging the constraints imposed by other components. Since a typical high-level behavioral specification might not contain enough detail to discharge such external constraints, lazy composition allows the constraints to be discharged lazily as the specification is being refined.

Blackbox assume-guarantee specifications can be independently refined to yield implementations in terms of smaller blackbox components. Abadi and Lamport [AL95] give a *decomposition* rule for showing that $P'\|Q'$ refines $P\|Q$ when P' refines P and Q' refines Q. This rule has a premise similar to premise 2 of the compositionality rule of Theorem 1 which has the same drawback of requiring the environment constraints to be anticipated in the blackbox specification.

2. *Composition is not necessarily conjunction.* Conjunction can be used to define the interleaving composition of two asynchronous transition systems by a suitably chosen global constraint (see [AL95] and Footnote 4). Instead of encoding composition using conjunction, we regard the definition of the precise notion of composition as something that is fixed by the model of computation and not by the inference rule for composition. For asynchronous composition, one takes the interleaving of the atomic actions of each component, whereas for synchronous composition, i.e., globally clocked systems, one takes the conjunction of the atomic actions. Other formalisms that have asynchronously operating components with synchronous communication, e.g., CSP [Hoa85], can be modelled by means of a suitable definition of composition.
3. *Environment assumptions are specified as abstract components not properties.* One difficulty with environment assumptions as properties is that they apply to both the environment and the component. Typically, these constraints should apply only to environment actions and not the component actions. If we take the example of a bank account component, the environment might be required to only deposit and not withdraw money from the component but such a constraint should not apply to the component. There is no elegant way of stating this distinction between the component and its environment when the environment constraints are stated as properties rather than abstract components.
4. *No assume-guarantee proof obligations are generated.* With lazy composition, properties of a component P_i are proved in the context of an abstract environment E_i. The composition rule ensures that all local properties are global properties of the composition. It does this by adding (conjoining, as explained below) the environment constraints of one component to the specification of the other component so that the resulting system has the form $(P_1 \wedge E_2)\|(P_2 \wedge E_1)$.

   This form of composition appears dishonest (and lazy) since it sidesteps the question of whether the original specification of one component satisfies the environment assumptions of the other. However, specifications are meant to be partial and are therefore not always strong enough to anticipate the environment constraints that can be placed on a component. The

best that one can do therefore is assert that if the specifications of each component is strengthened with the environment constraints required by the other component, then the resulting system satisfies the local properties of both components. If a component specification is strong enough to discharge any constraints placed on it, then the strengthening is redundant and can be eliminated by simplification. Otherwise, an implementation of the component is required to satisfy the stronger specification including the environment constraint.

The flexibility in postponing the assume-guarantee proof obligations is needed since some proofs might require the additional information that is provided when the specification is refined. If these proof obligations have to be proved as in the assume-guarantee proof method, then the component specifications must be quite detailed and strong. Since no proof obligations are discharged and environment assumptions impose additional, possibly unanticipated, constraints on a component specification, a component cannot be independently refined in the lazy composition approach. A component can only be independently refined when the component specification already implies all the environment constraints that might be required of it. There is, however, an advantage to refining in the global context where these assumptions are known since global properties can be exploited in the refinement (see Section 5).

5. *Composition can yield inconsistent specifications.* This is also the case when composition is defined as conjunction. In the case of lazy composition, this can arise because there is no computation that is compatible with the collection of constraints given in the specification.

Summarizing the discussion so far, lazy composition takes the middle ground between global verification as used in the Owicki–Gries approach and the strictly modular, property-based verification used in the assume-guarantee approach. Lazy composition is a proof style that uses a suitably weak characterization of a cooperative environment in composition with which a component can exhibit a given property. Once such an environment has been identified, the familiar verification techniques for proving safety, liveness, and refinement properties can be used. In the presentation of lazy composition, it will be assumed for convenience that there is a fixed environment specification for each component, but in practice, the environment can be varied according to the desired property of the component.

We now move on to the details of lazy composition for asynchronous transition systems while noting that the techniques can easily be adapted to other models and notions of composition. As already stated, an asynchronous transition system is given by a triple $\langle \Sigma; I, N \rangle$ consisting of the state Σ, an initialization predicate I on the state, and a binary next-state relation N. Given such a triple P of the form $\langle \Sigma; I, N \rangle$, the closed interpretation of P is written as $[\![P]\!]$ and defined as the set sequences $\{\sigma | I(\sigma(0)) \wedge \forall i : N(\sigma(i), \sigma(i+1))\}$. We focus mainly on closed interpretations since one cannot prove interesting properties of computations that admit arbitrary environment actions. When we are talking

about components, we will assume that Σ is the global state type and omit it from the transition system.

Given two transition systems, P_1 of the form $\langle I_1, N_1\rangle$, and P_2 of the form $\langle I_2, N_2\rangle$, the composition $P_1 \| P_2$ is the transition system $\langle I_1 \wedge I_2, N_1 \vee N_2\rangle$. Note that composition essentially yields the interleaving of the component transitions.

The environment E is also given as a transition system $\langle I^e, N^e\rangle$. A component together with its environment is given as a pair $P//E$. The set of computations corresponding to $P//E$, i.e., $[\![P//E]\!]$, is defined as $[\![P\|E]\!]$, i.e., the closed interpretation of $P\|E$. Note that though $P//E$ and $P\|E$ have the same computations, the notation $P//E$ is chosen to emphasize the syntactic asymmetry between component P and environment E.

Given two transition systems P_1 and P_2, the conjunction of these, $P_1 \wedge P_2$, is $\langle I_1 \wedge I_2, N_1 \wedge N_2\rangle$. Let P_i^e denote the component-environment specification $P_i//E_i$. Given two component–environment specifications P_1^e and P_2^e, the *closed co-imposition* of these two specifications $P_1^e \otimes P_2^e$ is defined as the transition system $(P_1 \wedge E_2)\|(P_2 \wedge E_1)$. The *open co-imposition* of P_1^e and P_2^e, written as $P_1^e \times P_2^e$, is defined as $(P_1^e \otimes P_2^e)//(E_1 \wedge E_2)$ and its computations contain actions corresponding to

1. P_1 but respecting E_2,
2. P_2 but respecting E_1, and
3. Environment actions respecting E_1 and E_2.

The closed co-imposition $P_1^e \otimes P_2^e$ yields a system with only the actions of P_1 and P_2, whereas the open co-imposition $P_1^e \times P_2^e$ yields a system with environment actions that are constrained to conform to both E_1 and E_2. Both operators are associative and commutative. It is easy to see that the property preservation result given in Theorem 2 holds so that $[\![P_1^e \otimes P_2^e]\!]$ and $[\![P_1^e \times P_2^e]\!]$ are both subsets of $[\![P_1^e]\!]$, and hence any properties of P_1^e are also properties of $P_1^e \otimes P_2^e$ and $P_1^e \times P_2^e$.

Theorem 2. *1.* $\models [\![P_1^e \otimes P_2^e]\!] \supset [\![P_1^e]\!]$
2. $\models [\![P_1^e \times P_2^e]\!] \supset [\![P_1^e]\!]$

We will henceforth ignore the closed co-imposition operator since its properties are similar to those of open co-imposition. The use of the co-imposition operation in lazy composition will be illustrated in Section 4. The obvious problem with lazy composition is that it asserts the property preservation of $P_1^e \times P_2^e$ and says nothing about $P_1\|P_2$. By discharging proof obligations similar to those in Theorem 1, we can show that the transition system specification $P_1^e \times P_2^e$ is equivalent to the specification $(P_1\|P_2)//(E_1 \wedge E_2)$, where the latter system contains actions corresponding to P_1 and P_2 without any restrictions, and the environment action $E_1 \wedge E_2$. In Section 5, we show that the environment constraints can be discharged in this manner by showing that P_2 refines E_1, and P_1 refines E_2. These refinement proofs can actually be carried out in the context of global invariants, i.e., invariants of $P_1^e \times P_2^e$. The resulting refinement proof obligations are similar to the assume-guarantee proof rule where $A^S \wedge C_1^S \wedge C_2^S$ must entail $A_1 \wedge A_2$.

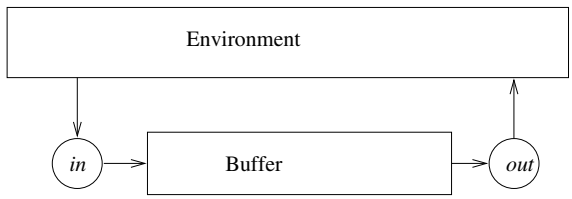


Fig. 3. A FIFO buffer with environment

4 Using Lazy Composition

Lazy composition will be illustrated by means of the example of a FIFO buffer component that is composed from two smaller FIFO buffer components. This examples has been frequently used with minor variations in the compositionality literature [Col93, AL95].

A single (bounded or unbounded) FIFO buffer component shown in Figure 3 consists of a buffer variable b that contains a queue of values, and the input and output variables *in* and *out* which contain values or are empty, i.e., contain a distinguished value $\perp$. Two history variables are used to specify the correct behavior of the buffer. The variable *inh* is a stack of all the non-$\perp$ values placed by the environment into *in*, and the variable *outh* is the stack of non-$\perp$ values read by the environment from *out*. The non-stuttering actions of the buffer are:

- Read a non-$\perp$ value from the variable *in* and enqueue it at the back of b while setting *in* to $\perp$. Formally, this is captured by the relation between the pre-state $\langle in, b, out, inh, outh\rangle$ and the post-state $\langle in', b', out', inh', outh'\rangle$ as

$$read \triangleq \begin{cases} \quad in \neq \perp \\ \wedge\ b' = enqueue(in, b) \\ \wedge\ in' = \perp \\ \wedge\ out' = out \\ \wedge\ outh' = outh \\ \wedge\ inh' = inh \end{cases}$$

- Dequeue a value from the front of queue b and place this value in the variable *out* when *out* is empty. Formally,

$$write \triangleq \begin{cases} \quad nonempty?(b) \\ \wedge\ out = \perp \\ \wedge\ b' = dequeue(b) \\ \wedge\ out' = front(b) \\ \wedge\ outh' = push(front(b), outh) \\ \wedge\ in' = in \\ \wedge\ inh' = inh \end{cases}$$

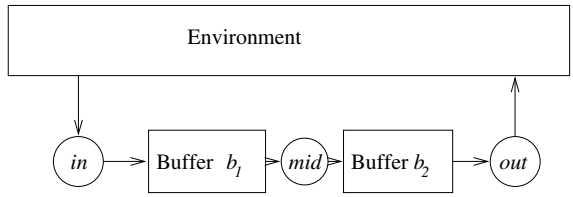


Fig. 4. FIFO buffer composed from smaller buffers

In the initial state, all the variables associated with the buffer are empty. Formally,

$$init_b \stackrel{\triangle}{=} (out = \bot \wedge b = outh = null).$$

The buffer component P is then given by the pair $\langle init_b, read \vee write \rangle$.

The environment component initializes the variables in and inh so that they are both empty:

$$init_e \stackrel{\triangle}{=} (in = \bot \wedge inh = null).$$

In each non-stuttering action, the environment leaves b unchanged and may change the value of in when empty and may set the value of out to $\bot$. Formally,

$$load \stackrel{\triangle}{=} (in = \bot \wedge in' \neq \bot \wedge inh' = push(in', inh))$$

$$unload \stackrel{\triangle}{=} (out \neq \bot \wedge out' = \bot \wedge outh' = outh)$$

$$env \stackrel{\triangle}{=} \begin{cases} (load \vee (in' = in \wedge inh' = inh)) \\ \wedge\ (unload \vee (out' = out \wedge outh' = outh)) \\ \wedge\ b' = b \end{cases}$$

The environment component E is given by the pair $\langle init_e, env \rangle$.

It is easy to prove by induction that

$$[\![P//E]\!] \models \textbf{invariant}\ inh = \overline{in} \circ q2s(b) \circ outh,$$

where $\circ$ is stack concatenation, $q2s(b)$ converts the queue b into a stack by repeatedly pushing elements from the front of queue b, and $\overline{in}$ is $push(in, empty)$ when $in \neq \bot$, and $empty$, otherwise. We have thus proved an invariant of a buffer component P by assuming that the environment behavior is as specified by E. Compositional reasoning is used when two such buffers are composed as shown in Figure 4 to implement a single buffer. We do this by taking one instance $P_1//E_1$ of the buffer as specified above but renaming the variables b to b_1, out to mid, and $outh$ to $midh$, and a second instance $P_2//E_2$ with b renamed to b_2, and where in and inh are just mid and $midh$, respectively. In other words, buffer P_1 communicates values to buffer P_2 via mid.

Having already proved the invariant above for a FIFO buffer P, the goal now is to prove a similar invariant $inh = \overline{in} \circ q2s(b) \circ outh$, for some b, for the

composition $(P_1 \| P_2)//(E_1 \wedge E_2)$ of the two buffers. However, we cannot use the invariant proved of P for composite buffers with P_1 and P_2 since those invariants are proved for the systems $P_1//E_1$ and $P_2//E_2$.

The claim $\models [\![(P_1 \| P_2)//(E_1 \wedge E_2)]\!] \supset [\![P_1//E_1]\!]$ is not provable since the definitions of P_1 and P_2 are not strong enough to imply the constraints E_2 and E_1, respectively. This is because E_1 specifies that each environment action must leave the buffer variable b_1 unchanged and that the variable *in* must be written only by the environment. The actions of P_2 place no constraints on the update of the values of b_1 or *in*. Since we cannot demonstrate $\models [\![(P_1 \| P_2)//(E_1 \wedge E_2)]\!] \supset [\![P_1//E_1]\!]$, the invariant for P_1^e, namely, $inh = in \circ b_1 \circ midh$, cannot be used as a global invariant of $(P_1 \| P_2)//(E_1 \wedge E_2)$.

The best that we can do therefore is to conclude $\models [\![P_1^e \times P_2^e]\!] \supset [\![P_1^e]\!] \wedge [\![P_2^e]\!]$, so that the conjunction of the individual invariants holds for $P_1^e \times P_2^e$. From the conjunction of the two invariants:

1. $[\![P_1^e \times P_2^e]\!] \models$ **invariant** $inh = \overline{in} \circ q2s(b_1) \circ midh$
2. $[\![P_1^e \times P_2^e]\!] \models$ **invariant** $midh = \overline{mid} \circ q2s(b_2) \circ outh$

we can conclude

$$[\![P_1^e \times P_2^e]\!] \models \textbf{invariant } inh = \overline{in} \circ q2s(b_1) \circ \overline{mid} \circ q2s(b_2) \circ outh.$$

So if we take b to be $s2q(q2s(b_1) \circ \overline{mid} \circ q2s(b_2))$ where $s2q$ is the inverse of $q2s$ and converts a stack back into the corresponding queue, we have the desired invariant $inh = \overline{in} \circ q2s(b) \circ outh$ for $P_1^e \times P_2^e$.

In proving this invariant, we have used only the corresponding invariants of the component buffers and some elementary lemmas about the concatenation operation. We have not directly used the specification of individual buffers themselves. We have worked at the level of the *specification* of the behavior of the individual buffers rather than the corresponding *program* which would be a complete specification of each transition. Since specifications can be partial, it makes sense to conjoin the environment constraints to the component specification rather than discharge them as proof obligations. Thus a more detailed implementation will have to satisfy the higher-level specification of the component as well as the constraints on the component imposed by the other components in the combined system.

When refining P_1 to a more refined specification or a program in the context $P_1^e \times P_2^e$, it is valid to use all the global invariants that have been proved of $P_1^e \times P_2^e$. The introductory example involving odd and even numbers can be used to illustrate the use of such invariants in refinement. The system P there is of the form $\langle I_P, N_P \rangle$ where

$$I_P \triangleq even?(z)$$
$$N_P \triangleq (z' = x + y) \wedge (x' = x) \wedge (y' = y)$$

If P's environment constraint D is of the form $\langle I_D, N_D \rangle$ where

$$I_D \triangleq odd?(x) \wedge odd?(y)$$

$$N_D \stackrel{\triangle}{=} odd?(x') \wedge odd?(y') \wedge z' = z$$

then we can prove the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$ for the system $P//D$. Let Q be defined to be $\langle I_Q, N_Q \rangle$ where

$$I_Q \stackrel{\triangle}{=} odd?(x) \wedge odd?(y)$$
$$N_Q \stackrel{\triangle}{=} (x' = x + z) \wedge (y' = y + z) \wedge (z' = z)$$

Let E be the unconstrained system consisting of the everywhere-true initialization predicate and next-state relation. We would now like to show that the constraint D is satisfied by Q, but this is not true in general. It does however hold in the context of the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$. The use of invariants allows $[\![(P//D) \times (Q//E)]\!]$ to be simplified to $[\![P \| Q \| D]\!]$ since $P \wedge E$ simplifies to P, $D \wedge E$ simplifies to D, and $Q \wedge D$ can be simplified to Q given

$$\vdash even?(z) \wedge odd?(x) \wedge odd?(y) \wedge N_Q \supset N_D.$$

We show how invariants can be used in proving a refinement relation between two transition systems using stepwise simulation in the next section.

5 Discharging Proof Obligations by Refinement

We now examine how the familiar notion of refinement via simulation can be used to simplify away the environment constraints E_1 and E_2 in the lazy composition $P_1^e \times P_2^e$. This is analogous to the assume-guarantee proof obligations (premise 2) except that lazy composition is more flexible about how and when these proof obligations are discharged. Recall that in the assume-guarantee approach, the assumptions of one component had to be discharged using the guarantee properties of all the components along with the global environment constraints. As we noted, this has the disadvantage that the guarantee properties have to be chosen to somehow anticipate the likely environment constraints. By contrast, in lazy composition, these proof obligations are discharged lazily during refinement.

The refinement rule establishes the conclusion $\models [\![P]\!] \supset [\![Q]\!]$ by showing that each transition of P can be simulated by a transition of Q . In particular, this means that P inherits all the properties of Q. The simulation of P transitions by Q transitions can be shown in the presence of invariants of P and Q. The invariants might be needed because the simulation relation between the actions of P and Q might not hold outside their respective reachable states. The invariant that is used for P can be an action invariant, a binary relation r on Σ such that $\forall i : r(\sigma(i), \sigma(i+1))$. In this case, we say that **invariant** r holds of σ. Given a state predicate p, an action r, and two transition systems P and Q of the form $\langle I_P, N_P \rangle$ and $\langle I_Q, N_Q \rangle$, respectively, the refinement rule is stated in Theorem 3.

Theorem 3.

$$\frac{\begin{array}{l}[\![P]\!] \models \textbf{invariant } r \\ [\![Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge r(s,s') \wedge N_P(s,s') \supset N_Q(s,s') \\ \vdash I_P(s) \supset I_Q(s)\end{array}}{\models [\![P]\!] \supset [\![Q]\!]}$$

The proof of the refinement rule is by a straightforward induction on the length of the computations in $[\![P]\!]$. The relevance of the refinement rule for compositional verification is that we can use it to eliminate the constraints imposed on one component by another. When composing specifications using the coimposition operator, we end up with a specification $P_1^e \times P_2^e$ which is equivalent to $(P_1 \wedge E_2) \| (P_2 \wedge E_1) \| (E_1 \wedge E_2)$. To eliminate, say, E_2 from this specification, we need to show that $(P_1 \wedge E_2) \| (P_2 \wedge E_1) \| (E_1 \wedge E_2)$ can be refined by $P_1 \| (P_2 \wedge E_1) \| (E_1 \wedge E_2)$. The constraint E_1 can also be similarly eliminated. This kind of refinement can be carried out with the aid of a simple corollary to the refinement rule that can be used to show that $[\![P \| Q]\!]$ refines $[\![P \wedge E \| Q]\!]$ by showing that each P transitions can be simulated by an E transition.

Corollary 4.

$$\frac{\begin{array}{l}[\![(P \wedge E) \| Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge N_P(s,s') \supset N_E(s,s') \\ \vdash I_P(s) \supset I_E(s)\end{array}}{\models [\![P \| Q]\!] \supset [\![(P \wedge E) \| Q]\!]}$$

Note that any global invariant p can be used in proving the stepwise simulation. This is what justifies the use in Section 4 of the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$ in showing that the strengthening of the specification Q with D is redundant.

6 Liveness

Compositional liveness reasoning is needed for showing progress properties for a component contingent on similar progress properties of other components. For example, the FIFO buffer can only guarantee that an output will always eventually be written if the environment can guarantee that a value in the *out* variable will always eventually be read.

Liveness or progress assumptions have to be handled with some care in compositional verification. For example, suppose a component P guarantees that output z is eventually 4 assuming the input x is eventually 3, and conversely, component Q guarantees an eventual output 3 on x assuming that the input z is eventually 4. If the guarantee properties are used to discharge assumptions, then the composed system $P \| Q$ guarantees that z will eventually take on the value 4 and that eventually x will take on the value 3. This would be unsound since the system actually need not obey either eventuality for x or z and the individual assume-guarantee properties would still be satisfied. The assume-guarantee

proof rule is carefully crafted to rule out this kind of circularity by ensuring in premise 2 that the assumptions have to be satisfied solely from the safety parts of the guarantee properties. Component liveness properties are instead expressed as implications in the property C_1 of a component P_1, where the antecedent of the implication is the fairness constraint on the other component. This antecedent is of course easily discharged in the conjunction $C_1 \wedge C_2$ if C_2 includes the fairness condition of P_2.

To admit proofs of liveness properties in lazy composition, it will be necessary to extend the notion of a transition system to include fairness conditions. An asynchronous transition system with fairness is of the form $\langle \Sigma; I, N, F\rangle$ where F is a *fairness* property that a valid computation must satisfy, i.e., $[\![\langle I, N, F\rangle]\!] \equiv [\![\langle I, N\rangle]\!] \wedge F$. It is desirable that the F component be used only to establish progress properties so that any safety property should follow from the system $\langle \Sigma; I, N\rangle$ without F. For this to be the case, the fairness condition F should be *machine closed*, i.e., any finite prefix $\sigma[n]$ in $\langle I, N\rangle$ should be extendable to a sequence $\sigma[n] \circ \rho$ in $[\![\langle I, N, F\rangle]\!]$. F is machine closed with respect to the transition system $[\![\langle I, N\rangle]\!]$ iff $[\![\langle I, N, F\rangle]\!]^S = [\![\langle I, N\rangle]\!]$. For example, if P is a transition system with only one state component x whose value is initially 0, and a next-state relation $x' = x + 2 \vee x' = x + 3 \vee x' = x$, then the property **eventually** $x = 3$ is not a machine-closed fairness condition since it excludes the computations in which x takes the value 2.

Typical notions of fairness such as weak and strong fairness are machine closed with respect to the closed interpretation of a single transition system. An action r is said to be enabled in a state s, formally $\mathit{enabled}(r)(s)$, iff there exists a state s' such that $r(s, s')$ holds. A predicate p holds infinitely often on a sequence σ iff $\forall i : \exists j : j > i \wedge p(\sigma(j))$. Similarly, an action r holds infinitely often on σ iff $\forall i : \exists j : j > i \wedge r(\sigma(j), \sigma(j+1))$. A sequence σ is said to be *weakly fair* with respect to an action r iff either $\neg \mathit{enabled}(r)$ holds infinitely often or r holds infinitely often on σ. A sequence σ is said to be *strongly fair* with respect to action r iff r holds infinitely often on σ when $\mathit{enabled}(r)$ does. It can be shown that F is machine closed with respect to transition system $\langle I, N\rangle$ if F is a conjunction of weak and strong fairness assertions on actions $r_1, \ldots, r_n$ such that each r_i is *unblocked* in $\langle I, N\rangle$,[5] i.e.,

$$[\![\langle I, N\rangle]\!] \models \textbf{invariant}\ \mathit{enabled}(r_i) \supset \mathit{enabled}(r_i \wedge N).$$

When F is machine closed with respect to $\langle I, N\rangle$, we say that the fair transition system $\langle I, N, F\rangle$ is machine closed.

The situation is not so simple for transition systems whose computations include both component and environment transitions. The definition of composition for fair asynchronous transition systems is

$$\langle I_1, N_1, F_1\rangle \| \langle I_2, N_2, F_2\rangle \stackrel{\triangle}{=} \langle I_1 \wedge I_2, N_1 \vee N_2, F_1 \wedge F_2\rangle.$$

[5] Abadi and Lamport [AL95] state this constraint differently by requiring each r_i to be a possible program action. This is equivalent since the fairness constraint r_i can just as well taken to be $N \wedge r_i$.

The purpose of distributing the fairness conditions among the various components is to allow componentwise properties to be deduced using just the relevant global fairness conditions. In particular, machine closure is defined only with respect to a closed interpretation so that it is only required for specifications such as $P//E$ or $P_1^e \times P_2^e$. Given the above definition of composition for fair asynchronous transition systems, all fairness conditions are global and apply to all components.

In a blackbox style of component specification, implementability considerations require the component fairness condition to be machine closed with respect to the open interpretation, and also *receptive*, i.e., machine closed without relying on cooperation from the environment. The receptiveness constraint on the fairness condition can exclude unconditional strong fairness constraints since a hostile environment can enable and disable a component action r without allowing the component a chance to execute r. Receptiveness is a sensible restriction when specifying an open component operating in an uncontrolled environment, but this is not the situation in compositional verification since the environment includes components whose specifications are an integral part of the design.

Given the definition of composition extended with fairness conditions, the definitions of the operations $\otimes$ and $\times$ remain unchanged from Section 3. The property preservation results claimed in Theorem 2 also holds in the presence of fairness conditions.

There is however one serious problem with lazy composition in the presence of fairness. The co-imposition $P_1^e \times P_2^e$ of machine-closed specifications $P_1//E_1$ and $P_2//E_2$ is not necessarily machine closed. The co-imposition contains the conjunctions $P_1 \wedge E_2$, $P_2 \wedge E_1$, and $E_1 \wedge E_2$. The conjunction of two transitions systems $\langle I_1, N_1, F_1\rangle \wedge \langle I_2, N_2, F_2\rangle$ is defined as $\langle I_1 \wedge I_2, N_1 \wedge N_2, F_1 \wedge F_2\rangle$. Since the actions of the conjoined transition system are specified by $N_1 \wedge N_2$ which is more constrained than either N_1 or N_2, the fairness condition $F_1 \wedge F_2$ might not be machine closed in the resulting transition system. For example, let $x' = x + 1$ be a possible action of P_1 where P_1 initializes x to 0 and has no actions that decrement or reset x. Then it can be proved of $[\![P_1^e]\!]$ that if the increment action is weakly fair, then **eventually** $x = 3$. However, if E_2 in P_2^e requires that x not be incremented, then the set of computations $[\![P_1^e \times P_2^e]\!]$ is empty since the only possible computations are those where the value of x is never changed and these are ruled out by the weak fairness requirement on the increment action. Of course, the property **eventually** $x = 3$ is vacuously preserved in this case. Note that machine closure is violated in this example even if E_2 contains no fairness conditions simply because E_2 blocks a fair action of P_1.

There is therefore a proof obligation that the system $P_1^e \times P_2^e$ be shown to be machine closed. In the special case of fairness conditions that only contain weak and strong fairness assertions, this proof obligation can be discharged by showing that each fair action is unblocked in the combined system.

The notion of refinement used to eliminate environment constraints has to be extended to fair asynchronous transition systems. The goal is to show that $\models [\![\langle I_P, N_P, F_P\rangle]\!] \supset [\![\langle I_Q, N_Q, F_Q\rangle]\!]$. For this, we need to add one additional

premise to the refinement rule in Section 5.

Theorem 5.

$$\frac{\begin{array}{l}[\![P]\!] \models \textbf{invariant } r \\ [\![Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge r(s,s') \wedge N_P(s,s') \supset N_Q(s,s') \\ \vdash I_P(s) \supset I_Q(s) \\ \models [\![\langle I_P, N_P, F_P\rangle]\!] \supset F_Q\end{array}}{\models [\![P]\!] \supset [\![Q]\!]}$$

The discharging of the new premise can require temporal reasoning. For the case of fairness conditions that are conjunctions of weak and strong fairness assertions, one can simply show that to any weakly fair action r_i in Q, there is a weakly or strongly fair action r'_j in P such that

$$[\![\langle I_P, N_P\rangle]\!] \models \textbf{invariant } r'_j \supset r_i$$

and

$$[\![\langle I_P, N_P\rangle]\!] \models \textbf{invariant } \mathit{enabled}(r_i) \supset \mathit{enabled}(r'_j \wedge N_P).$$

Similarly, to each strongly fair action in Q, there must be a corresponding strongly fair action in P.

Returning to the example of the FIFO buffer, if the actions *read* and *write* are weakly fair, and the *unload* action for the buffer environment is weakly fair, then in any fair computation of this transition system it is always the case that a state in which $x = \mathit{in} \neq \perp$ is eventually followed by (i.e., *leads to*, in the terminology of temporal logic) a state in which $\mathit{out} = x$.

7 Discussion

We have argued thus far that lazy composition is superior to the assume-guarantee method for compositional verification on the grounds that:

1. Lazy composition employs proof methods that are already familiar whereas the assume-guarantee proof rule is quite formidable.
2. Assume-guarantee methods require specifications that can anticipate future environment constraints.
3. The assume-guarantee assumptions apply to both component and environment and it is awkward to restrict these so that they only constrain the environment.
4. Assume-guarantee specifications are more appropriate for writing blackbox characterizations of open components rather than for compositional verification where the point is to achieve a useful decomposition of the verification task.

The advantage of lazy composition with respect to non-compositional, global reasoning as characterized by the Owicki–Gries approach [OG76] is that it combines the simplicity of global reasoning with the economy of using an abstract

characterization of the environment rather than the actual components in the environment. This abstract characterization can be used to prove a number of component properties. The actual components can then be shown to conform to this abstract characterization by means of a refinement proof.

The Owicki–Gries approach is subsumed by lazy composition. If P is a component that is required to satisfy an invariant p, then we can take the environment E to be the transition system that merely preserves p, i.e., $\vdash N_E(s, s') \wedge p(s) \supset p(s')$. Then the refinement proof obligation reduces to a global demonstration that each component that is composed with P preserves the invariant. This is obviously the most general assumption one can make of an environment to P given that one wants to establish the invariant p, but it is not the optimal way to use lazy composition. The more appropriate use of lazy composition is by describing the allowed or intended environment actions that are relevant to the state variables that are read or written by component P and that are needed to obtain useful properties of P. Thus lazy composition modularizes the global reasoning by identifying suitable abstractions for the environment of each component.

7.1 Other Applications of Lazy Composition

We have employed lazy composition in the verification of the safety properties of an N-process mutual exclusion algorithm [Sha97] and the alternating-bit communication protocol [BSW69]. These verifications have been carried out using PVS [ORS92]. The mutual exclusion algorithm has been verified using a combination of induction, abstraction, and model checking. The algorithm uses a Boolean *turn* variable for each process to arbitrate access to successive rounds of competition using 2-process mutual exclusion, for eventual access to the critical section. The environment to each process has to be constrained to not affect the value of this *turn* variable in an undesirable way, e.g., when a process has checked the *turn* value and has entered its critical section.

The example of the alternating-bit protocol consists of a sender process, a receiver process, and the message and acknowledgement channels. The sender process constrains its environments merely to drop messages from the message channel, and the receiver process similarly constrains the value of the acknowledgement channel. With these constraints, it is possible to carry out a modular verification of the safety property of the alternating-bit protocol where all the invariants are proved solely by local reasoning in terms of the receiver or the sender process, possibly using previously proved global invariants.

8 Conclusions

We have presented the details of the paradigm of lazy compositional verification. This approach has several advantages over the assume-guarantee paradigm. We have formalized lazy composition verification within PVS [ORS92] and verified

several medium-scale examples with this approach. We do not yet have any conclusive evidence that the method scales up to larger systems. Lazy composition can be adapted to models other than asynchronous transition systems by suitably altering the definitions of composition, conjunction, and refinement. Lazy composition does not need any new verification machinery since it builds on existing techniques for proving safety, liveness, and refinement properties.

References

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AP93] Martín Abadi and Gordon D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*, volume 191 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260, 261, May 1969.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[CMP94] Edward Chang, Zohar Manna, and Amir Pnueli. Compositional verification of real-time systems. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 458–465, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[Col93] P. Collette. Application of the composition principle to Unity-like specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 230–242, Berlin, 1993. Springer-Verlag.

[Col94] Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50(1):31–35, April 1994.

[dBdRR90] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[dBdRR94] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes*

in Computer Science, Noordewijkerhout, The Netherlands, 1994. Springer Verlag.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science.* Springer Verlag, 1991.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[Jos90] B. Josko. Verifying the correctness of AADL modules using model checking. In de Bakker et al. [dBdRR90], pages 386–400.

[Kur93] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1993.

[KV96] O. Kupferman and M. Y. Vardi. Module checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification96*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer Verlag, 1996.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth Annual Symposium on Principles of Distributed Computing, New York*, pages 137–151. ACM Press, 1987.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[PJ91] P. K. Pandya and M. Joseph. P–A logic — a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logic and Models of Concurrent Systems*, NATO-ASI, pages 123–144. Springer Verlag, 1984.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, Ithaca, NY, October 1987.

[Sha93a] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[Sha93b] N. Shankar. A lazy approach to compositional verification. Technical Report SRI-CSL-93-8, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[Sha97] N. Shankar. Machine-assisted verification using theorem proving and model checking. In Manfred Broy and Birgit Scheider, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Science*, pages 499–528. Springer, 1997.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer Verlag, 1985.

[XCC94] Q.-W. Xu, A. Cau, and P. Collette. On unifying assumption–commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR'94*, volume 836 of *Lecture Notes in Computer Science*, pages 267–282. Springer Verlag, 1994.

[XdRH97] Q.-W. Xu, W.-P. de Roever, and J.-F. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

Compositional Reasoning Using the Assumption–Commitment Paradigm

Qiwen Xu and Mohalik Swarup*

International Institute for Software Technology
United Nations University
P.O. Box 3058, Macau
Email: qxu@iist.unu.edu

Abstract. Assumption–Commitment paradigms have been investigated to derive tractable rules for composing specifications of concurrent systems. We first give a short survey of several typical composition rules, and then we adopt the principle to reason about real time systems. An extension of Duration Calculus capable of describing infinite behaviours and instantaneous actions is proposed. In the calculus, verification techniques based on assumption–commitment are incorporated.

1 Introduction

Compositionality is an important property for verification and development of any sizeable systems [10]. In verification, it allows the likely formidable task of proving correctness of the whole system to be decomposed into more manageable pieces, and in development, it additionally supports early reasoning of designs before they are further developed and implemented. However, in the presence of concurrency, compositionality is difficult to achieve, due to interactions among processes. A compositional theory consists of a semantics and a set of rules for verification and development, with the rules justified in the semantics. In the literature, several compositional semantics were formulated, usually in the context of some particular forms of concurrency mechanisms. Typically, the semantics of a system is given as a set of behaviours consisting of interleaving atomic actions from different processes, and the semantics of a parallel composition can be viewed as some forms of intersections of the semantics of constituent components.

Defining the semantics of parallel composition as set intersection leads to a simple rule of composition:

Rule 1

$$\frac{P_i \;\; \underline{sat} \;\; S_i}{P_1 \parallel P_2 \;\; \underline{sat} \;\; S_1 \wedge S_2}$$

* On leave from The Institute of Mathematical Sciences, C. I. T. campus, Chennai 600 113, India. Email: swarup@imsc.ernet.in.

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 565-583, 1998.
Springer-Verlag Berlin Heidelberg 1998

Usually the specification S_i says that P_i guarantees some properties under certain conditions. Therefore, it is most natural to express the specification by logical implication, that is, as $A_i \Rightarrow C_i$, where A_i is called an assumption and C_i is called a commitment. Sometimes, assumptions and commitments of several components are mutually dependent, and then composing them to conclude a global property becomes a nontrivial task. For instance, it is easy to see that the following simple minded rule is unsound.

An incorrect rule (circular reasoning)

$$\frac{\begin{array}{l} A \wedge C_1 \Rightarrow A_2 \qquad A \wedge C_2 \Rightarrow A_1 \\ P_i \ \underline{sat} \ A_i \Rightarrow C_i \end{array}}{P_1 \parallel P_2 \ \underline{sat} \ A \Rightarrow C_1 \wedge C_2}$$

In some cases, the dependency of components is acyclic, and then the following rule could be used

Rule 2

$$\frac{\begin{array}{l} A \wedge C_1 \wedge \cdots \wedge C_{i-1} \Rightarrow A_i \\ P_i \ \underline{sat} \ A_i \Rightarrow C_i \end{array}}{P_1 \parallel \cdots \parallel P_n \ \underline{sat} \ A \Rightarrow C_1 \wedge \cdots \wedge C_n}$$

In the general case, where the dependency of components is circular, assumptions and commitments have to be composed in more involved ways. One method to avoid circular reasoning is to introduce a more sophisticated interpretation of an assumption–commitment specification than simple implication. This was first proposed by Misra and Chandy [18] for the verification of `OCCAM`-like communication based programs. The method was further developed by several other researchers, e.g., Pandya and Joseph [23]. Jones [12] proposed a similar method for shared variable concurrency, where the assumption–commitment pair can be interpreted in the same way. Jones' method is further developed by Stirling [27], Stølen [28] and ourselves [31].

Underlying these methods is a 'Non First Strike' principle, where the assumption – commitment pair (A, C) is defined by a 'spiral' interpretation:

> If the environment satisfies assumption A until the current moment, then the component satisfies commitment C after the current step, that is, until the next moment.

When a system is composed of several components, the environment of a component is formed by the rest of the system. If each component satisfies the individual specification, which says that the component can only make a wrong move after the environment has, it follows that none of the components can make a wrong move.

This paper is organised as follows. First, we give an overview of the assumption – commitment paradigm by reviewing several typical rules, and these include the rules for verifying `OCCAM`-like and shared variable programs, as well as a rule

in a temporal setting. Next we investigate compositional verification of real time systems. For this, one needs a logic capable of expressing timing information. We propose an extension of Duration Calculus, which is an interval based temporal logic, and our extension is capable of describing both infinite behaviours and instantaneous actions. Verification techniques based on assumption–commitment are incorporated in the calculus. The paper is concluded with a short discussion.

2 The Assumption–Commitment Paradigm

2.1 Extensions of Hoare Logic

Assumption–commitment rules were first studied as extensions of Hoare Logic. Among several variations we briefly review a typical one, which can be considered as roughly about total correctness. The specification is a tuple $(p,\ A,\ C,\ q)$ where p and q are precondition and postcondition whereas A and C are assumption and commitment describing interactions between the environment and the component. To give a more precise interpretation of the correctness notion, we introduce some definitions. A behaviour is a finite or infinite sequence of states $\sigma = \sigma_0\sigma_1 \ldots \sigma_i \ldots$, where σ_i is the $(i+1)$-th state of σ. As usual, $\sigma \models \varphi$ indicates that σ satisfies predicate φ. Let $\sigma|_k$ denote the prefix of σ of length k. A process P satisfies a specification $(p,\ A,\ C,\ q)$, denoted as P $\underline{sat}$ $(p,\ A,\ C,\ q)$, is interpreted as follows: for any behaviour σ of P

I) if $\sigma_0 \models p$ and for any k, $\sigma|_k \models A$, then $\sigma|_{k+1} \models C$,
II) if $\sigma_0 \models p$ and $\sigma \models A$, then
- P performs only a finite amount of transitions,
- if σ is a finite behaviour, then the final state satisfies q.

OCCAM-like Programs

The semantics of an OCCAM-like program is defined by observations over communication traces and local states. Let the variable space be extended with a variable h storing communications performed so far. The semantics can be defined as a set of behaviours of the extended states. In the specification, A and C are predicates over h. Projections of the trace onto channels a, a and b for example are denoted by h_a and h_{ab} respectively. Precondition p and postcondition q are predicates over all the variables. For any sequence of states σ, predicate φ which is either the assumption A or commitment C, define

$\sigma \models \varphi$ iff $\sigma_i \models \varphi$ for any state σ_i in σ.

Subsequently, condition I) above becomes

if $\sigma_0 \models p$ and for any k, any $0 \leq i < k$, $\sigma_i \models A$, then for any $0 \leq i \leq k$ $\sigma_i \models C$.

The following parallel composition rule was studied in [18, 23]:

Rule 3

$$\frac{\begin{array}{lll} P_1 \ \underline{sat} \ (p,\ A_1,\ C_1,\ q_1) & A \wedge C_1 \Rightarrow A_2 & p \wedge A \Rightarrow A_1 \wedge A_2 \\ P_2 \ \underline{sat} \ (p,\ A_2,\ C_2,\ q_2) & A \wedge C_2 \Rightarrow A_1 & C_1 \wedge C_2 \Rightarrow C \end{array}}{P_1 \parallel P_2 \ \underline{sat} \ (p,\ A,\ C,\ q_1 \wedge q_2)}$$

Suppose process P_1 has two input channels a and b, with channel a from the overall environment and channel b from P_2. Assume (by A) that the overall environment either sends nothing or number 2 on channel a, and suppose process P_2 guarantees (by C_2) to send nothing or number 5 on channel b. Then the combined information process P_1 may assume is the *conjunction* of the two: the only possible message on channel a is 2 and the only possible message on channel b is 5.

Example: Consider the simple program $P_1 \stackrel{\text{def}}{=} a?x; b!x^2; c?y \parallel P_2 \stackrel{\text{def}}{=} b?z; c!z^3$. Assume the input value on channel a, if there is one, is 3. This is expressed by letting A be the predicate $h_a \preceq < (a,3) >$, where $\preceq$ denotes prefix relation. Then we know that the value passed to process P_2 is 9, and the value sent back to P_1 is 729. The program satisfies the specification

$$\begin{array}{ll} p:\ h_{abc} = <>, & q:\ x = 3 \wedge y = 729 \wedge z = 9, \\ A : h_a \preceq < (a,3) >, & C : h_b \preceq < (b,9) > \wedge\ h_c \preceq < (c,729) > \end{array}$$

and this can be proved by using the parallel composition rule. Indeed, P_1 satisfies

$$\begin{array}{ll} p:\ \ h_{abc} = <>, & q_1:\ x = 3 \wedge y = 729, \\ A_1 : h_a \preceq < (a,3) > \wedge\ h_c \preceq < (c,729) >, & C_1 : h_b \preceq < (b,9) > \end{array}$$

and P_2 satisfies

$$\begin{array}{ll} p:\ \ h_{abc} = <>, & q_2:\ z = 9, \\ A_2 : h_b \preceq < (b,9) >, & C_2 : h_c \preceq < (c,729) > . \end{array}$$

Shared Variable Programs

The standard way to give a compositional semantics to shared variable programs, as suggested first by Aczel (cited e.g., in [10]), is to define the semantics as a set of labelled state transition sequences, where the label records whether the transition is from the environment or from the component. This can be expressed in our setting by introducing a variable, say u, in the state, to record the transition agent. More precisely, in behaviour σ, a transition (σ_i, σ_{i+1}) is from component P if $\sigma_{i+1}(u) = P$ and from its environment if $\sigma_{i+1}(u) \neq P$.

In the specification, predicates A and C are binary state predicates, with the convention that unprimed variables refer to the state before and primed variables to the state after the transition. In the literature, e.g. [28, 31], the interpretation that a process P satisfies specification $(p,\ A,\ C,\ q)$ was defined as: for any behaviour σ of P

I') if $\sigma_0 \models p$, and for any k, any transition (σ_i, σ_{i+1}) such that $i < k$ and $\sigma_{i+1}(u) \neq P$, $(\sigma_i, \sigma_{i+1}) \models A$, then $(\sigma_i, \sigma_{i+1}) \models C$ for any $i < k$ such $\sigma_{i+1}(u) = P$,

II') if $\sigma_0 \models p$ and any $(\sigma_i, \sigma_{i+1}) \models A$ where $\sigma_{i+1}(u) \neq P$, then
- P performs only a finite amount of transitions,
- if σ is a finite behaviour, then the final state satisfies q.

Note although condition I') is not directly expressed in the spiral form, it is equivalent to the following

if $\sigma_0 \models p$, and for any k, any transition (σ_i, σ_{i+1}) such that $i < k$ and $\sigma_{i+1}(u) \neq P$, $(\sigma_i, \sigma_{i+1}) \models A$, then $(\sigma_i, \sigma_{i+1}) \models C$ for any $i < k+1$ such $\sigma_{i+1}(u) = P$.

The parallel composition rule for shared variable programs is [12]:

Rule 4

$$\frac{\begin{array}{lll} P_1 \ \underline{sat} \ (p,\ A_1,\ C_1,\ q_1) & A \vee C_1 \Rightarrow A_2 & \\ P_2 \ \underline{sat} \ (p,\ A_2,\ C_2,\ q_2) & A \vee C_2 \Rightarrow A_1 & C_1 \vee C_2 \Rightarrow C \end{array}}{P_1 \parallel P_2 \ \underline{sat} \ (p,\ A,\ C,\ q_1 \wedge q_2)}$$

The assumption A_1 specifies the state changes that the component process P_1 can tolerate from its environment. Both state changes by process P_2 (for which C_2 is guaranteed) as well as state changes of the overall environment (for which A is assumed) must be viewed as state changes by the environment of P_1. Since those state changes are *interleaved* in the execution model, the condition $A \vee C_2 \Rightarrow A_1$ is precisely the one needed to ensure that the assumption of process P_1 is respected by its environment. Similar arguments also hold for process P_2. Finally, if both C_1 and C_2 are guaranteed, then the condition $C_1 \vee C_2 \Rightarrow C$ ensures that C is guaranteed for the state changes by $P_1 \| P_2$.

Example: $x := x + 1 \parallel x := x + 1$.
Verification of this program needs auxiliary variables. Let P_1 be the program $(x, z_1 := x + 1, z_1 + 1)$, and P_2 the program $(x, z_2 := x + 1, z_2 + 1)$, then P_1 satisfies

$$\begin{array}{ll} p: & x = z_1 = z_2 = 0, \\ A: & x' = z_1' + z_2' \wedge z_1' = z_1, \\ C: & x' = z_1' + z_2' \wedge z_2' = z_2, \\ q: & x = z_1 + z_2 \wedge z_1 = 1 \end{array}$$

and P_2 satisfies

$$\begin{array}{ll} p: & x = z_1 = z_2 = 0, \\ A_1: & x' = z_1' + z_2' \wedge z_2' = z_2, \\ C_1: & x' = z_1' + z_2' \wedge z_1' = z_1, \\ q: & x = z_1 + z_2 \wedge z_2 = 1. \end{array}$$

By the parallel composition rule, $P_1 \parallel P_2$ satisfies

$$\begin{array}{ll} p: & x = z_1 = z_2 = 0, \\ A_2: & x' = x \wedge z_1' = z_1 \wedge z_2' = z_2, \\ C_2: & true, \\ q: & x = z_1 + z_2 \wedge z_1 = 1 \wedge z_2 = 1. \end{array}$$

This implies that $x = 2$ is a valid postcondition.

2.2 Assumption - Commitment in Temporal Logics

To express more general properties, some forms of temporal logics should be used. Assumption–commitment paradigm has been studied by Pnueli [24], Barringer & Kuiper [4], and more recently by Abadi & Lamport [1–3]. In this section, we present a rule which is formulated in [9, 30], based mainly on the work by Abadi & Lamport.

A temporal predicate φ is a *safety* predicate iff for any behaviour σ:

$$\sigma \models \varphi \qquad \text{iff} \qquad \text{for any finite } k \text{ such that } 0 \leq k \leq |\sigma| : \sigma|_k \models \varphi.$$

where $|\sigma|$ denotes the length of σ. The assumption–commitment specification is of the form $(A, < C^S, C^R >)$ where A, C^S and C^R are temporal predicates. The assumption about the environment is described by A, which is restricted to be a safety predicate. The commitment is divided into a safety predicate C^S and a remaining predicate C^R. Liveness should be described by C^R, but for the soundness of the composition rule it is not necessary to insist on C^R to be a pure liveness predicate. The spiral interpretation is formally defined as

$$\begin{array}{lll} \sigma \models \varphi_1 \hookrightarrow \varphi_2 & \text{iff} & \text{for any finite } k \text{ such that } 1 \leq k \leq |\sigma|, \\ & & \text{if } \sigma|_{k-1} \models \varphi_1 \text{ then } \sigma|_k \models \varphi_2 \end{array}$$

and

$$\begin{array}{l} \models P \ \underline{sat} \ (A, \ < C^S, C^R >) \\ \text{iff} \\ \text{for all } \sigma \text{ of } P: \ \sigma \models (A \hookrightarrow C^S) \ \wedge \ (A \Rightarrow C^R). \end{array}$$

Therefore, for any behaviour σ of P,

I") if any prefix of σ satisfies A, then C^S holds after the transition extending that prefix, and

II") if σ satisfies A, then σ also satisfies C^R.

Rule 5

$$\frac{\begin{array}{l} init(A \Rightarrow A_1 \wedge A_2) \\ A \wedge C_1^S \wedge C_2^S \Rightarrow A_1 \wedge A_2 \\ C_1^S \wedge C_2^S \Rightarrow C^S \\ A \wedge C_1^R \wedge C_2^R \Rightarrow C^R \\ P_i \ \underline{sat} \ (A_i, \ < C_i^S, C_i^R >) \end{array}}{P_1 \parallel P_2 \ \underline{sat} \ (A, \ < C^S, C^R >)}$$

where *init* is defined as

$$\sigma \models init\,\varphi \qquad \text{iff} \qquad \sigma_0 \models \varphi.$$

It is shown in [9, 30] that previous two rules, developed independently for OCCAM-like and shared variable programs, can be considered as special cases of the present rule.

3 Duration Calculus

To verify real-time systems, one needs a logic capable of expressing timing information. Among the real-time temporal logics that have been developed, one class of the logics is point-based, usually as extensions of well-established temporal logics, whereas the second class, known as Duration Calculus [5, 7], is interval-based and is an extension of Interval Temporal Logic (ITL) [19] to dense time domains.

3.1 The Classical Duration Calculus

The classical Duration Calculus [5], abbreviated as DC, was developed to reason about piece-wise continuous Boolean functions of time called states, which model the status of the system. In DC, time is represented by non-negative reals. Intervals and interpretations of state variables are defined as follows

$$\mathsf{Intv} \stackrel{\text{def}}{=} \{[c,d] \in \mathsf{Time} \times \mathsf{Time} \mid c \leq d\}$$
$$\mathcal{I} \in SVar \to \mathsf{Time} \to \{0,1\}.$$

Roughly speaking, a model is a pair $(\mathcal{I}, [c,d])$. A Boolean state expression B is constructed from (Boolean) state variables with Boolean connectives and its duration in a model $(\mathcal{I}, [c,d])$ is defined as

$$[\![\int B]\!](\mathcal{I}, [c,d]) \stackrel{\text{def}}{=} \int_c^d [\![B]\!](\mathcal{I}, t)dt$$

where $[\![B]\!](\mathcal{I}, t)$ denotes the value of B at time t under state interpretation $\mathcal{I}$. The length l of an interval is defined as

$$l \stackrel{\text{def}}{=} \int 1$$

and it is easy to prove

$$[\![l]\!](\mathcal{I}, [c,d]) = d - c.$$

The modality 'chop' of ITL is defined as follows: for any formulae A and B

$$(\mathcal{I}, [c,d]) \models A;B$$
$$\text{iff there exists } m \text{ such that } c \leq m \leq d \text{ and}$$
$$(\mathcal{I}, [c,m]) \models A \text{ and } (\mathcal{I}, [m,d]) \models B.$$

The axiomatic system of DC includes that of ITL and the following axioms about duration.

$$\int 0 = 0$$

$$\int 1 = l$$

$$\int B \geq 0$$

$$\int B_1 + \int B_2 = \int (B_1 \vee B_2) + \int (B_1 \wedge B_2)$$

$$((\int B = x); (\int B = y)) \Rightarrow (\int B = x + y)$$

$\int B_1 = \int B_2$, provided $B_1 \Leftrightarrow B_2$ holds in propositional logic.

3.2 Duration Calculus with Weakly Monotonic Time and Infinite Intervals

The model of DC with piece-wise continuous functions is not adequate for lower level description. At the lower level, a real-time control system for example, typically contains primitives representing computation steps and switches between phases of dynamical activities. These discrete actions are usually considered instantaneous, and subsequently several such actions may happen at the same real time point governed possibly by a causal order. Such abstraction, proposed initially in the work on synchronous languages, provides substantial simplification in verification. To reason about discrete actions and their compositions, one needs a more involved logic. Koymans [16] suggested that a time point can be defined as a pair (r, n), with r denoting the real time and n the causal order. A variant of Duration Calculus, called Weakly Monotonic Duration Calculus, abbreviated as WDC, was formed by Pandya and Dang over such time structures [22]. A similar logic was suggested by Liu, Ravn and Li [17].

Both DC and WDC are defined over finite intervals. However, real-time systems often exhibit infinite behaviours. In [6], an extension of DC was studied by Zhou, Dang and Li, where infinite behaviours are described by their finite approximations. This approach avoids direct interpretation of formulae over infinite intervals but has the disadvantage that properties are somewhat cumbersome to express. As an alternative formulation, we have included infinite intervals directly in DC [29] following an approach by Moszkowski for ITL [21]. In this section a further extension is proposed where we include infinite intervals along both real and causal dimensions. The resulting logic is called Duration Calculus with Weakly Monotonic Time and Infinite Intervals, abbreviated as WDCI.

A time domain of WDCI is a total order $(\mathcal{T}, <)$, where

- $\mathcal{T} \subseteq (\mathsf{NonNegReal} \cup \{\infty\}) \times (\mathsf{Nat} \cup \{\infty\})$, with $(0, 0) \in \mathcal{T}$
- $(r_1, n_1) < (r_2, n_2)$ iff
  $(r_1 \neq \infty \wedge r_1 \leq r_2 \wedge n_1 < n_2) \vee (r_1 < r_2 \wedge n_1 \neq \infty \wedge n_1 \leq n_2)$
- for any $t \notin \mathcal{T}$, there exists $t' \in \mathcal{T}$, such that $t \not\leq t'$ and $t' \not\leq t$.

The last condition indicates that a time domain is maximal, in the sense that adding any other time point will cause the set to be no longer a total order.

By the definition, exactly one of (r, ∞), (∞, n) and (∞, ∞) is in a time domain and it is the maximal element. For any $t = (r, n)$, we write $t = \infty$ iff $n = \infty$ or $r = \infty$. An interval over $\mathcal{T}$ is a pair of time points $[t_1, t_2]$ of $\mathcal{T}$, where $t_1 \neq \infty$ and $t_1 \leq t_2$. A model σ is a tuple $((\mathcal{T}, <), \mathcal{I}, [t_1, t_2])$, where

- $\mathcal{I}$: $SVar \to \mathcal{T} \to \mathsf{Values}$
- $[t_1, t_2]$ is an interval over $\mathcal{T}$.

A model will often be simply written as $(\mathcal{T}, \mathcal{I}, [t_1, t_2])$ with the order omitted. We define WDCI terms and formulae as follows. The durations of a state expression along real time and casual order are two terms denoted respectively by $\int S$ and $\hat{\int} S$. Let

$$[\![S]\!](\mathcal{T}, \mathcal{I}, r) \stackrel{\text{def}}{=} \begin{cases} [\![S]\!](\mathcal{T}, \mathcal{I}, (r, n)) & \text{if } \{n | (r, n) \in \mathcal{T}\} \text{ is singleton} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![S]\!](\mathcal{T}, \mathcal{I}, n) \stackrel{\text{def}}{=} \begin{cases} [\![S]\!](\mathcal{T}, \mathcal{I}, (r, n)) & \text{if } (r, n),\ (r, n+1) \in \mathcal{T} \\ 0 & \text{otherwise.} \end{cases}$$

The first definition is well-formed because if $\{n | (r, n) \in \mathcal{T}\}$ is singleton, then obviously $n_1 = n_2$ for any $(r, n_1) \in \mathcal{T}$ and $(r, n_2) \in \mathcal{T}$. The second definition is also well-formed because for any $(r_1, n) \in \mathcal{T}$, $(r_2, n) \in \mathcal{T}$, $(r_1, n+1) \in \mathcal{T}$ and $(r_2, n+1) \in \mathcal{T}$, it is easy to prove $r_1 = r_2$. Define

$$[\![\int S]\!](\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) \stackrel{\text{def}}{=} \int_{r_1}^{r_2} [\![S]\!](\mathcal{T}, \mathcal{I}, r)\, dr$$
$$[\![\hat{\int} S]\!](\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) \stackrel{\text{def}}{=} \sum_{n=n_1}^{n_2 - 1} [\![S]\!](\mathcal{T}, \mathcal{I}, n).$$

The lengths of an interval along real time and casual order are respectively

$$l \stackrel{\text{def}}{=} \int 1$$
$$k \stackrel{\text{def}}{=} \hat{\int} 1.$$

It is easy to prove that their values in a model $(\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)])$ are

$$[\![l]\!](\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) = r_2 \Leftrightarrow r_1$$
$$[\![k]\!](\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) = n_2 \Leftrightarrow n_1.$$

For a state expression S, $\mathbf{b}.S$ and $\mathbf{e}.S$ are two terms, denoting the values of S at the beginning and the end of the interval:

$$[\![\mathbf{b}.S]\!](\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) = [\![S]\!](\mathcal{I}, r_1, n_1)$$
$$[\![\mathbf{e}.S]\!](\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) = [\![S]\!](\mathcal{I}, r_2, n_2).$$

Primitive formulae of WDCI are constructed from terms using comparison operators in arithmetics, such as $<$, $=$ etc, and can be combined by Boolean connectives and modality operators. The chop modality is defined as follows: for any formulae A and B

$$(\mathcal{T}, \mathcal{I}, [(r_1, n_1), (r_2, n_2)]) \models A; B$$

iff there exists $(r,n) \in \mathcal{T}$, such that $(r,n) \neq \infty$, $(\mathcal{T},\mathcal{I},[(r_1,n_1),(r,n)]) \models A$ and $(\mathcal{T},\mathcal{I},[(r,n),(r_2,n_2)]) \models B$, or $(r_2,n_2) = \infty$ and $(\mathcal{T},\mathcal{I},[(r_1,n_1),(r_2,n_2)]) \models A$. Formula A is valid,

$$\models A \quad \text{iff} \quad \text{for any model } \sigma,\ \sigma \models A.$$

We next introduce some derived modalities

$$\begin{aligned}
&\Diamond\!\!\!\!\cdot\, A \stackrel{\text{def}}{=} A;\mathsf{true}\\
&\boxed{\text{i}}A \stackrel{\text{def}}{=} \neg\Diamond\!\!\!\!\cdot\,\neg A\\
&\Diamond A \stackrel{\text{def}}{=} \mathsf{true};A;\mathsf{true}\\
&\Box A \stackrel{\text{def}}{=} \neg\Diamond\neg A.
\end{aligned}$$

A model satisfies $\Diamond\!\!\!\!\cdot\, A$ and $\Diamond A$ if respectively there is a prefix interval and a sub-interval that satisfies A, and a model satisfies $\boxed{\text{i}}A$ and $\Box A$ if respectively all prefix intervals and all sub-intervals satisfy A. In this paper, we assume modalities bound closer that Boolean operators. Let

$$\begin{aligned}
&\mathsf{fin} \stackrel{\text{def}}{=} l < \infty \wedge k < \infty\\
&\widehat{\mathsf{fin}} \stackrel{\text{def}}{=} k < \infty\\
&\mathsf{point} \stackrel{\text{def}}{=} l = 0 \wedge k = 0.
\end{aligned}$$

They characterise intervals which respectively are finite, finite on causal order and points. For a Boolean expression S, define

$$\lceil S\rceil \stackrel{\text{def}}{=} \neg((l > 0 \vee k > 0);(\mathsf{point} \wedge \neg\mathbf{b}.S);(l > 0 \vee k > 0)).$$

This denotes that S holds everywhere inside the interval. Let

$$\begin{aligned}
&\lceil\!\lceil S\rceil\!\rceil \stackrel{\text{def}}{=} \lceil S\rceil \wedge \mathbf{b}.S\\
&\lceil S\rceil\!\rceil \stackrel{\text{def}}{=} \lceil S\rceil \wedge \mathbf{e}.S\\
&\lceil\!\lceil S\rceil\!\rceil \stackrel{\text{def}}{=} \lceil S\rceil \wedge \mathbf{b}.S \wedge \mathbf{e}.S.
\end{aligned}$$

These specify that S holds everywhere inside the interval and in addition that S holds at the beginning, the end, and both the beginning and the end of the interval respectively. Let

$$\begin{aligned}
&\mathsf{dint} \stackrel{\text{def}}{=} l = 0 \wedge k = 1\\
&\mathsf{cint} \stackrel{\text{def}}{=} l > 0 \wedge k = 0\\
&\mathsf{unit} \stackrel{\text{def}}{=} \mathsf{dint} \vee \mathsf{cint}.
\end{aligned}$$

Intervals that satisfy cint and dint are called respectively continuous and discrete, and collectively unit.

The usual theorems of ITL and DC are still valid. For example

$$\begin{aligned}
&(A;B);C \ \Leftrightarrow\ A;(B;C)\\
&(l = m_1 + m_2) \ \Leftrightarrow\ (l = m_1);(l = m_2).
\end{aligned}$$

In addition, below is an induction theorem based on the observation that any finite interval can be chopped into a series of continuous intervals and discrete intervals. This will be used when we prove the soundness of the assumption–commitment rule.

Theorem If $\models R(\mathsf{point})$ and $\models R(X) \Rightarrow R(X \vee X; \mathsf{unit})$, then $\models R(\mathsf{fin})$.

WDCI and in particular its axiomatisation are currently under further investigation.

4 Compositional Verification of Real-time Systems

4.1 Semantics

We consider several commands, which form the basis of a real-time language with shared variables

$$P ::= \mathsf{await}\ B \to \bar{x} := \bar{e} \mid \mathsf{delay}\ r \mid P_1;P_2 \mid \mathsf{if}\ B\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\ \mathsf{fi}$$
$$S ::= P_1 \parallel P_2 \parallel \cdots \parallel P_n.$$

As by usual convention, $\bar{x}$ denotes a vector of variables, and $\bar{e}$ denotes a matching vector of expressions. The guarded assignment $\mathsf{await}\ B \to \bar{x} := \bar{e}$ is blocked until the Boolean condition B becomes true and the assignment is executed taking zero real time and one unit of causal time. Real time will only pass with all the variables unchanged when there are no instantaneous actions enabled.

We give the semantics in WDCI, that is, we map a program to a WDCI formula describing the behaviours of the program. Let $\mathsf{idle}_i \stackrel{\mathrm{def}}{=} \Box(\mathsf{dint} \Rightarrow \mathbf{e}.u \neq i)$, where u is the variable recording the index of the process that has contributed the last discrete transition. The formula says that over the interval the i-th component does not contribute any discrete transitions. The semantics of the statements is as follows, where we assume the assignment, the delay and the conditional statements are from the i-th component:

$$\begin{aligned}
&[\![\ \mathsf{await}\ B \to \bar{x} := \bar{e}\ ]\!] = (\Box(\mathsf{cint} \Rightarrow [\![\neg B]\!]) \wedge \mathsf{idle}_i);\\
&\qquad\qquad (\mathsf{dint} \wedge B(\mathbf{b}.\bar{x}) \wedge \mathbf{e}.\bar{x} = \mathbf{b}.\bar{e} \wedge \mathbf{e}.u = i)\\
&[\![\ \mathsf{delay}\ r\ ]\!] \stackrel{\mathrm{def}}{=} \mathsf{idle}_i \wedge l \leq r \wedge (\widehat{\mathsf{fin}} \Rightarrow l = r)\\
&[\![\ P_1;P_2\ ]\!] \stackrel{\mathrm{def}}{=} [\![\ P_1\ ]\!];[\![\ P_2\ ]\!]\\
&[\![\ \mathsf{if}\ B\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2\ \mathsf{fi}\ ]\!] \stackrel{\mathrm{def}}{=} (l = 0 \wedge \mathsf{idle}_i);(\mathsf{dint} \wedge \mathbf{e}.\bar{x} = \mathbf{b}.\bar{x} \wedge \mathbf{e}.u = i)\\
&\qquad\qquad ;((B(\mathbf{b}.\bar{x}) \wedge [\![\ P_1\ ]\!]) \vee (\neg B(\mathbf{b}.\bar{x}) \wedge [\![\ P_2\ ]\!]))\\
&[\![\ P_1 \parallel P_2 \parallel \cdots \parallel P_n\ ]\!] \stackrel{\mathrm{def}}{=} [\![\ P_1\ ]\!] \wedge [\![\ P_2\ ]\!] \wedge \cdots \wedge [\![\ P_n\ ]\!].
\end{aligned}$$

The sub-formula before the chop in the semantics of the assignment describes the waiting period. In this period, apparently no discrete transitions are from the component, and for any time point over a continuous sub-interval the Boolean guard does not hold. The reason for this is clear, because otherwise the assignment is enabled and one discrete transition should be taken causing the interval

to be non-continuous. For the semantics of the delay statement, it is obvious that there are no discrete transitions from the component and the real-time duration is not greater than r. The reason that we cannot ensure the elapsed real-time is always exactly r is because in our model other components may execute infinite number of discrete transitions before the real time passes – in this case, it is sometimes figuratively said that time has been stopped.

The semantics contains two additional formulae:

$$\Box(\mathsf{cint} \Rightarrow (\mathbf{b}.\bar{x} = \mathbf{e}.\bar{x}))$$

which says that no variables are changed over any continuous intervals, and for a closed system $P_1 \parallel P_2 \parallel \cdots \parallel P_n$

$$\exists 0 \leq i \leq n.\, \Box(\mathsf{dint} \Rightarrow (\mathbf{e}.u = i))$$

which says that any discrete transition is from one of the components.

For technical convenience, we assume each P_i in $P_1 \parallel P_2 \parallel \cdots \parallel P_n$ is infinite. This can be guaranteed by appending an infinite delay statement in the end, but we shall omit such statements in the examples. A command which is notably missing is the iteration statement. The semantics of the iteration statement can be defined using fixed-points, but this involves some technical details which can be found in [25] where we formalise the semantics of a large subset of Verilog, a hardware description language widely used in industry.

4.2 Simple Composition

We consider the verification of a segment of Fisher's mutual exclusion algorithm using the simple composition rule.

Example

P_1 :		P_2 :
delay a_1;		delay a_2;
$x := 1$;		$x := 2$;
delay b_1;		delay b_2;
await $x = 1$;		await $x = 2$;
CS_1		CS_2

The system has two processes and the algorithm ensures that they will not be in their critical sections CS_1 and CS_2 at the same time by appropriate timing parameters. Overloading the notation a little, let CS_i also represent a Boolean state variable whose value is 1 exactly when P_i is in its critical section. Mutual exclusion can be expressed as $[\![\neg(CS_1 \wedge CS_2)]\!]$.

Intuitively, the correctness of the algorithm can be understood as follows. If P_2 ensures it does not change the value of x after b_1 real time units, then it follows that the value of x is 1 when P_1 is in CS_1, since the critical section can only be entered when $x = 1$ and this happens after b_1 time units. Similarly, if P_1 ensures it does not change the value of x after b_2 real time units, then the

value of x is 2 when P_2 is in CS_2. Therefore, if $b_1 > a_2$ and $b_2 > a_1$ then P_1 and P_2 will not be in critical sections simultaneously, for otherwise x would be both 1 and 2 at the same time and this is impossible.

Let formula $\mathsf{keep}(P_2, x)$ denote that over the interval, no transitions from P_2 change x. This could be defined as $\Box(\mathsf{dint} \wedge (\mathbf{b}.x \neq \mathbf{e}.x) \Rightarrow (\mathbf{e}.u \neq P_2))$. Process P_1 satisfies $(A_1 \Rightarrow C_1) \wedge C_1'$ where

$$
\begin{array}{l}
A_1 : \mathbf{b}.\neg CS_1 \wedge ((l = b_1); \mathsf{keep}(P_2, x)) \wedge \mathsf{keep}(P_2, CS_1), \\
C_1 : [\![CS_1 \Rightarrow x = 1]\!], \\
C_1' : ((l = a_1); \mathsf{keep}(P_1, x)) \wedge \mathsf{keep}(P_1, CS_2).
\end{array}
$$

Similarly, process P_2 satisfies $(A_2 \Rightarrow C_2) \wedge C_2'$ where

$$
\begin{array}{l}
A_2 : \mathbf{b}.\neg CS_2 \wedge ((l = b_2); \mathsf{keep}(P_1, x)) \wedge \mathsf{keep}(P_1, CS_2), \\
C_2 : [\![CS_2 \Rightarrow x = 2]\!], \\
C_2' : ((l = a_2); \mathsf{keep}(P_2, x)) \wedge \mathsf{keep}(P_2, CS_1).
\end{array}
$$

Since $b_1 > a_2$ and $b_2 > a_1$, it is easy to prove that $\mathbf{b}.\neg CS_2 \wedge C_1' \Rightarrow A_2$ and $\mathbf{b}.\neg CS_1 \wedge C_2' \Rightarrow A_1$, therefore it follows from the simple composition rule that

$$P_1 \parallel P_2 \ \ \underline{sat} \ \ (\mathbf{b}.\neg CS_1 \wedge \mathbf{b}.\neg CS_2) \Rightarrow (C_1 \wedge C_2)$$

and it is easy to see that

$$C_1 \wedge C_2 \Rightarrow [\![\neg(CS_1 \wedge CS_2)]\!].$$

4.3 Spiral Reasoning

In this section, we formulate the spiral reasoning rule in WDCI. First, we redefine several concepts:

- A formula A is a *safety* property iff $A \Leftrightarrow \boxed{\mathsf{i}}(\mathsf{fin} \Rightarrow \mathsf{A})$ is valid;
- $A \hookrightarrow B \stackrel{\text{def}}{=} \boxed{\mathsf{i}}(\mathsf{fin} \wedge (A; \mathsf{unit}) \Rightarrow B)$;
- A program P satisfies a specification $(A, \ < C^S, C^R >)$, denoted by $P \ \ \underline{sat} \ \ (A, \ < C^S, C^R >)$, if $[\![P]\!] \Rightarrow (A \hookrightarrow C^S) \ \wedge \ (A \Rightarrow C^R)$.

Rule 6

$$
\frac{\begin{array}{l}
\mathsf{point} \wedge A \Rightarrow A_1 \wedge A_2 \\
A \wedge C_1^S \wedge C_2^S \Rightarrow A_1 \wedge A_2 \\
C_1^S \wedge C_2^S \Rightarrow C^S \\
A \wedge C_1^R \wedge C_2^R \Rightarrow C^R \\
P_i \ \ \underline{sat} \ \ (A_i, \ < C_i^S, C_i^R >)
\end{array}}{P_1 \parallel P_2 \ \ \underline{sat} \ \ (A, \ < C^S, C^R >)}
$$

where A, A_i are safety properties.

We prove the soundness of the rule in WDCI. Let Ψ denote the premises of the rule, namely

$$\begin{aligned}\Psi =&(\mathsf{point} \wedge A \Rightarrow A_1 \wedge A_2)\\ &\wedge(A \wedge C_1^S \wedge C_2^S \Rightarrow A_1 \wedge A_2)\\ &\wedge(C_1^S \wedge C_2^S \Rightarrow C^S)\\ &\wedge(A \wedge C_1^R \wedge C_2^R \Rightarrow C^R)\\ &\wedge(P_i \ \ \underline{sat} \ \ (A_i, \ \ < C_i^S, C_i^R >).\end{aligned}$$

Lemma 1. $\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \Rightarrow \boxed{i}(\mathsf{fin} \Rightarrow A_1 \wedge A_2)$.

Proof. We prove by induction. Let

$$R(X) \stackrel{\text{def}}{=} \Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \Rightarrow \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2).$$

Base step:

$$\begin{aligned}&\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A && \{A \text{ is a safety}\}\\ \Rightarrow&\Box\Psi \wedge \boxed{i}(\mathsf{fin} \Rightarrow A) && \{\mathsf{point} \Rightarrow \mathsf{fin} \text{ and WDCI}\}\\ \Rightarrow&\Box\Psi \wedge \boxed{i}(\mathsf{point} \Rightarrow A) && \{\Psi \text{ and WDCI}\}\\ \Rightarrow&\boxed{i}(\mathsf{point} \Rightarrow A_1 \wedge A_2).\end{aligned}$$

Induction step:

$$\begin{aligned}&\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A\\ &\qquad \wedge \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2) \wedge \mathsf{fin} \wedge (X \vee X; \mathsf{unit}) && \{\text{WDCI}\}\\ \Rightarrow&(A_1 \wedge A_2) \vee (\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A\\ &\qquad \wedge \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2) \wedge \mathsf{fin} \wedge (X; \mathsf{unit})) && \{\text{WDCI}\}\\ \Rightarrow&(A_1 \wedge A_2) \vee (\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S)\\ &\qquad \wedge \mathsf{fin} \wedge ((A_1 \wedge A_2); \mathsf{unit})) && \{\text{WDCI}\}\\ \Rightarrow&(A_1 \wedge A_2) \vee (\Box\Psi \wedge A \wedge C_1^S \wedge C_2^S) && \{\text{WDCI}\}\\ \Rightarrow&A_1 \wedge A_2.\end{aligned}$$

Therefore

$$\begin{aligned}&\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \wedge \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2)\\ \Rightarrow&(\mathsf{fin} \wedge (X \vee X; \mathsf{unit}) \Rightarrow (A_1 \wedge A_2))\end{aligned}$$

and it follows from

$$\begin{aligned}&\boxed{i}(\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \wedge \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2))\\ \Leftrightarrow&(\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \wedge \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2))\end{aligned}$$

that

$$\begin{aligned}&(\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \wedge \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2))\\ \Rightarrow&\boxed{i}(\mathsf{fin} \wedge (X \vee X; \mathsf{unit}) \Rightarrow (A_1 \wedge A_2)).\end{aligned}$$

Subsequently

$$\begin{aligned}&(\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \Rightarrow \boxed{i}(\mathsf{fin} \wedge X \Rightarrow A_1 \wedge A_2))\\ \Rightarrow&(\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A\\ &\qquad \Rightarrow \boxed{i}(\mathsf{fin} \wedge (X \vee X; \mathsf{unit}) \Rightarrow (A_1 \wedge A_2)))\end{aligned}$$

that is, $R(X) \Rightarrow R(X \vee X; \mathsf{unit})$. Therefore, by the induction theorem, we have

$$\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge A \Rightarrow \boxed{\mathsf{i}}(\mathsf{fin} \Rightarrow A_1 \wedge A_2).$$

□

Lemma 2. $\Psi \vdash (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \Rightarrow (A \hookrightarrow C^S)$.

Proof.

$$\begin{aligned}
&\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge \mathsf{fin} \wedge (A; \mathsf{unit}) && \{\text{Lemma 1 and WDCI}\}\\
\Rightarrow&\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \wedge \mathsf{fin} \wedge ((A_1 \wedge A_2); \mathsf{unit}) && \{\text{WDCI}\}\\
\Rightarrow&\Box\Psi \wedge C_1^S \wedge C_2^S && \{\Psi \text{ and WDCI}\}\\
\Rightarrow&C^S.
\end{aligned}$$

Therefore

$$\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_2 \hookrightarrow C_2^S) \Rightarrow \boxed{\mathsf{i}}(\mathsf{fin} \wedge (A; \mathsf{unit}) \Rightarrow C^S).$$

□

Lemma 3.

$$\Psi \vdash (A_1 \hookrightarrow C_1^S) \wedge (A_1 \Rightarrow C_1^R) \wedge (A_2 \hookrightarrow C_2^S) \wedge (A_2 \Rightarrow C_2^R) \Rightarrow (A \Rightarrow C^R).$$

Proof.

$$\begin{aligned}
&\Box\Psi \wedge (A_1 \hookrightarrow C_1^S) \wedge (A_1 \Rightarrow C_1^R)\\
&\qquad \wedge (A_2 \hookrightarrow C_2^S) \wedge (A_2 \Rightarrow C_2^R) \wedge A && \{\text{Lemma 1}\}\\
\Rightarrow&\Box\Psi \wedge (A_1 \Rightarrow C_1^R) \wedge (A_2 \Rightarrow C_2^R) \wedge A \wedge \boxed{\mathsf{i}}(\mathsf{fin} \Rightarrow A_1 \wedge A_2) && \{\text{WDCI}\}\\
\Rightarrow&\Box\Psi \wedge (A_1 \Rightarrow C_1^R) \wedge (A_2 \Rightarrow C_2^R) \wedge A\\
&\qquad \wedge \boxed{\mathsf{i}}(\mathsf{fin} \Rightarrow A_1) \wedge \boxed{\mathsf{i}}(\mathsf{fin} \Rightarrow A_2) && \{A_i \text{ is safety}\}\\
\Rightarrow&\Box\Psi \wedge (A_1 \Rightarrow C_1^R) \wedge (A_2 \Rightarrow C_2^R) \wedge A \wedge A_1 \wedge A_2 && \{\text{WDCI}\}\\
\Rightarrow&\Box\Psi \wedge A \wedge C_1^R \wedge C_2^R && \{\text{WDCI}\}\\
\Rightarrow&C^R.
\end{aligned}$$

□

Theorem $\Psi \vdash [\![P_1 \parallel P_2]\!] \Rightarrow (A \hookrightarrow C^S) \wedge (A \Rightarrow C^R)$.

Proof.

$$\begin{aligned}
&\Psi \vdash [\![P_1 \parallel P_2]\!] \Rightarrow [\![P_1]\!] \wedge [\![P_2]\!] && \{\text{Semantics}\}\\
&\Psi \vdash [\![P_1]\!] \wedge [\![P_2]\!]\\
&\quad \Rightarrow (A_1 \hookrightarrow C_1^S) \wedge (A_1 \Rightarrow C_1^R) \wedge (A_2 \hookrightarrow C_2^S) \wedge (A_2 \Rightarrow C_2^R) && \{\text{WDCI}\}\\
&\Psi \vdash [\![P_1 \parallel P_2]\!] \Rightarrow (A \hookrightarrow C^S) \wedge (A \Rightarrow C^R) && \{\text{Lemmata } 2, 3\}.
\end{aligned}$$

□

Example

P_1 :
do
if $x \neq 0$ then $x := 1$
else skip fi;
delay 2
od

$\|$

P_2 :
do
if $x \neq 0$ then $x := 2$
else skip fi;
delay 3
od

If the initial value of x is 0, then the Boolean test in each process always yields the value false and subsequently the statement skip (which can be defined as $x := x$) is selected unless the other process has changed x to a non-zero value. Since no process can change the value first, x remains to be 0 all the time. Formally, we can prove process P_1 satisfies

$$A_1 : [\![x = 0]\!] \vee [\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee \mathbf{e}.u \neq 2)),$$
$$C_1^S : [\![x = 0]\!] \vee [\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee \mathbf{e}.u \neq 1))$$

and P_2 satisfies

$$A_2 : [\![x = 0]\!] \vee [\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee \mathbf{e}.u \neq 1)),$$
$$C_2^S : [\![x = 0]\!] \vee [\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee \mathbf{e}.u \neq 2)).$$

Since

$$\begin{aligned}
& C_1^S \wedge C_2^S \\
\Rightarrow & [\![x = 0]\!] \vee ([\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee \mathbf{e}.u \neq 1)) \\
& \qquad \wedge [\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee \mathbf{e}.u \neq 2))) \\
\Rightarrow & [\![x = 0]\!] \vee [\![x = 0]\!]; (\mathsf{dint} \wedge (\mathbf{e}.x = 0 \vee (\mathbf{e}.u \neq 1 \wedge \mathbf{e}.u \neq 2)) \\
\Rightarrow & [\![x = 0]\!] \vee [\![x = 0]\!]; (\mathsf{dint} \wedge \mathbf{e}.x = 0) \\
\Rightarrow & [\![x = 0]\!].
\end{aligned}$$

It follows from the composition rule that $P_1 \parallel P_2$ satisfies

$$A : \mathbf{b}.x = 0,$$
$$C^S : [\![x = 0]\!].$$

5 Conclusion

In this paper, we have discussed the assumption–commitment paradigm for achieving compositionality. We have surveyed several rules which have been developed in different settings and argue that they are based on the same principle. There has been much related work. Being a topic of extensive research, compositionality has been studied by many researchers, e.g., by Hooman [11], Zwiers [32] and Jonsson [14]. In a closer context, recent work using assumption–commitment paradigm includes those by Jones [13] in object-orientation, by Collette [8] on UNITY, by Jonsson and Tsay [15] on linear-time temporal logic.

By now, the principle of composing assumption–commitment specifications has been well understood. Consequently, it can be applied readily to different

formalisms, and we have done so in this paper for verification of real-time systems using Duration Calculus.

There exists also work which is related but from somewhat different perspectives. Moszkowski studied what kind of formulae can be used as assumptions and commitments in ITL [20]. Shankar [26] proposes a new approach, called lazy composition, but it seems that the mathematical theory of lazy composition is the same as that of assumption–commitment, and his contribution can probably be best viewed as methodological study of when and how to discharge mutually dependent proof obligations.

Acknowledgements The first author thanks Pierre Collette and Antonio Cau for joint work which is partly reviewed here and for many discussions on the assumption–commitment paradigm. We thank Wang Ji for his initial work in embedding the assumption–commitment rule in ITL, and Dang Van Hung, Willem-Paul de Roever, P.K. Pandya and Zhou Chaochen for discussions and useful comments.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of REX Workshop on Real-Time: Theory and Practice, LNCS 600*, pages 1–27, Mook, The Netherlands, 1992. Springer–Verlag.
2. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15:73–132, 1993.
3. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
4. H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Proc. of Seminar on Concurrency, LNCS 197*. Springer–Verlag, 1985.
5. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
6. Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A duration calculus with infinite intervals. In *Fundamentals of Computation Theory, Horst Reichel (Ed.)*, pages 16–41. LNCS 965, Springer-Verlag, 1995.
7. Zhou Chaochen, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid systems. In *Hybrid Systems, R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (Eds.)*, pages 36–59. LNCS 736, Springer-Verlag, 1993.
8. P. Collette. Application of the composition principle to Unity-like specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proc. TAPSOFT 93, LNCS 668*. Springer–Verlag, 1993.

9. P. Collette and A. Cau. Parallel composition of assumption–commitment specifications: a unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 1995.
10. W.-P. de Roever. The quest for compositionality. In *Proc. of IFIP Working Conf., The Role of Abstract Models in Computer Science*. Elsevier Science B.V. (North-Holland), 1985.
11. J. Hooman. *Specification and Compositional Verification of Real-Time Systems, LNCS 558*. Springer–Verlag, 1991.
12. C.B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.
13. C.B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–121, October 1996.
14. B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.
15. B. Jonsson and Y.-K. Tsay. Reasoning about assumption/guarantee specifications in linear-time temporal logic. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proc. TAPSOFT 95, LNCS 915*. Springer–Verlag, 1995.
16. R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. LNCS 651, Springer–Verlag, 1992.
17. Z. Liu, A.P. Ravn, and X.-S. Li. Verifying duration properties of timed transition systems. In *Proc. IFIP Working Conference PROCOMET'98*. Chapman & Hall, 1998.
18. J. Misra and M. Chandy. Proofs of networks of processes. IEEE SE, 7(4):417–426, 1981.
19. B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
20. B. Moszkowski. Some very compositional temporal properties. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 307–326. Elsevier Science B.V. (North-Holland), 1994.
21. B. Moszkowski. Compositional reasoning about projected and infinite time. In *Proc. the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 238–245. IEEE Computer Society Press, 1995.
22. P.K. Pandya and Dang Van Hung. Duration calculus with weakly monotonic time. Technical Report Detfors 6, UNU/IIST, P.O. Box 3058 Macau, 1996.
23. P.K. Pandya and M. Joseph. P-A logic - a compositional proof system for distributed programs. Distributed Computing, 5:37–54, 1991.
24. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logic amd Models of Concurrent Systems, NATO ASI Series*, pages 123–144. Springer-Verlag, 1984.
25. G. Schneider and Q.-W. Xu. Formalising semantics of hardware description language Verilog in duration calculus. Technical report, UNU/IIST Technical Report draft, P.O. Box 3058 Macau, 1998.
26. N. Shankar. Lazy compositional verification. In *this volume*, 1998.
27. C. Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58:347–359, 1988.
28. K. Stølen. A method for the development of totally correct shared-state parallel programs. In J.C.M. Baeten and J.F. Groote, editors, *Proc. 2nd International Conference on Concurrency Theory (CONCUR'91), LNCS 527*, Armsterdam, The Netherlands, 1991. Springer–Verlag.

29. H.-P. Wang and Q.-W. Xu. Infinite duration calculus with fixed-point operators. Technical Report draft, UNU/IIST, P.O.Box 3058, Macau, September 1997.
30. Q.-W. Xu, A. Cau, and P. Collette. On unifying assumption–commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *Proc. 5th International Conference on Concurrency Theory (CONCUR'94), LNCS 836*, pages 267–282, Uppsala, Sweden, August 1994. Springer–Verlag.
31. Q.-W. Xu, W.-P. de Roever, and J.-F. He. The rely–guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing, 9(2):149–174, 1997.
32. J. Zwiers. *Compositionality, Concurrency and Partial Correctness, LNCS 321.* Springer–Verlag, 1989.

An Adequate First Order Interval Logic

Zhou Chaochen[1] and Michael R. Hansen[2]

[1] UNU/IIST, P.O. Box 3058, Macau
zcc@iist.unu.edu
[2] Department of Information Technology, Technical University of Denmark,
DK-2800 Lyngby
mrh@it.dtu.dk

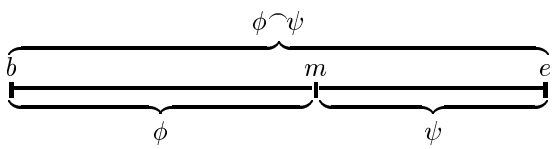
Abstract. This paper introduces *left* and *right* neighbourhoods as primitive interval modalities to define other unary and binary modalities of intervals in a first order logic with interval length. A *complete* first order logic for the neighbourhood modalities is presented.
It is demonstrated how the logic can support formal specification and verification of liveness and fairness, and also of various notions of real analysis.

1 Introduction

Interval temporal logics, based on ITL [6], have shown to be useful for the specification and verification of safety properties of real-time systems. In these logics one can succinctly express properties like: "for all intervals of a given size, ϕ must hold", and "if ϕ holds for an interval, then there is a subinterval where ψ holds", and so on. However, these logics cannot express more abstract liveness properties like "eventually there is an interval where ϕ holds" and "ϕ will hold infinitely often in the future".

The reason for this limitation is that the basic *modality* used in ITL, called *chop* $\frown$, is a *contracting* modality, in the sense that the truth value of a formula $\phi\frown\psi$ on an interval $[b, e]$ only depends on subintervals of $[b, e]$:

$\phi\frown\psi$ holds on $[b, e]$
iff there exists $m \in [b, e]$ such that ϕ holds on $[b, m]$ and ψ holds on $[m, e]$.



The formulas of ITL are constructed from atomic formulas using the connectives of first order logic and the chop modality. It is clear that ITL formulas

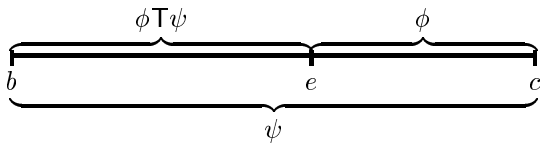
W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 584-608, 1998.
Springer-Verlag Berlin Heidelberg 1998

cannot express abstract liveness properties as the truth value of a formula does not depend on intervals outside the interval under consideration[1].

When logics based on ITL, e.g. [23], are used to specify Hybrid Systems, a limitation is that notions from real analysis, such as limits, continuity, and differentiability, which are definable through the notion *neighbourhood* cannot be formalized in the ITL based logics. E.g. the definition of limit at a point must refer to neighbourhood properties of the point, i.e. properties over super-intervals of the point. To cope with this, an informal mathematical theory of real analysis is assumed in [23] and in other languages for specifying hybrid systems as well, e.g. in Hybrid Statecharts [12], Hybrid Automata [2] and TLA$^+$ [11]. This is fine for verification using paper and pencil; but for proofs of formulas involving the notion neighbourhood, it is impossible to get support from theorem provers for ITL logics.

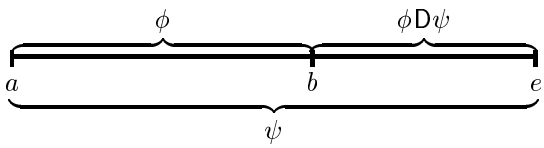
In order to improve the expressiveness of ITL, people have introduced *infinite* intervals [13, 22] and *expanding* modalities [18, 5, 14, 16]. However, all those modalities are a little complicated for different reasons. For example, [18] establishes a complete propositional calculus for three binary interval modalities. In addition to the chop (designated as C in [18]), it introduces two modalities T and D, which are *expanding* in the sense that the truth value of formulas $\phi\mathsf{T}\psi$ and $\phi\mathsf{D}\psi$ on an interval $[b, e]$ depends on intervals "outside" $[b, e]$:

$\phi\mathsf{T}\psi$ holds on $[b, e]$
iff there exists $c \geq e$ such that ϕ holds on $[e, c]$ and ψ holds on $[b, c]$.



So T refers to an expansion of a given interval in future time. Symmetrically, D refers to an expansion in past time.

$\phi\mathsf{D}\psi$ holds on $[b, e]$
iff there exists $a \leq b$ such that ϕ holds on $[a, b]$ and ψ holds on $[a, e]$.



[1] This holds when the truth value of atomic formulas formulas does not depend on "outside intervals".

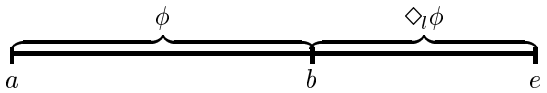
Abstract liveness can be specified using these modalities [16] and there is a complete axiomatization of a propositional modal logic with the three modalities C, T, and D. Some of the axioms and rules for this logic are, however, quite complicated.

Expanding modalities are not necessarily binary, and in [1] there is a lists of thirteen possible unary interval modalities. Furthermore, in [7] it is shown that six of them are basic in the sense that the remaining unary modalities can be derived from the basic ones in propositional logic. Two of the basic modalities are contracting, and four are expanding. Confined to propositional logic, one cannot derive the chop from the thirteen unary modalities [17]. However, this confinement becomes unnecessary, when one uses interval logic to formulate real analysis, since real analysis is traditionally a first order theory.

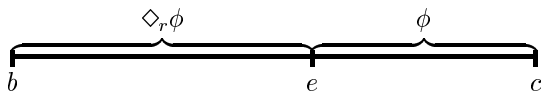
In this paper we present a first order logic for intervals, which has two simple expanding modalities, designated $\Diamond_l \phi$ (reads: "for some left neighbourhood") and $\Diamond_r \phi$ (reads: "for some right neighbourhood"):

$\Diamond_l \phi$ holds on $[b, e]$ iff there exists $\delta \geq 0$ such that ϕ holds on $[b \Leftarrow \delta, b]$, and
$\Diamond_r \phi$ holds on $[b, e]$ iff there exists $\delta \geq 0$ such that ϕ holds on $[e, e + \delta]$.

With $\Diamond_l$ ($\Diamond_r$) one can reach *left* (*right*) neighbourhoods of the beginning (ending) point of an interval:



where $a = b \Leftarrow \delta$



where $c = e + \delta$

When the interval is a *point* interval (i.e. $b = e$ in the definition), they can become the modalities for the conventional left and right neighbourhoods of point, by assuming that the length of the neighbourhoods are non-zero. We therefore call $\Diamond_l$ ($\Diamond_r$) as *left* (*right*) neighbourhood modality. They are expanding modalities, and very similar to $\langle \mathrm{A} \rangle$ and $\langle \bar{\mathrm{A}} \rangle$ of the six basic modalities of [7].

Summary of paper: Syntax and semantics are, in Section 2, given to a first order interval logic based on the two modalities $\Diamond_l$ and $\Diamond_r$. This logic is called *Neighbourhood Logic* (abbreviated to NL). The adequacy of NL is established in Section 3, in the sense that the six basic unary modalities of [7] and the three binary modalities of [17] are expressible in NL. In Section 4 we give a complete axiomatization of NL and some sample proofs. This axiomatization is very similar to the one for ITL in [4]. (We refer to [3] for the proof of the completeness.) This proof system for NL is much simpler than the proof system for the modalities C, T, and D given in [17]. In Section 5 we show that Duration Calculus [21]

can be established as an extension of NL. Furthermore, it is illustrated how abstract liveness and fairness properties and properties of delay insensitive circuits can be specified. Concepts from real analysis are specified in Section 6 and the last section contains a discussion.

2 Syntax and Semantics of NL

2.1 Syntax

The formulas of NL are constructed from the following sets of symbols:

GVar : An infinite set of *global variables* $x, y, z, \ldots$. These variables are called global since their meaning is independent of time and time intervals.

TVar : An infinite set of *temporal variables* $\ell, v, v_1, v_2, v_3, ...$, where ℓ is a special symbol denoting the interval length. The meaning of a temporal variable will be a real valued interval function.

FSymb : An infinite set of *global function symbols* $f^n, g^m, \ldots$ equipped with arities $n, m \geq 0$. If f^n has arity $n = 0$ then f^n is called a *constant*, denoting a real number, such as 0 and 1. The meaning of a global function symbol f^n, $n > 0$, will be an n-ary function on real numbers, which will be independent of time intervals.

RSymb : An infinite set of *global relation symbols* $G^n, H^m, \ldots$ equipped with arities $n, m \geq 0$. If G^n has arity $n = 0$, then G^n is called a *constant*, denoting one of the Boolean value tt or ff. The meaning of a global relation symbol G^n, $n > 0$, will be an n-ary Boolean valued function on real numbers, which will be independent of time intervals.

PLetter : An infinite set of *temporal propositional letters* $X, Y, \ldots$. The meaning of each temporal propositional letter will be a Boolean valued interval function.

The set of *terms*, $\theta, \theta_i \in \mathit{Terms}$, is defined by the abstract syntax:

$$\theta ::= x \mid \ell \mid v \mid f^n(\theta_1, \ldots, \theta_n)$$

The set of *formulas*, $\phi, \psi \in \mathit{Formulas}$, is defined by the abstract syntax:

$$\phi ::= X \mid G^n(\theta_1, \ldots, \theta_n) \mid \neg\phi \mid \phi \vee \psi \mid (\exists x)\phi \mid \Diamond_l \phi \mid \Diamond_r \phi$$

We also use φ, ϕ_i, ψ_i, and φ_i to denote formulas.

We use standard notation for constants, e.g. 0 and 1, and for function and relation symbols of real arithmetic, e.g. $+$ and $\leq$.

2.2 Semantics

The meaning of terms and formulas are explained in this section. To do so, we must explain the meaning of an interval, the meaning of function and relation

symbols, and the meaning of global and temporal variables. Time and time intervals are defined by:

$$\begin{aligned}\mathbb{T}\text{ime} &= \mathbb{R}\\ \mathbb{I}\text{ntv} &= \{[b,e] \mid b,e \in \mathbb{T}\text{ime} \text{ and } b \leq e\}\end{aligned}$$

We consider functions and relations of real arithmetic, so we assume that each n-ary function symbol f_i^n is associated with a function $\underline{f}_i^n \in \mathbb{R}^n \to \mathbb{R}$, and each n-ary relation symbol G_i^n is associated with a function $\underline{G}_i^n \in \mathbb{R}^n \to \{\text{tt,ff}\}$. In particular tt and ff are associated with true and false, respectively. The meaning of function symbols, such as 0, + and $\Leftrightarrow$, is the *standard* one for the real numbers, so is the meaning of the relation symbols $=$, $\geq$ etc.

The meaning of global variables is given by a *value assignment*, which is a function associating a real number with each global variable:

$$\mathcal{V} \in GVar \to \mathbb{R} \ .$$

Let Val be the set of all value assignments.

The meaning of temporal variables and temporal propositional letters, i.e. the "interval dependent symbols", is given by an interpretation:

$$\mathcal{J} \in \begin{pmatrix} TVar \\ \cup \\ PLetters \end{pmatrix} \to \begin{pmatrix} \mathbb{I}\text{ntv} \to \mathbb{R} \\ \cup \\ \mathbb{I}\text{ntv} \to \{\text{tt,ff}\} \end{pmatrix}$$

where $\mathcal{J}(v)([b,e]) \in \mathbb{R}$ and $\mathcal{J}(X)([b,e]) \in \{\text{tt,ff}\}$

associating a real valued interval function with each temporal variable and a Boolean valued interval function with each temporal propositional letter. We will use the following abbreviations:

$$v_{\mathcal{J}} \mathrel{\widehat{=}} \mathcal{J}(v) \text{ and } X_{\mathcal{J}} \mathrel{\widehat{=}} \mathcal{J}(X) \ .$$

The *semantics of a term* θ in an interpretation $\mathcal{J}$ is a function

$$\mathcal{J}[\![\theta]\!] \in Val \times \mathbb{I}\text{ntv} \to \mathbb{R}$$

defined inductively on the structure of terms by:

$$\begin{aligned}\mathcal{J}[\![x]\!](\mathcal{V},[b,e]) &= \mathcal{V}(x)\\ \mathcal{J}[\![\ell]\!]\ (\mathcal{V},[b,e]) &= e \Leftrightarrow b\\ \mathcal{J}[\![v]\!](\mathcal{V},[b,e]) &= v_{\mathcal{J}}([b,e])\\ \mathcal{J}[\![f^n(\theta_1,\ldots,\theta_n)]\!]\ (\mathcal{V},[b,e]) &= \underline{f}^n(c_1,\ldots,c_n)\end{aligned}$$

where $c_i = \mathcal{J}[\![\theta_i]\!]\ (\mathcal{V},[b,e])$, for $1 \leq i \leq n$.

The *semantics of a formula* ϕ in an interpretation $\mathcal{J}$ is a function

$$\mathcal{J}[\![\phi]\!] \in Val \times \mathbb{I}\text{ntv} \to \{\text{tt,ff}\}$$

defined inductively on the structure of formulas below, where the following abbreviations will be used:

$$\mathcal{J},\mathcal{V},[b,e] \models \phi \mathrel{\widehat{=}} \mathcal{J}[\![\phi]\!]\ (\mathcal{V},[b,e]) = \mathrm{tt}$$
$$\mathcal{J},\mathcal{V},[b,e] \not\models \phi \mathrel{\widehat{=}} \mathcal{J}[\![\phi]\!]\ (\mathcal{V},[b,e]) = \mathrm{ff}$$

The definition of $\mathcal{J}[\![\phi]\!]$ is by induction on the structure of formulas:

1) $\mathcal{J},\mathcal{V},[b,e] \models X$ iff $X_{\mathcal{J}}([b,e]) = \mathrm{tt}$

2) $\mathcal{J},\mathcal{V},[b,e] \models G^n(\theta_1,\ldots,\theta_n)$ iff $\underline{G}^n(\mathcal{J}[\![\theta_1]\!](\mathcal{V},[b,e]),\ldots,\mathcal{J}[\![\theta_n]\!](\mathcal{V},[b,e])) = \mathrm{tt}$

3) $\mathcal{J},\mathcal{V},[b,e] \models \neg\phi$ iff $\mathcal{J},\mathcal{V},[b,e] \not\models \phi$

4) $\mathcal{J},\mathcal{V},[b,e] \models \phi \vee \psi$ iff $\mathcal{J},\mathcal{V},[b,e] \models \phi$ or $\mathcal{J},\mathcal{V},[b,e] \models \psi$

5) $\mathcal{J},\mathcal{V},[b,e] \models (\exists x)\phi$ iff $\mathcal{J},\mathcal{V}',[b,e] \models \phi$
for some value assignment $\mathcal{V}'$ which is x-equivalent to $\mathcal{V}$,
i.e. $\mathcal{V}(y) = \mathcal{V}'(y)$ for any global variable $y \neq x$.

6) $\mathcal{J},\mathcal{V},[b,e] \models \Diamond_l\phi$ iff there exists $\delta \geq 0$: $\mathcal{J},\mathcal{V},[b \Leftrightarrow\delta, b] \models \phi$

7) $\mathcal{J},\mathcal{V},[b,e] \models \Diamond_r\phi$ iff there exists $\delta \geq 0$: $\mathcal{J},\mathcal{V},[e, e+\delta] \models \phi$

A formula ϕ is *valid*, written $\models \phi$, iff $\mathcal{J},\mathcal{V},[b,e] \models \phi$, for any interpretation $\mathcal{J}$, value assignment $\mathcal{V}$, and interval $[b,e]$. Furthermore, a formula ψ is *satisfiable* iff $\mathcal{J},\mathcal{V},[b,e] \models \psi$ for some interpretation $\mathcal{J}$, value assignment $\mathcal{V}$, and interval $[b,e]$.

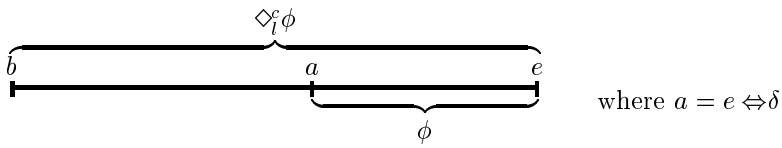
Abbreviations and conventions. We shall use conventional abbreviations for the connectives $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ of propositional logic and the conventional abbreviation for the quantifier $\forall$ of predicate logic. Furthermore, the following abbreviations will be used:

$\Diamond_l^c\phi \mathrel{\widehat{=}} \Diamond_r\Diamond_l\phi$ reads: "for some left neighbourhood of end point: ϕ", and
$\Diamond_r^c\psi \mathrel{\widehat{=}} \Diamond_l\Diamond_r\psi$ reads: "for some right neighbourhood of beginning point: ψ".

The modalities $\Diamond_l^c$ and $\Diamond_r^c$ are called the converses of the modalities $\Diamond_l$ and $\Diamond_r$, respectively.

The following semantic calculations show the meaning of $\Diamond_l^c$:

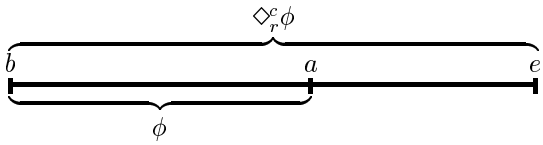
$$\begin{aligned}
&\mathcal{J},\mathcal{V},[b,e] \models \Diamond_l^c\phi \\
&\text{iff } \mathcal{J},\mathcal{V},[b,e] \models \Diamond_r\Diamond_l\phi \\
&\text{iff there exists } \delta' \geq 0\text{: } \mathcal{J},\mathcal{V},[e,e+\delta'] \models \Diamond_l\phi \\
&\text{iff there exists } \delta \geq 0\text{: } \mathcal{J},\mathcal{V},[e \Leftrightarrow\delta, e] \models \phi\ .
\end{aligned}$$



where $a = e \Leftrightarrow\delta$

A similar calculation for $\Diamond_r^c$ will establish that

$$\mathcal{J}, \mathcal{V}, [b, e] \models \Diamond_r^c \psi \text{ iff for some } \delta \geq 0\text{: } \mathcal{J}, \mathcal{V}, [b, b+\delta] \models \psi \ .$$



where $a = b + \delta$

The above figures show the cases where $e \Leftrightarrow \delta > b$ and $b + \delta < e$. However, it is also possible that $e \Leftrightarrow \delta \leq b$ and $b + \delta \geq e$.

When $\neg$, $(\exists x), (\forall x), \Diamond_l, \Diamond_r, \Diamond_l^c$ and $\Diamond_r^c$ occur in formulas they have higher precedence than the binary connectives, e.g. the formula

$$(\Diamond_l^c \phi) \Rightarrow (((\forall x)(\neg \psi)) \wedge \varphi)$$

can be written as:

$$\Diamond_l^c \phi \Rightarrow ((\forall x) \neg \psi \wedge \varphi) \ .$$

Furthermore, the following abbreviations for quantifiers will be used:

$\exists x > \theta.\phi \quad \widehat{=} (\exists x)(x > \theta \wedge \phi)$ similar for other relations $\geq, \leq, \ldots$
$\forall x > \theta.\phi \quad \widehat{=} (\forall x)(x > \theta \Rightarrow \phi)$ similar for other relations $\geq, \leq, \ldots$
$\forall x_1, x_2, \ldots, x_n.\phi \ \widehat{=} (\forall x_1)(\forall x_2) \cdots (\forall x_n)\phi$
$\exists x_1, x_2, \ldots, x_n.\phi \ \widehat{=} (\exists x_1)(\exists x_2) \cdots (\exists x_n)\phi$

3 Adequacy of Neighbourhood Modalities

In this section, we show that the six basic unary interval modalities of [7] and the three binary interval modalities (i.e. $\frown$, T and D) of [18] can be defined in NL.

The six basic modalities of [7] is denoted by:

Modality	Intervals, reachable from "current interval":
$\langle \mathrm{A} \rangle$	non-point right neighbourhoods
$\langle \bar{\mathrm{A}} \rangle$	non-point left neighbourhoods
$\langle \mathrm{B} \rangle$	strict prefix intervals
$\langle \bar{\mathrm{B}} \rangle$	intervals, which have current interval as a strict prefix
$\langle \mathrm{E} \rangle$	strict suffix intervals
$\langle \bar{\mathrm{E}} \rangle$	intervals, which have current interval as a strict suffix

The meaning of these six unary modalities and the three binary modalities $\frown$, T and D is given by:

1. $\mathcal{J}, \mathcal{V}, [b, e] \models \langle \mathrm{A} \rangle \phi$ iff there exists $a > e$: $\mathcal{J}, \mathcal{V}, [e, a] \models \phi$
2. $\mathcal{J}, \mathcal{V}, [b, e] \models \langle \bar{\mathrm{A}} \rangle \phi$ iff there exists $a < b$: $\mathcal{J}, \mathcal{V}, [a, b] \models \phi$
3. $\mathcal{J}, \mathcal{V}, [b, e] \models \langle \mathrm{B} \rangle \phi$ iff there exists a such that $b \leq a < e$ and $\mathcal{J}, \mathcal{V}, [b, a] \models \phi$
4. $\mathcal{J}, \mathcal{V}, [b, e] \models \langle \bar{\mathrm{B}} \rangle \phi$ iff there exists $a > e$: $\mathcal{J}, \mathcal{V}, [b, a] \models \phi$
5. $\mathcal{J}, \mathcal{V}, [b, e] \models \langle \mathrm{E} \rangle \phi$ iff there exists a such that $b < a \leq e$ and $\mathcal{J}, \mathcal{V}, [a, e] \models \phi$
6. $\mathcal{J}, \mathcal{V}, [b, e] \models \langle \bar{\mathrm{E}} \rangle \phi$ iff there exists $a < b$: $\mathcal{J}, \mathcal{V}, [a, e] \models \phi$
7. $\mathcal{J}, \mathcal{V}, [b, e] \models \phi \frown \psi$
   iff there exists $m \in [b, e]$: $\mathcal{J}, \mathcal{V}, [b, m] \models \phi$ and $\mathcal{J}, \mathcal{V}, [m, e] \models \psi$
8. $\mathcal{J}, \mathcal{V}, [b, e] \models \phi \mathsf{T} \psi$ iff there exists $a \geq e$: $\mathcal{J}, \mathcal{V}, [e, a] \models \phi$ and $\mathcal{J}, \mathcal{V}, [b, a] \models \psi$
9. $\mathcal{J}, \mathcal{V}, [b, e] \models \phi \mathsf{D} \psi$ iff there exists $a \leq b$: $\mathcal{J}, \mathcal{V}, [a, b] \models \phi$ and $\mathcal{J}, \mathcal{V}, [a, e] \models \psi$

Theorem 1. *(Adequacy.) The above nine modalities can be expressed in NL.*

Proof. The following equivalences establish the theorem. Each of them is easily checked using the semantic definitions.

1. $\langle \mathrm{A} \rangle \phi \Leftrightarrow \Diamond_r((\ell > 0) \wedge \phi)$
   where $(\ell > 0)$ guarantees that the right expansion is a *non-point* interval.
2. $\langle \bar{\mathrm{A}} \rangle \psi \Leftrightarrow \Diamond_l((\ell > 0) \wedge \psi)$
   where $(\ell > 0)$ guarantees that the left expansion is a *non-point* interval.
3. $\langle \mathrm{B} \rangle \phi \Leftrightarrow \exists x.((\ell = x) \wedge \Diamond_r^c((\ell < x) \wedge \phi))$
   where $\Diamond_r^c$ defines an interval that has the same beginning point as the original interval, and $(\ell < x)$ stipulates that the defined interval is a strict subinterval of the original one.
4. $\langle \bar{\mathrm{B}} \rangle \phi \Leftrightarrow \exists x.((\ell = x) \wedge \Diamond_r^c((\ell > x) \wedge \phi))$
   It is similar to $\langle \mathrm{B} \rangle \phi$, except that $(\ell > x)$ stipulates that the defined interval is a strict super-interval of the original one.
5. $\langle \mathrm{E} \rangle \phi \Leftrightarrow \exists x.((\ell = x) \wedge \Diamond_l^c((\ell < x) \wedge \phi))$
   The definition is similar to $\langle \mathrm{B} \rangle \phi$, except that $\Diamond_l^c$ here defines an interval that has the same ending point as the original interval.
6. $\langle \bar{\mathrm{E}} \rangle \phi \Leftrightarrow \exists x.((\ell = x) \wedge \Diamond_l^c((\ell > x) \wedge \phi))$
   It is similar to $\langle \mathrm{E} \rangle \phi$, except that $(\ell > x)$ stipulates that the defined interval is a strict super-interval of the original one.
7. $\phi \frown \psi \Leftrightarrow \exists x, y.((\ell = x + y) \wedge \Diamond_r^c((\ell = x) \wedge \phi \wedge \Diamond_r((\ell = y) \wedge \psi)))$
   where $(\ell = x + y)$ stipulates that the two consecutive right expansions of lengths x and y exactly cover the original interval.
8. $\phi \mathsf{T} \psi \Leftrightarrow \exists x, y.((\ell = x) \wedge \Diamond_r((\ell = y) \wedge \phi \wedge \Diamond_l^c((\ell = x + y) \wedge \psi)))$
   where $(\ell = x + y)$ guarantees that the left expansion, $\Diamond_l^c$, exactly covers the original interval and its right expansion, $\Diamond_r$.
9. $\phi \mathsf{D} \psi \Leftrightarrow \exists x, y.((\ell = x) \wedge \Diamond_l((\ell = y) \wedge \phi \wedge \Diamond_r^c((\ell = x + y) \wedge \psi)))$
   where $(\ell = x + y)$ guarantees that the right expansion of $\Diamond_r^c$ exactly covers the original interval and its left expansion, $\Diamond_l$.

4 Proof System

In this section we present a proof system for NL. There is a completeness result [3] for this proof systems with respect to a class of interval models which contains the real intervals. In the last subsection we discuss this result.

In the following axiom and rule schemas, $\Diamond$ is a parameter, which can be instantiated by either $\Diamond_l$ or $\Diamond_r$. As usual when instantiating a schema, the instantiation must be consistent for all occurrences of $\Diamond$ in the schema. Moreover, we adopt the abbreviations:

$$\begin{aligned}
\bar{\Diamond} &\mathrel{\widehat{=}} \begin{cases} \Diamond_r, \text{ if } \Diamond = \Diamond_l \\ \Diamond_l, \text{ if } \Diamond = \Diamond_r \end{cases} \\
\Box &\mathrel{\widehat{=}} \neg\Diamond\neg \\
\bar{\Box} &\mathrel{\widehat{=}} \neg\, \bar{\Diamond}\, \neg \\
\Diamond^c &\mathrel{\widehat{=}} \bar{\Diamond}\, \Diamond
\end{aligned}$$

To formulate the axioms and inference rules, we need the standard notion of free (global) variables. Moreover, a term is called *rigid* if it does not contain any temporal variable and a formula is called *modality free* if neither $\Diamond_l$ nor $\Diamond_r$ occur in the formula. A formula is called *rigid* if it is modality free, and contains only rigid terms and no temporal propositional letter.

The axiom schemas of NL are:

Rigid formulas are not connected to intervals:

A1 $\quad \Diamond\phi \;\Rightarrow\; \phi$, provided ϕ is rigid

Interval length is non-negative:

A2 $\quad \ell \geq 0$

Neighbourhoods can be of arbitrary lengths:

A3 $\quad (x \geq 0) \;\Rightarrow\; \Diamond(\ell = x)$

Neighbourhood modalities distribute over disjunction and existential quantification:

A4 $\quad \begin{array}{l} \Diamond(\phi \vee \psi) \Rightarrow \Diamond\phi \;\vee\; \Diamond\psi \\ \Diamond\exists x.\phi \quad\;\; \Rightarrow \exists x.\Diamond\phi \end{array}$

Neighbourhood is determined by its length:

A5 $\quad \Diamond((\ell = x) \wedge \phi) \;\Rightarrow\; \Box((\ell = x) \Rightarrow \phi)$

Left (right) neighbourhoods of an interval always start at the same point:

A6 $\quad \Diamond\, \bar{\Diamond}\, \phi \;\Rightarrow\; \Box\, \bar{\Diamond}\, \phi$

Left (right) neighbourhood of the ending (beginning) point of an interval can be the same interval, if they have the same length:

A7 $\quad (\ell = x) \;\Rightarrow\; (\phi \Leftrightarrow \Diamond^c((\ell = x) \wedge \phi))$

Two consecutive left (right) expansions can be replaced by a single left (right) expansion, if the length of the expansion is the sum of the two formers:

A8 $\quad ((x \geq 0) \wedge (y \geq 0))$
$\Rightarrow\; (\Diamond((\ell = x) \wedge \Diamond((\ell = y) \wedge \Diamond\phi)) \;\Leftrightarrow\; \Diamond((\ell = x + y) \wedge \Diamond\phi))$

The rule schemas of NL are:

M	If $\phi \;\Rightarrow\; \psi$ then $\Diamond\phi \;\Rightarrow\; \Diamond\psi$	(Monotonicity)
N	If ϕ then $\Box\phi$	(Necessity)
MP	If ϕ and $\phi \;\Rightarrow\; \psi$ then ψ	(Modus Ponens)
G	If ϕ then $(\forall x)\phi$	(Generalization)

The Monotonicity and Necessity rules are taken from modal logic and the Modus Ponens and Generalization rules are taken from first order predicate logic.

The proof system also contains axioms of first order predicate logic with equality as well as first order theory of real numbers. Any axiomatic basis can be chosen, and we will use "PL" when we refer to axioms, theorems and inference rules of the first order predicate logic and the first order theory of real numbers. Special care must, however, be taken when universally quantified formulas are instantiated and when an existential quantifier is introduced.

To formulate axiom schemas for quantification we define: A term θ is called *free for* x in ϕ if x does not occur freely in ϕ within a scope of $\exists y$ or $\forall y$, where y is any variable occurring in θ.

The following axiom schemas are sound:

$$\begin{array}{l} \forall x.\phi(x) \;\Rightarrow\; \phi(\theta) \\ \phi(\theta) \;\Rightarrow\; \exists x.\phi(x) \end{array} \qquad \left(\begin{array}{l} \text{if either } \theta \text{ is free for } x \text{ in } \phi(x) \text{ and } \theta \text{ is rigid} \\ \text{or } \theta \text{ is free for } x \text{ in } \phi(x) \text{ and } \phi(x) \text{ is modality free.} \end{array} \right)$$

A *proof* in NL of ϕ is a finite sequence of formulas $\phi_1 \cdots \phi_n$, where ϕ_n is ϕ, and each ϕ_i is either an instance of one of the above axiom schemas or obtained by applying one of the above inference rules to previous members of the sequence. We write $\vdash \phi$ to denote that there exists a proof of ϕ and call ϕ a *theorem* of NL.

A *deduction* of ϕ from a set of formulas Γ (called *assumptions*) is a finite sequence of formulas $\phi_1 \cdots \phi_n$, where ϕ_n is ϕ, and each ϕ_i is either a member of Γ, an instance of one of the above axiom schemas or obtained by applying one of the above inference rules to previous members of the sequence. We write $\Gamma \vdash \phi$ to denote that there exists a deduction of ϕ from Γ.

Properties about Neighbourhood Logic are another kind of theorems also called meta-theorems. One such example is the soundness of the proof system:

Theorem 2. *(Soundness.)*

$$\textit{if } \vdash \phi \textit{ then } \models \phi$$

Proof. A proof of the soundness theorem can be given by proving that each axiom is sound and that each inference rule preserves soundness. In [19], NL is encoded in PVS and the soundness of NL is checked by PVS.

4.1 Theorems of NL

We list and give proofs of a set of theorems of NL. These theorems are used in the completeness proof of NL and they can also help us to understand the calculus.

We will denote theorems and deductions in NL by T1, T2, etc., to distinguish them from meta-theorems.

The first deduction to be derived is the monotonicity of $\Box$:

T1 $\quad \phi \Rightarrow \psi \vdash \Box\phi \Rightarrow \Box\psi$

The following deduction establishes T1:

1. $\phi \Rightarrow \psi$	assumption
2. $\neg\psi \Rightarrow \neg\phi$	1., PL
3. $\Diamond\neg\psi \Rightarrow \Diamond\neg\phi$	2., M
4. $\neg\Diamond\neg\phi \Rightarrow \Diamond\neg\psi$	3., PL

T2 *a.* $\Diamond\text{true}$ where $\text{true} \mathrel{\widehat{=}} (0 = 0)$
b. $\Diamond\text{false} \Rightarrow \text{false}$ where $\text{false} \mathrel{\widehat{=}} \neg\text{true}$

The following is a proof for T2*a*:

1. $(0 \geq 0) \Rightarrow \Diamond(\ell = 0)$	A3
2. $\Diamond(\ell = 0)$	PL$(0 \geq 0)$, 1., MP
3. $\Diamond\text{true}$	M

The second part, T2b, is an instance of A1.

The following theorems express together with A4 that $\Diamond$ commutes with disjunction and existential quantification:

T3 *a.* $(\Diamond\phi \vee \Diamond\psi) \Rightarrow \Diamond(\phi \vee \psi)$
b. $\exists x.\Diamond\phi \Rightarrow \Diamond\exists x.\phi$

Proof for T3*a*:

1. $\phi \Rightarrow (\phi \vee \psi)$	PL
2. $\Diamond\phi \Rightarrow \Diamond(\phi \vee \psi)$	1., M
3. $\psi \Rightarrow (\phi \vee \psi)$	PL
4. $\Diamond\psi \Rightarrow \Diamond(\phi \vee \psi)$	3., M
5. $(\Diamond\phi \vee \Diamond\psi) \Rightarrow \Diamond(\phi \vee \psi)$	2., 4., PL

Proof for T3*b*:

1. $\phi \Rightarrow \exists x.\phi$	PL
2. $\Diamond\phi \Rightarrow \Diamond\exists x.\phi$	1., M
3. $\forall x.(\Diamond\phi \Rightarrow \Diamond\exists x.\phi)$	2., G
4. $\forall x.(\Diamond\phi \Rightarrow \Diamond\exists x.\phi) \Rightarrow (\exists x.\Diamond\phi \Rightarrow \Diamond\exists x.\phi)$	PL, x is not free in $\Diamond\exists x.\phi$
5. $\exists x.\Diamond\phi \Rightarrow \Diamond\exists x.\phi$	3., 4., MP

We will use the following convention for presenting proof:

$$\begin{array}{ll} \phi_1 & \\ \Rightarrow \phi_2 & \text{is an abbreviation for the proof:} \\ \Rightarrow \phi_3 & \end{array} \qquad \begin{array}{ll} 1.\ \phi_1 \Rightarrow \phi_2 & \\ 2.\ \phi_2 \Rightarrow \phi_3 & \\ 3.\ \phi_1 \Rightarrow \phi_3 & 1., 2., \text{PL} \end{array}$$

and

$$\begin{array}{ll} \phi_1 & \\ \Leftrightarrow \phi_2 & \text{is an abbreviation for the proof:} \\ \Leftrightarrow \phi_3 & \end{array} \qquad \begin{array}{ll} 1.\ \phi_1 \Leftrightarrow \phi_2 & \\ 2.\ \phi_2 \Leftrightarrow \phi_3 & \\ 3.\ \phi_1 \Leftrightarrow \phi_3 & 1., 2., \text{PL} \end{array}$$

This generalizes to longer chains: $\phi_1 \Rightarrow \cdots \Rightarrow \phi_n$ and $\phi_1 \Leftrightarrow \cdots \Leftrightarrow \phi_n$.

T4
a. $\Box\phi \Rightarrow \Diamond\phi$
b. $(\Diamond\phi \wedge \Box\psi) \Rightarrow \Diamond(\phi \wedge \psi)$
c. $(\Box\phi \wedge \Box\psi) \Leftrightarrow \Box(\phi \wedge \psi)$

We only present proofs for the first two parts. Proof for T4*a*:

$\Box\phi$	
$\Rightarrow \Diamond(\phi \vee \neg\phi)$	T2*a*., PL
$\Rightarrow \Diamond\phi \vee \Diamond\neg\phi$	A4
$\Rightarrow \Diamond\phi$	Def.$\Box$, PL

Proof for T4*b*:

$\Diamond\phi \wedge \Box\psi$	
$\Rightarrow \Diamond((\phi \wedge \psi) \vee (\phi \wedge \neg\psi)) \wedge \Box\psi$	PL, M
$\Rightarrow (\Diamond(\phi \wedge \psi) \vee \Diamond(\phi \wedge \neg\psi)) \wedge \Box\psi$	A4
$\Rightarrow (\Diamond(\phi \wedge \psi) \wedge \Box\psi) \vee (\Diamond\neg\psi \wedge \Box\psi)$	PL, M
$\Rightarrow \Diamond(\phi \wedge \psi)$	PL, Def.$\Box$

T5
a. $\phi \Rightarrow \Diamond^c\phi$
b. $\bar{\Diamond}^c\Diamond\phi \Leftrightarrow \Diamond\phi$
c. $(\Diamond\phi \wedge \bar{\Diamond}^c\psi) \Leftrightarrow \Diamond(\phi \wedge \bar{\Diamond}\,\psi)$

Proof for T5a, where we assume that x is not free in ϕ:

$$
\begin{array}{lll}
1. & (\ell = x) \wedge \phi & \\
& \Rightarrow \Diamond^c((\ell = x) \wedge \phi) & \text{A7} \\
& \Rightarrow \Diamond^c \phi & \text{PL, M} \\
2. & \phi & \\
& \Rightarrow \exists x.(\ell = x) \wedge \phi & \text{PL} \\
& \Rightarrow \exists x.(\ell = x \wedge \phi) & \text{PL, } x \text{ not free in } \phi \\
3. & \phi \ \Rightarrow \ \Diamond^c \phi & 1., 2., \text{PL}(\exists_)
\end{array}
$$

Proof for T5b. The direction $\Leftarrow$ follows from T5a and M. The direction $\Rightarrow$ is established by:

$$
\begin{array}{lll}
1. & \bar{\Diamond}^c \Diamond \phi & \\
& \Rightarrow \Box\ \bar{\Diamond}\ \Diamond \phi & \text{A6 } (\bar{\Diamond}^c \Diamond = \Diamond\ \bar{\Diamond}\ \Diamond) \\
& \Rightarrow \Box\ \bar{\Box}\ \Diamond \phi & \text{A6, T1} \\
2. & \bar{\Diamond}^c \Diamond \phi\ \wedge\ \neg \Diamond \phi & \\
& \Rightarrow \Box\ \bar{\Box}\ \Diamond \phi \wedge\ \bar{\Diamond}^c \neg \Diamond \phi & 1., \text{T5}a, \text{PL} \\
& \Rightarrow \Diamond(\bar{\Box}\ \Diamond \phi \wedge\ \bar{\Diamond}\ \neg \Diamond \phi) & \text{T4}b(\bar{\Diamond}^c = \Diamond\ \bar{\Diamond}) \\
& \Rightarrow \Diamond\ \bar{\Diamond}\ (\Diamond \phi\ \wedge\ \neg \Diamond \phi) & \text{T4}b, \text{M} \\
& \Rightarrow \text{false} & \text{PL, T2}b, \text{M} \\
3. & \bar{\Diamond}^c \Diamond \phi\ \Rightarrow\ \Diamond \phi & 2., \text{PL}
\end{array}
$$

Proof for T5c. The direction $\Leftarrow$ follows from PL and M. The direction $\Rightarrow$ is established by:

$$
\begin{array}{ll}
& \Diamond \phi \wedge\ \bar{\Diamond}^c \psi \\
\Rightarrow & \Diamond \phi\ \wedge\ \Box\ \bar{\Diamond}\ \psi \text{ A6, PL} \\
\Rightarrow & \Diamond(\phi \wedge\ \bar{\Diamond}\ \psi)\ \ \text{T4}b
\end{array}
$$

T6

$a.\ \Diamond\ \bar{\Box}\ \phi\ \Rightarrow\ \Box\ \bar{\Box}\ \phi$

$b.\ \bar{\Diamond}^c \Box \phi\ \Leftrightarrow\ \Box \phi$

$c.\ (\Diamond \phi\ \wedge\ \Diamond\ \bar{\Box}\ \psi)\ \Leftrightarrow\ \Diamond(\phi \wedge\ \bar{\Box}\ \psi)$

The proof for these theorems are similar to those for T5.

T7 $\quad (\Diamond((\ell = x) \wedge \phi)\ \wedge \Diamond((\ell = x) \wedge \psi))\ \Rightarrow\ \Diamond((\ell = x) \wedge \phi \wedge \psi)$

Proof for T7:

$$
\begin{array}{ll}
& \Diamond((\ell = x) \wedge \phi)\ \wedge \Diamond((\ell = x) \wedge \psi) \\
\Rightarrow & \Box((\ell = x) \Rightarrow \phi)\ \wedge\ \Box((\ell = x) \Rightarrow \psi)\ \wedge \Diamond(\ell = x) \quad \text{A5, M, PL} \\
\Rightarrow & \Box((\ell = x) \Rightarrow (\phi \wedge \psi)) \wedge \Diamond(\ell = x) \quad \text{T4}c, \text{PL} \\
\Rightarrow & \Diamond((\ell = x) \wedge ((\ell = x) \Rightarrow (\phi \wedge \psi))) \quad \text{T4}b, \text{PL} \\
\Rightarrow & \Diamond((\ell = x) \wedge \phi \wedge \psi) \quad \text{PL, M}
\end{array}
$$

If $(x \geq 0) \wedge (y \geq 0)$ then the following formulas are theorems[2]:

$a.\ (\ell = x) \Rightarrow (\Diamond((\ell = y) \wedge \Diamond\phi) \ \Leftrightarrow\ \Diamond^c((\ell = x + y) \wedge \Diamond\phi))$

$b.\ (\ell = x) \Rightarrow (\Diamond((\ell = y) \wedge \phi) \ \Leftrightarrow\ \Diamond^c((\ell = x + y) \wedge \bar{\Diamond}^c((\ell = y) \wedge \phi)))$

$c.\ \Diamond((\ell = x) \wedge \Diamond((\ell = y) \wedge \phi)) \ \Leftrightarrow\ \Diamond((\ell = x + y) \wedge \bar{\Diamond}^c((\ell = y) \wedge \phi))$

T8

$$d.\ (y \geq x) \Rightarrow \left(\begin{array}{l} \bar{\Diamond}^c((\ell = x) \wedge \Diamond^c((\ell = y) \wedge \Diamond\phi)) \\ \Leftrightarrow \Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi) \end{array}\right)$$

$$e.\ (y \geq x) \Rightarrow \left(\begin{array}{l} \bar{\Diamond}^c((\ell = x) \wedge \Diamond^c((\ell = y) \wedge \phi)) \Leftrightarrow \\ \Diamond((\ell = y \Leftrightarrow x) \wedge \bar{\Diamond}^c((\ell = y) \wedge \phi)) \end{array}\right)$$

Proof for T8a:

$$\begin{array}{ll} & \ell = x \\ \Rightarrow & \Diamond((\ell = y) \wedge \Diamond\phi) \Leftrightarrow \ \Diamond^c((\ell = x) \wedge \Diamond((\ell = y) \wedge \Diamond\phi)) & \text{A7} \\ \Rightarrow & \Diamond((\ell = y) \wedge \Diamond\phi) \Leftrightarrow \ \Diamond^c((\ell = x + y) \wedge \Diamond\phi) & \text{A8, M, PL} \end{array}$$

Proof for T8b:

$$\begin{array}{ll} & \ell = x \\ \Rightarrow & \Diamond((\ell = y) \wedge \phi) \Leftrightarrow \ \Diamond((\ell = y) \wedge \bar{\Diamond}^c((\ell = y) \wedge \phi)) & \text{A7, M} \\ \Rightarrow & \Diamond((\ell = y) \wedge \phi) \Leftrightarrow \ \Diamond^c((\ell = x + y) \wedge \bar{\Diamond}^c((\ell = y) \wedge \phi)) & \text{T8}a\text{, PL} \end{array}$$

We give a proof for d, leaving the proofs for c and e for the reader.

Proof for T8d: Assume $y \geq x \geq 0$:

1. $\bar{\Diamond}^c((\ell = x) \wedge \Diamond^c((\ell = y) \wedge \Diamond\phi))$
   $\Leftrightarrow \bar{\Diamond}^c((\ell = x) \wedge \Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi))$ T8$a(y = x + (y \Leftrightarrow x))$, M
2. $\bar{\Diamond}^c((\ell = x) \wedge \Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi))$
   $\Rightarrow \bar{\Diamond}^c(\ell = x) \wedge \bar{\Diamond}^c\Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi)$ M, PL
   $\Rightarrow \Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi)$ T5b, PL
3. $\text{true} \ \Rightarrow\ \bar{\Diamond}\ (\ell = x)$ PL, A3
4. $\Diamond\text{true} \ \Rightarrow\ \Diamond\ \bar{\Diamond}\ (\ell = x)$ 3., M
5. $\bar{\Diamond}^c(\ell = x)$ 4., T2a, MP
6. $\Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi))$
   $\Rightarrow \bar{\Diamond}^c(\ell = x) \wedge \bar{\Diamond}^c\Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi))$ 5., T5a, PL
   $\Rightarrow \Diamond(\bar{\Diamond}\ (\ell = x) \wedge \Diamond^c((\ell = y \Leftrightarrow x) \wedge \Diamond\phi))$ T5$c(\bar{\Diamond}^c\Diamond = \Diamond\ \bar{\Diamond}\ \Diamond = \Diamond\Diamond^c)$
   $\Rightarrow \Diamond\ \bar{\Diamond}\ ((\ell = x) \wedge \Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi))$ T5c, M
7. $\bar{\Diamond}^c((\ell = x) \wedge \Diamond^c((\ell = y) \wedge \Diamond\phi)) \Leftrightarrow \Diamond((\ell = y \Leftrightarrow x) \wedge \Diamond\phi)$ 1., 2., 6., PL

[2] Here, by 'If ϕ then ψ is a theorem', we mean: $\vdash \phi \Rightarrow \psi$.

T9 $a.\ (\ell = 0) \ \Rightarrow\ (\phi \ \Leftrightarrow\ \Diamond((\ell = 0) \wedge \phi))$
$b.\ \Diamond\phi \ \Leftrightarrow\ \Diamond((\ell = 0) \wedge \Diamond\phi)$

Proof for T9a:

$$
\begin{array}{ll}
\ell = 0 & \\
\Rightarrow \Diamond((\ell = 0) \wedge \phi) \Leftrightarrow\ \Diamond^c((\ell = 0) \wedge \bar{\Diamond}^c((\ell = 0) \wedge \phi)) & \text{T8}b(0 + 0 = 0) \\
\Rightarrow \Diamond((\ell = 0) \wedge \phi) \Leftrightarrow \bar{\Diamond}^c((\ell = 0) \wedge \phi) & \text{A7} \\
\Rightarrow \Diamond((\ell = 0) \wedge \phi) \Leftrightarrow\ \phi & \text{A7}
\end{array}
$$

Proof for T9b, where we assume that y is not free in ϕ.

$$
\begin{array}{ll}
\Diamond((\ell = 0) \wedge \Diamond\phi) & \\
\Leftrightarrow \Diamond((\ell = 0) \wedge \Diamond(\exists y \geq 0.(\ell = y) \wedge \phi)) & \text{PL,A2,M} \\
\Leftrightarrow \exists y \geq 0.\Diamond((\ell = 0) \wedge \Diamond((\ell = y) \wedge \phi)) & \text{A4, PL, T3}b \\
\Leftrightarrow \exists y \geq 0.\Diamond((\ell = 0 + y) \wedge \bar{\Diamond}^c((\ell = y) \wedge \phi)) & \text{T8}c\text{, PL} \\
\Leftrightarrow \exists y \geq 0.\Diamond((\ell = y) \wedge \phi) & \text{A7}(0 + y = y)\text{, PL, M} \\
\Leftrightarrow \Diamond\phi & \text{PL, M, A4, A2, T3}b
\end{array}
$$

A deduction theorem can be proved for NL similar to the deduction theorem for Interval Temporal Logic [8]. The following abbreviation is useful to formulate the theorem:

$$\Box_a\psi \mathrel{\widehat{=}} \Box_r\Box_r\Box_\ell\Box_\ell\psi \text{ reads: "for all intervals: } \psi\text{"}$$

Theorem 3. *(Deduction.)*
If a deduction $\Gamma, \phi \vdash \psi$, involves no application of the generalization rule G of which the quantified variable is free in ϕ, then $\Gamma \vdash \Box_a\phi \Rightarrow \psi$

Proof. See [15].

Furthermore, many interesting theorems of NL are proved in [15], e.g. that A1 is true for formulas ϕ which only contain rigid terms, but may not be modality free.

4.2 A completeness result for NL

So far, real numbers ($\mathbb{R}$) have been used as the domain of time and values. It is well-known that it is impossible to have a complete axiomatization for the real numbers. One can develop different first order theories for the real numbers, but none of them can become complete. Hence no one can develop a consistent calculus which can prove any valid formula of NL, when real numbers are taken as time and value domain. However, given a first order theory of the real numbers, denoted $\mathcal{A}$, we can investigate a completeness of the calculus with respect to a notion of $\mathcal{A}$-*validity*. Roughly speaking, a formula is $\mathcal{A}$-valid, if it is valid in models with time and value domains that satisfy $\mathcal{A}$.

A first order theory $\mathcal{A}$ is called a first order theory of the real numbers, if $\mathcal{A}$ includes the following axioms:

D1 Axioms for $=$:

1. $x = x$
2. $(x = y) \Rightarrow (y = x)$
3. $((x = y) \wedge (y = z)) \Rightarrow (x = z)$
4. $((x_1 = y_1) \wedge ... \wedge (x_n = y_n)) \Rightarrow (f^n(x_1, ..., x_n) = f(y_1, ..., y_n))$
   where f^n is an n-ary function symbol.
5. $((x_1 = y_1) \wedge ... \wedge (x_n = y_n)) \Rightarrow (G^n(x_1, ..., x_n) \Leftrightarrow G^n(y_1, ..., y_n))$
   where G^n is an n-ary relation symbol.

D2 Axioms for $+$:

1. $(x + 0) = x$
2. $(x + y) = (y + x)$
3. $(x + (y + z)) = ((x + y) + z)$
4. $((x + y) = (x + z)) \Rightarrow (y = z)$

D3 Axioms for $\geq$:

1. $0 \geq 0$
2. $((x \geq 0) \wedge (y \geq 0)) \Rightarrow ((x + y) \geq 0)$
3. $(x \geq y) \Leftrightarrow \exists z \geq 0.x = (y + z)$
4. $\neg(x \geq y) \Leftrightarrow (y > x)$
   where $(y > x) \mathrel{\widehat{=}} ((y \geq x) \wedge \neg(y = x))$

D4 Axiom for $\Leftrightarrow$:

$((x \Leftrightarrow y) = z) \Leftrightarrow (x = (y + z))$

The above axioms constitute a *minimal* first order theory that can guarantee the completeness of the calculus with respect to $\mathcal{A}$-validity. However, they are far from the 'best' set of axioms to characterize the real numbers. For example, a singleton of 0 will satisfy all the above axioms. One may like to introduce *multiplication* and *division*, or to have additional axioms and rules that capture more properties of the real numbers, such as *infinitude* and *density*:

D5 Axioms for infinitude:

$$\exists y.(y > x)$$

D6 Axioms for density:

$$(x > y) \Rightarrow \exists z.((x > z) \wedge (z > y)).$$

Given a first order theory $\mathcal{A}$ of the real numbers, a set $\mathbb{D}$ is called an $\mathcal{A}$-set, if the function symbols and the relation symbols are defined on $\mathbb{D}$ and satisfy $\mathcal{A}$. When an $\mathcal{A}$-set $\mathbb{D}$ is chosen as a time and value domain of NL, we denote the set of time intervals of $\mathbb{D}$ by $\mathbb{I}\text{ntv}_{\mathbb{D}}$, a value assignment from global variables to $\mathbb{D}$ by $\mathcal{V}_{\mathbb{D}}$, and an interpretation with respect to $\mathbb{D}$ by $\mathcal{J}_{\mathbb{D}}$, where

$$\mathbb{I}\text{ntv}_{\mathbb{D}} \mathrel{\widehat{=}} \{[b, e] | (b, e \in \mathbb{D}) \wedge (b \leq e)\},$$

$$\mathcal{V}_{\mathbb{D}} : GVar \rightarrow \mathbb{D},$$

and

$$\mathcal{J}_{\mathbb{D}}(v) : (\mathbb{I}\mathrm{ntv}_{\mathbb{D}} \to \mathbb{D}), \text{ for } v \in \mathit{TVar} \text{ and}$$
$$\mathcal{J}_{\mathbb{D}}(X) : (\mathbb{I}\mathrm{ntv}_{\mathbb{D}} \to \{\mathrm{tt},\mathrm{ff}\}), \text{ for } X \in \mathit{PLetter}$$

A *model* $\mathcal{M}_{\mathbb{D}}$ is a pair consisting of an $\mathcal{A}$-set $\mathbb{D}$ and an interpretation $\mathcal{J}_{\mathbb{D}}$.

The truth value of a formula ϕ given a model $\mathcal{M}_{\mathbb{D}}$, a value assignment $\mathcal{V}_{\mathbb{D}}$, and an interval $[b,e] \in \mathbb{I}\mathrm{ntv}_{\mathbb{D}}$ is similar to the semantic definitions on Page 6. We write $\mathcal{M}_{\mathbb{D}}, \mathcal{V}_{\mathbb{D}}, [b,e] \models_{\mathbb{D}} \phi$ to denote that ϕ is true for the given model, value assignment, and interval.

A formula ϕ is $\mathcal{A}$-*valid* (written $\models_{\mathcal{A}} \phi$) iff ϕ is true for every $\mathcal{A}$-model $\mathcal{M}_{\mathbb{D}}$, value assignment $\mathcal{V}_{\mathbb{D}}$, and interval $[b,e] \in \mathbb{I}\mathrm{ntv}_{\mathbb{D}}$. Furthermore, ϕ is $\mathcal{A}$-*satisfiable* iff ϕ is true for some $\mathcal{A}$-model $\mathcal{M}_{\mathbb{D}}$, value assignment $\mathcal{V}_{\mathbb{D}}$, and interval $[b,e] \in \mathbb{I}\mathrm{ntv}_{\mathbb{D}}$.

The proof system is sound and complete with respect to the class of $\mathcal{A}$-models:

Theorem 4. *(Soundness.) If* $\vdash \phi$ *then* $\models_{\mathcal{A}} \phi$

Theorem 5. *(Completeness.) If* $\models_{\mathcal{A}} \phi$ *then* $\vdash \phi$

A proof of the soundness theorem can be given by proving that each axiom is sound and that each inference rule preserves soundness in the sense that it gives a sound formula when applied to sound formulas. A proof of the completeness theorem can be developed along the line proposed in [4]. One can first prove a completeness of the calculus with respect to a kind of Kripke models, and then map the interval models to the Kripke models. The proof details are presented in [3].

5 Duration Calculus Based on NL

The NL based Duration Calculus (DC) can be established as an extension of NL in the same way as it was established as an extension of ITL [21, 8]. The induction rules of DC must, however, be weakened when DC is based on NL [15].

5.1 Syntax

The idea is to give temporal variables $v \in \mathit{TVar}$ a structure:

$$\int S$$

where S is called a *state expression* and is generated from a set SVar of *state variables* $P, Q, R, \ldots$, according to the following abstract syntax:

$$S ::= 0 \mid 1 \mid P \mid \neg S_1 \mid S_1 \vee S_2$$

We will use the same abbreviations for propositional connectives in state expressions as introduced for NL formulas.

Remark: The propositional connectives $\neg$ and $\vee$ occur both in state expressions and in formulas but, as we shall see below, with different semantics. This does not give problems as state expressions always occur in the context of $\int$.

5.2 Semantics

When we generate temporal variables from state variables, the semantics of temporal variables must be derived from the semantics of the state variables. To this end we introduce an *interpretation for state variables* (and propositional letters) as a function:

$$\mathcal{I} \in \begin{pmatrix} SVar \\ \cup \\ PLetters \end{pmatrix} \to \begin{pmatrix} \mathbb{T}\mathrm{ime} \to \{0,1\} \\ \cup \\ \mathbb{I}\mathrm{ntv} \to \{\mathrm{tt},\mathrm{ff}\} \end{pmatrix}$$

where $\mathcal{I}(P) \in \mathbb{T}\mathrm{ime} \to \{0,1\}$ and $\mathcal{I}(X) \in \mathbb{I}\mathrm{ntv} \to \{\mathrm{tt},\mathrm{ff}\}$, and each function $\mathcal{I}(P)$ has at most a finite number of discontinuity points in any interval. Hence, $\mathcal{I}(P)$ is integrable in any interval.

The semantics of a state expression S, given an interpretation $\mathcal{I}$, is a function:

$$\mathcal{I}[\![S]\!] \in \mathbb{T}\mathrm{ime} \to \{0,1\}$$

defined inductively on the structure of state expressions by:

$$\begin{array}{ll} \mathcal{I}[\![0]\!](t) & = 0 \\ \mathcal{I}[\![1]\!](t) & = 1 \\ \mathcal{I}[\![P]\!](t) & = \mathcal{I}(P)(t) \\ \mathcal{I}[\![(\neg S)]\!](t) & = 1 \Leftrightarrow \mathcal{I}[\![S]\!](t) \\ \mathcal{I}[\![(S_1 \vee S_2)]\!](t) & = \begin{cases} 0 \text{ if } \mathcal{I}[\![S_1]\!](t) = 0 \text{ and } \mathcal{I}[\![S_2]\!](t) = 0 \\ 1 \text{ otherwise} \end{cases} \end{array}$$

We shall use the abbreviation $S_{\mathcal{I}} \mathrel{\widehat{=}} \mathcal{I}[\![S]\!]$. We see by this semantics that each function $S_{\mathcal{I}}$ has at most a finite number of discontinuity points in any interval and is thus integrable in any interval.

The semantics of temporal variables, which now have the form $\int S$, is given by a function $\mathcal{I}[\![\int S]\!] \in \mathbb{I}\mathrm{ntv} \to \mathbb{R}$ defined by:

$$\mathcal{I}[\![\int S]\!][b,e] = \int_b^e S_{\mathcal{I}}(t)dt$$

This function can be used to induce an interpretation $\mathcal{J}_{\mathcal{I}}$ for temporal variables v of the form $\int S$ and temporal propositional letters from $\mathcal{I}$:

$$\begin{array}{ll} \mathcal{J}_{\mathcal{I}}(X) = \mathcal{I}(X) & \text{for any temporal propositional letter } X \\ \mathcal{J}_{\mathcal{I}}(v) & = \mathcal{I}[\![\int S]\!] \text{ when } v \text{ is } \int S \end{array}$$

The semantics of a duration calculus formula ϕ, given an interpretation $\mathcal{I}$ to state variables, is a function:

$$\mathcal{I}[\![\phi]\!] \in Val \times \mathbb{I}\mathrm{ntv} \to \{\mathrm{tt},\mathrm{ff}\}$$

for which we use the abbreviations:

$$\mathcal{I}, \mathcal{V}, [b,e] \models_{dc} \phi \mathrel{\widehat{=}} \mathcal{I}[\![\phi]\!]\ (\mathcal{V}, [b,e]) = \mathrm{tt}$$
$$\mathcal{I}, \mathcal{V}, [b,e] \not\models_{dc} \phi \mathrel{\widehat{=}} \mathcal{I}[\![\phi]\!]\ (\mathcal{V}, [b,e]) = \mathrm{ff}$$

The function can be defined as follows:

$$\mathcal{I}, \mathcal{V}, [b,e] \models_{dc} \phi \text{ iff } \mathcal{J}_{\mathcal{I}}, \mathcal{V}, [b,e] \models \phi$$

The notions of satisfiability and validity of DC formulas are defined as for NL formulas.

5.3 Proof system

All axioms and inference rules of NL are adopted as axioms and inference rules of DC. Furthermore, axioms and inference rules are added since temporal variables now have a structure. The original axioms for durations are still sound when DC is based on NL:

(DC-A1) $\int 0 = 0$

(DC-A2) $\int 1 = \ell$

(DC-A3) $\int S \geq 0$

(DC-A4) $\int S_1 + \int S_2 = \int(S_1 \vee S_2) + \int(S_1 \wedge S_2)$

(DC-A5) $((\int S = x)^\frown(\int S = y)) \Rightarrow (\int S = (x+y))$

(DC-A6) $\int S_1 = \int S_2$, provided $S_1 \Leftrightarrow S_2$ holds in propositional logic

We must add inference rules to formalize the finite variability of state expressions[3]. Let X be a temporal propositional letter and ϕ be a formula in which X does not occur. Let S be any state expression and let $H(X)$ denote the formula $\Box_a(X \Rightarrow \phi)$.

The two induction rules are:

$$\text{IR1}\quad \begin{array}{l}\text{If} \quad H(\lceil\!\rceil) \ \text{ and } \ H(X) \Rightarrow (H(X^\frown\lceil S\rceil) \wedge H(X^\frown\lceil\neg S\rceil)) \\ \text{then } \phi\end{array}$$

and

$$\text{IR2}\quad \begin{array}{l}\text{If} \quad H(\lceil\!\rceil) \ \text{ and } \ H(X) \Rightarrow (H(\lceil\neg S\rceil^\frown X) \wedge H(\lceil S\rceil^\frown X)) \\ \text{then } \phi\end{array}$$

where $H(\psi)$ denote the formula obtained from $H(X)$ by replacing X with ψ and

$$\lceil\!\rceil \ \widehat{=}\ \ell = 0$$
$$\lceil S\rceil \ \widehat{=}\ (\int S = \ell) \wedge (\ell > 0)$$

[3] It turns out that the original induction rules for DC are not sound when DC is based on NL. A counter-example is given in [15].

With these axioms and rules for Duration Calculus, a deduction theorem for DC and a relative completeness result for DC with respect to valid formulas of NL can be proved [15]. These results extend earlier results obtained for DC based on ITL [8].

5.4 Specification of limits, liveness, and fairness

Limit: The following formula is true when the limit of $\int S$ over an infinite interval does not exist:

$$\forall x.\Diamond_r(\int S > x)$$

and the following formula is true when the limit of $\int S$ over an infinite interval is v:

$$\forall \epsilon > 0.\exists T.\Box_r((\ell > T) \Rightarrow (|\int S \Leftrightarrow v| < \epsilon)) \ .$$

Fairness: Suppose two processes are competing for a resource and $S_i(t) = 1$ denotes that process $i, i = 1, 2$, has access to the resource at time t. Assume that S_1 and S_2 are mutually exclusive (i.e. $\neg(S_1 \wedge S_2)$).

The two processes should have the same access time for the resource in the limit:

$$\forall \epsilon > 0.\exists T.\Box_r((\ell > T) \Rightarrow (|\int S_1 \Leftrightarrow \int S_2| < \epsilon)) \ .$$

Liveness: The following formula specifies that the state S occurs infinitely often:

$$inf\,(S) \mathrel{\widehat{=}} \Box_r\Diamond_r\Diamond_r[\![S]\!] \ .$$

For example, an oscillator for S is specified by:

$$inf\,(S) \wedge inf\,(\neg S) \ .$$

Strong fairness: If S_1 denotes a request for a resource and S_2 denotes response from the resource, then strong fairness requires that if there are infinitely often requests then there must be responses infinitely often also. This is formalized by:

$$inf\,(S_1) \Rightarrow inf\,(S_2) \ .$$

Weak fairness: The following formula express that a state S stabilizes to $S = 1$ from some time on:

$$stabilize(S) \mathrel{\widehat{=}} \Diamond_r\Box_r([\![\,]\!] \vee [\![S]\!]) \ .$$

Weak fairness requires that if request for a resource stabilizes, then there will be response from the resource infinitely often:

$$stabilize(S_1) \ \Rightarrow \ inf\,(S_2) \ .$$

5.5 Example: Delay insensitive circuits

A delay insensitive circuit is a circuit consisting of components having an unknown delay. The delay may vary with time and place, for example because it is data or temperature dependent.

In [9] there is a DC specification of a delay insensitive circuit and a proof of its correctness. This specification contains a free (global) variable for each component denoting a positive delay. The presence of all these variables causes that the specification and also its correctness proof are rather clumsy. In this example we sketch how aspects of delay insensitive circuits can be succinctly modeled using NL without use of (global) variables.

We consider circuits constructed from *components* where a component C has a list of *input ports* $\bar{x} = x_1, \ldots, x_m$ and an *output port* y, and it performs a function $f : \{0,1\}^m \to \{0,1\}$. However, there is an unknown delay from the input ports to the output port.

The input and output ports are modeled by state variables:

$$x_i, y : \mathbb{T}\mathrm{ime} \to \{0,1\} \ .$$

If $\bar{z} = z_1, \ldots, z_l$ is a sequence of state expressions and $\bar{a} \in \{0,1\}^l$, then $\bar{z} = \bar{a}$ denotes the state expression: $z_1' \wedge \cdots \wedge z_l'$ where $z_i' = z_i$ if $a_i = 1$, and $z_i' = \neg z_i$ if $a_i = 0$.

There are two requirements for a component C:

1. If $\bar{x} = \bar{i}$ then either y becomes $f(\bar{i})$ or $\bar{x}$ changes. This requirement is expressed as:

$$F_1 \mathrel{\widehat{=}} \Box_a(\llbracket \bar{x} = \bar{i} \rrbracket \ \Rightarrow \ \Diamond_r \Diamond_r (\llbracket y = f(\bar{i}) \rrbracket \vee \neg \llbracket \bar{x} = \bar{i} \rrbracket))$$

2. If y becomes $f(\bar{i})$, then y persists unless $\bar{x}$ changes. There are two ways to interpret this.
   (a) y persists for some time, no matter whether $\bar{x}$ changes:

$$F_{2,\mathrm{a}} \mathrel{\widehat{=}} \Box_a(\llbracket \bar{x} = \bar{i} \wedge y = f(\bar{i}) \rrbracket \ \Rightarrow \ \Diamond_r \llbracket y = f(\bar{i}) \rrbracket)$$

   (b) y may change as soon as $\bar{x}$ changes:

$$F_{2,\mathrm{b}} \mathrel{\widehat{=}} \Box_a((\llbracket \bar{x} = \bar{i} \rrbracket \wedge \Diamond_r^c \llbracket y = f(\bar{i}) \rrbracket) \ \Rightarrow \ \llbracket y = f(\bar{i}) \rrbracket)$$

An important property of the component is that the output stabilizes if the input stabilizes, and it is possible to prove that:

$$(F_1 \wedge F_2) \ \Rightarrow \ (\mathit{stabilize}(\bar{x} = \bar{i}) \ \Rightarrow \ \mathit{stabilize}(y = f(\bar{i})))$$

where F_2 is either $F_{2,\mathrm{a}}$ or $F_{2,\mathrm{b}}$.

Suppose a circuit N of such components is constructed in a way where output ports can be connected with input ports, but output ports are not connected with each other. The circuit will have a sequence of input ports $\bar{X} = X_1, \ldots, X_k$, i.e. the input ports of the components which receive input from the environment.

Furthermore, there will be a sequence of output ports $\bar{Y} = Y_1, \ldots, Y_l$, which send output to the environment.

Suppose the circuit should perform a function $f : \{0,1\}^k \to \{0,1\}^l$ in the sense that if the input stabilizes to $\bar{in} \in \{0,1\}^k$ then the output must eventually stabilize to $f(\bar{in})$. This can be specified as:

$$stabilize(\bar{X} = \bar{in}) \;\Rightarrow\; stabilize(\bar{Y} = f(\bar{in}))$$

and it is a proof obligation for the correctness of the circuit that this formula is implied by the conjunction of the formulas for the components.

6 Real Analysis

In this section, we express in NL notions of limits, continuity, differentiation etc. in a way very close to conventional mathematics of real analysis.

For every state S, we introduce a temporal propositional letter denoted: $\mathrm{at}(S)$ with the meaning that S holds (i.e. $S = 1$) at a time i.e.:

$$\mathcal{I}, \mathcal{V}, [b, e] \;\models\; \mathrm{at}(S) \text{ iff } b = e \text{ and } S_{\mathcal{I}}(b) = 1$$

The additional axioms and rule are:

- Either S or $\neg S$ holds at a time:

$$\mathrm{at}(\neg S) \Leftrightarrow\; ((\ell = 0) \wedge \neg \mathrm{at}(S))$$

- $(S_1 \vee S_2)$ holds at a time, iff either S_1 or S_2 holds:

$$\mathrm{at}(S_1 \vee S_2) \;\Leftrightarrow\; (\mathrm{at}(S_1) \vee \mathrm{at}(S_2))$$

- Monotonicity:

$$\text{If } S_1 \Rightarrow S_2, \text{ then } \mathrm{at}(S_1) \Rightarrow\; \mathrm{at}(S_2)$$

For expressing real analysis, we introduce an abbreviation, $\mathrm{in}(\phi)$, which specifies that ϕ holds at every point (interval) *inside* a non-point interval.

$$\mathrm{in}(\phi) \;\widehat{=}\; (\ell > 0) \wedge \neg((\ell > 0)^\frown(\neg\phi \wedge (\ell = 0))^\frown(\ell > 0))$$

We will abbreviate $\mathrm{in}(\mathrm{at}(S))$ to $\mathrm{in}(S)$, which means that $S = 1$ everywhere inside a non-point interval.

Limits: Let $\alpha : \mathbb{R} \to \mathbb{R}$.

The *right limit* of α at a point is v if for any $\epsilon > 0$ there exists a right neighbourhood of the point such that for any t in the neighbourhood: $|\alpha(t) \Leftrightarrow v| < \epsilon$. $|\alpha \Leftrightarrow v| < \epsilon$ is either true or false for $t \in \mathbb{R}$, and is therefore regarded as a state variable[4].

[4] Formally speaking, α belongs to another additional function symbol set of NL. $-$ and $<$ are operators over the additional symbol set, and are defined as *point-wise* generalization of the standard ones.

This definition from real analysis can be expressed directly in NL as:

$$Lmt^{+}(\alpha, v) \mathrel{\widehat{=}} (\ell = 0) \ \wedge \ \forall \epsilon > 0.\Diamond_r \mathrm{in}(|\alpha \Leftrightarrow v| < \epsilon)$$

The value of the right limit of α at a point, denoted α^{+}, can be defined by the *description* operator, ι, of first order predicate logic:

$$\alpha^{+} \mathrel{\widehat{=}} \iota y.Lmt^{+}(\alpha, y)$$

Similarly, the *left limit* of α at a point is v and the value of the left limit at a point can be expressed as:

$$\begin{array}{ll} Lmt^{-}(\alpha, v) & \mathrel{\widehat{=}} (\ell = 0) \wedge \forall \epsilon > 0.\Diamond_l \mathrm{in}(|\alpha \Leftrightarrow v| < \epsilon) \\ \alpha^{-} & \mathrel{\widehat{=}} \iota y.Lmt^{-}(\alpha, y) \end{array}$$

The following formula expresses that the limit of α is v, when t approaches ∞:

$$\forall \epsilon > 0.\exists T > 0.\Box_r(\ell > T \ \Rightarrow \ ((\ell = T)^\frown \mathrm{in}(|\alpha \Leftrightarrow v| < \epsilon)))$$

Continuity: A function α is *continuous* at a point, if its left limit is equal to its right limit, and equals its value at the point. This is expressed in NL as:

$$Cnt(\alpha) \mathrel{\widehat{=}} (\alpha^{+} = \alpha^{-} = \iota y.\mathrm{at}(\alpha = y))$$

Thus, the continuity of α inside an interval is expressed in NL as: $\mathrm{in}(Cnt(\alpha))$.

Derivatives: The following formulas express that the left (right) *derivative* of α at a point is v:

$$\begin{array}{l} Dft^{-}(\alpha, v) \mathrel{\widehat{=}} \exists y.(\mathrm{at}(\alpha = y) \wedge \forall \epsilon > 0.\exists \delta > 0.\Box_l((\delta > \ell > 0) \Rightarrow |\frac{\alpha_b - y}{\ell} \Leftrightarrow v| < \epsilon)) \\ Dft^{+}(\alpha, v) \mathrel{\widehat{=}} \exists y.(\mathrm{at}(\alpha = y) \wedge \forall \epsilon > 0.\exists \delta > 0.\Box_r((\delta > \ell > 0) \Rightarrow |\frac{\alpha_e - y}{\ell} \Leftrightarrow v| < \epsilon)) \end{array}$$

where α_b (α_e) stands for $\iota y.\Diamond_l \mathrm{at}(\alpha = y)$ ($\iota y.\Diamond_r \mathrm{at}(\alpha = y)$), which defines the value of α at the beginning (ending) point of an arbitrary given interval.

We can express that v is the derivative of α at a point and the derivative of α as:

$$\begin{array}{ll} Dft(\alpha, v) & \mathrel{\widehat{=}} Dft^{-}(\alpha, v) \wedge Dft^{+}(\alpha, v) \\ \dot{\alpha} & \mathrel{\widehat{=}} \iota y.Dft(\alpha, y) \end{array}$$

Thus, $\mathrm{in}(\dot{\alpha} = \beta)$ expresses that the differential equation $\dot{\alpha} = \beta$ holds within an interval, where $\beta : \mathbb{R} \to \mathbb{R}$.

7 Discussions

Some research of DC ([24, 10, 20]) has already applied the notion of left and right neighbourhoods for describing instant actions in ITL. An instant state transition from S_1 to S_2 can be modeled as a neighbourhood property of the transition, such that S_1 holds in a left neighbourhood of the transition and S_2 holds in a right neighbourhood of it. They use the notation $\nwarrow S_1$ and $\nearrow S_2$ to express those neighbourhood properties, and the conjunction $(\nwarrow S_1 \wedge \nearrow S_2)$ to specify the transition. $\diamond_l$ and $\diamond_r$ can be regarded as a generalization of $\nwarrow$ and $\nearrow$, since $\nwarrow S_1$ can be defined as $\diamond_l [\![S_1]\!]$, and $\nearrow S_2$ as $\diamond_r [\![S_2]\!]$, where S_1 and S_2 are considered state variables.

In order to develop formal technique for designing hybrid systems, this paper tries to axiomatize a mathematical theory of real analysis in the framework of interval temporal logic. Such an axiomatization requests a long term effort, and this paper presents a very tentative attempt in this direction.

References

1. J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.
2. R. Alur, C. Courcoubetis, T. Henzinger, and P-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems, R.L. Grossman, A. Nerode, A.P. Ravn and H. Rischel (Eds)*, pages 209–229. LNCS 736, Springer-Verlag, 1993.
3. R. Barua and Zhou Chaochen. Neighbourhood logics: NL and NL^2. Technical report, UNU/IIST Report No. 120, UNU/IIST, International Institute for Software Technology, P.O. Box 3058, Macau, 1997.
4. B. Dutertre. Complete proof systems for first order interval temporal logic. In *Tenth Annual IEEE Symp. on Logic in Computer Science*, pages 36–43. IEEE Press, 1995.
5. Marcin Engel and Hans Rischel. Dagstuhl-seminar specification problem - a duration calculus solution. Technical report, Department of Computer Science, Technical University of Denmark – Private Communication, 1994.
6. J. Halpern, B. Moskowski, and Z. Manna. A hardware semantics based on temporal intervals. In *ICALP'83*, volume 154 of *LNCS*, pages 278–291. Springer-Verlag, 1983.
7. J.Y. Halpern and Y. Shoham. A propositional modal logic of time intervals. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 279–292. IEEE Computer Society Press, 1986.
8. M.R. Hansen and Zhou Chaochen. Duration calculus: Logical foundations. *Formal Aspects of Computing* 9: 283-330, 1997.
9. M.R. Hansen, Zhou Chaochen, and Jørgen Staunstrup. A real-time duration semantics for circuits. In *TAU'92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Princeton Univ., NJ.* ACM/SIGDA, 1992.
10. M.R. Hansen, P.K. Pandya, and Zhou Chaochen. Finite divergence. *Theoretical Computer Science*, 138:113–139, 1995.

11. L. Lamport. Hybrid systems in tla$^+$. In *Hybrid Systems, R.L. Grossman, A. Nerode, A.P. Ravn and H. Rischel (Eds)*, pages 77–102. LNCS 736, Springer-Verlag, 1993.
12. Z. Manna and A. Pnueli. Verifying hybrid systems. In *Hybrid Systems, R.L. Grossman, A. Nerode, A.P. Ravn and H. Rischel (Eds)*, pages 4–35. LNCS 736, Springer-Verlag, 1993.
13. B. Moszkowski. Compositional reasoning about projected and infinite time. In *First IEEE Intl. Conf. on Engineering of Complex Computer Systems*. IEEE press, 1995.
14. P.K. Pandya. Weak chop inverses and liveness in duration calculus. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 148–167. Springer-Verlag, 1996.
15. Suman Roy and Zhou Chaochen. Notes on neighbourhood logic. Technical report, UNU/IIST Report No. 97, UNU/IIST, International Institute for Software Technology, P.O. Box 3058, Macau, 1997.
16. J.U. Skakkebæk. Liveness and fairness in duration calculus. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *LNCS*, pages 283–298. Springer Verlag, 1994.
17. Y. Venema. Expressiveness and completeness of an interval tense logic. *Notre Dame Journal of Formal Logic*, 31(4):529–547, 1990.
18. Y. Venema. A modal logic for chopping intervals. *J. Logic Computat.*, 1(4):453–476, 1991.
19. Mao Xiaoguang, Xu Qiwen, Dang Van Hung, and Wang Ji. Towards a proof assistant for interval logics. Technical report, UNU/IIST Report No. 77, UNU/IIST, International Institute for Software Technology, P.O. Box 3058, Macau, 1996.
20. Zhou Chaochen and Michael R. Hansen. Chopping a point. In *BCS-FACS 7th Refinement Workshop*. Electronic Workshops in Computing, Springer-Verlag, 1996.
21. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
22. Zhou Chaochen, Dang Van Hung, and Li Xiaoshan. A duration calculus with infinite intervals. In Horst Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *LNCS*, pages 16–41. Springer-Verlag, 1995.
23. Zhou Chaochen, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems, R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (Eds.)*, volume 736 of *LNCS*, pages 36–59. Springer-Verlag, 1993.
24. Zhou Chaochen and Li Xiaoshan. A mean value calculus of durations. In Prentice Hall International, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 431–451. Prentice Hall International, 1994.

Compositional Transformational Design for Concurrent Programs

Job Zwiers *

University of Twente, Department of Computer Science,
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. Partial order based techniques are incorporated in a interleaving semantics, based on coding partial order information in linear time temporal logic. An example of development is given.

1 Introduction and Overview

Compositional program development means that the syntactic structure of a program guides the development. Such guidance need not be limited to classical top-down or bottom-up development, but applies also to *transformational* development. Compositionality here requires that a program transformation on P is carried out by identifying some syntactic subprogram S_1 of P and replacing it by some other subprogram S_2. In this paper we combine transformation of concurrent programs with the classical Owicki-Gries style techniques for proving the correctness of concurrent programs. This transformational method is particularly attractive for the design of concurrent and distributed programs. Unlike the sequential case, algorithms for concurrent and distributed programs usually take the system architecture into account, and are therefore dependent on aspects as the number of parallel processing units available, the structure and performance of the network connecting those processing units. This leads to a wide variety of algorithms, each one tailored for some particular architecture, for solving one and the same task. The transformational approach often allows one to start off with a *single* common algorithm, and then to transform it in various ways, so as to conform to the requirements of particular architectures. Informally we often make a distinction between the "physical structure" and the "logical structure" of programs. The physical structure refers to a program S_D that has a syntactic structure that is reasonably close to the actual physical architecture that will be used to execute the program. For instance, one expects a simple and straightforward mapping from the parallel processes of S_D to the actual processing units available in a given system architecture. We assume here that the physical structure can be described as a collection of distributed and communicating network nodes P_i, each of which executes a number of sequential phases $P_{i,1} \cdots P_{i,m}$. Schematically, this can be described as a program of the following form:

$$S_D \stackrel{\text{def}}{=} [(P_{1,1}\ ;\ P_{1,2}\ ;\ \cdots\ ;\ P_{1,m})\ \|\ \cdots\ \|\ (P_{n,1}\ ;\ P_{1,2}\ ;\ \cdots\ ;\ P_{n,m}).]$$

* The author thanks Mannes Poel for collaboration and fruitful discussions

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 609-631, 1998.
Springer-Verlag Berlin Heidelberg 1998

The logical structure on the other hand is a program S_L that has a syntactic structure that allows for simple verification. As observed for instance in [EF82, SdR87, JZ92b, JZ92a, ZJ94, JPXZ94, FPZ93] the "logical" structure, i.e. the structure that one would use to explain the protocol, often appears to be *sequentially phased*, that is, as a program of the form:

$$S_L \stackrel{\text{def}}{=} [P_{1,1} \parallel \cdots \parallel P_{n,1}] ; \cdots ; [P_{1,m} \parallel \cdots \parallel P_{n,m}.]$$

One of the aims of this paper is to explain under which circumstances such sequentially phased programs S_L are actually *equivalent* to their corresponding distributed version S_D, where "equivalence" is to be understood here as: "computing the same input-output relation". In this paper we consider both shared variable programs and programs with explicit communication commands. The latter class is regarded here as a subclass of all shared variable programs, where shared variables are used only in the form of communication buffers and synchronization flags, so to say. The techniques for shared variables remain valid for communication based programs, but, due to the restrictions, a more attractive formulation is possible than the transformation laws for *general* shared variable programs. This is the principle of *communication closed layers*, originally proposed in [EF82]. Informally for a program S_L as above, a sequential phase of the form $L_i \stackrel{\text{def}}{=} [P_{1,i} \parallel \cdots \parallel P_{n,i}]$, for some i such that $1 \leq i \leq m$, is called communication closed if for every communication channel c the number of **send** commands executed inside L_i equals the number of **receive** commands for c inside L_i. The communication closed layers law claims that when all sequential phases L_i, for $1 \leq i \leq m$, are communication closed, then S_L is equivalent to the distributed version S_D. Technically speaking this law can be justified from the laws for commuting actions. In practice however, "counting" **send** and **receive** actions is often simpler than reasoning directly in terms of commuting actions. Moreover, communication closedness is a property that can be checked for each sequential phase *in isolation*, that is, without consideration of other phases. For (general) shared variable programs on the other hand, one has to reason about ordering of non-commuting actions *across sequential phases*.

The techniques used in this paper are based on *precedence relations* among occurrences of program actions. Basically, such properties have the form "a_1 precedes a_2", with meaning that in every possible computation of the program the points where a_1 is executed all precede the points where a_2 is executed. The question is how to deal with this class of properties within proofs in the style of the Owicki-Gries method. In principle, the Owicki-Gries method uses assertions on *states*, whereas precedence properties are assertions that are interpreted over *computations*. For instance, an assertion that asserts that "all occurrences of action a_1 precede all occurrences of action a_2" cannot be interpreted as being "*true*" or "*false*" in an arbitrary program state σ. A possible solution is to introduce a boolean typed auxiliary variable for action a_2, that we will call "$a_2.occurred$", and which is set to "*true*" as soon as action a_2 is executed. In this way, the above precedence relation could be verified by means of an Owicki-Gries style proof outline where a_1 has a precondition that implies "$\neg a_2.occurred$". We

prefer to give a more systematic approach, based on so called *temporal logic* [MP92, Lam83b]. This is a logic with formulae ϕ interpreted over computations, rather than states, and which is perfect for formulating precedence properties. We use only a small fragment of linear time temporal logic, with operators that refer to the past history of a given state. Other models, based on partial orders, like in [Pra86, Maz89, Zwi91, JPZ91, JZ92a, FPZ93, ZJ94, JPXZ94] are often used, and are slightly more natural for dealing with precedence relations among actions. (In essence, a computation in the partial order model of for instance [JPZ91] can be seen as a collection of actions together with a set of precedence relations on these actions.) Closely related are logics like Interleaving Set Temporal Logic (ISTL) [KP87, KP92].

1.1 Related work on Communication closed Layers

Stomp and de Roever [SdR87, SdR94] introduce what they call a *principle for sequentially phased reasoning* which allows for *semantically defined* layers that should correspond to the intuitive ideas of the designers. In [Sto89, Sto90] this principle is applied to the derivation of a broadcast protocol. Chou and Gafni [CG88] group classes of actions and define a sequential structure on such classes (so-called *stratification*). Both approaches are closely related to the idea of *communication closed layers* for communication based programs introduced by Elrad and Francez [EF82] and studied in [GS86]. In [JZ92a] an explanation of [GHS83] is given, inspired by ideas from [SdR87, SdR89], based on *layered composition,* sometimes called "conflict (based) composition" or "weak sequential composition" [JPZ91, FPZ93, RW94]. Layer composition of the form $S_1 \bullet S_2$ is a program composition operator like parallel or sequential composition, that is useful for composing "sequential phases" or "communication closed layers". The idea is that when a program of the form $S_1 \bullet S_2 \bullet \cdots \bullet S_n$ runs as a "closed system", that is, without interference by other parallel programs, then the input-output behavior is the same as that of the sequential program $S_1 \,;\, S_2 \,;\, \cdots \,;\, S_n$. Therefore, it can be analyzed and shown correct by means of classical Hoare's logic for sequential programs, at least at this level of detail. On a *more* detailed level of design the components $S_1, S_2, \ldots, S_n$ themselves can be *parallel programs*. The layer composition operator then turns out to be different from ordinary sequential composition: actions a_1 and a_2 from different layers are sequentially ordered only if a_1 and a_2 are non-commuting. Using the layer composition operator one can describe the distributed version S_D and the sequentially phased or layered version S_L that we discussed (using sequential rather than layer composition) above, as follows:

$$S_D \stackrel{\text{def}}{=} [(P_{1,1} \bullet P_{1,2} \bullet \cdots \bullet P_{1,m}) \parallel \cdots \parallel (P_{n,1} \bullet P_{1,2} \bullet \cdots \bullet P_{n,m}).]$$

$$S_L \stackrel{\text{def}}{=} [P_{1,1} \parallel \cdots \parallel P_{n,1}] \bullet \cdots \bullet [P_{1,m} \parallel \cdots \parallel P_{n,m}.]$$

A generalized version of the communication closed layers (CCL) law was proposed in [JPZ91], for shared variables rather than communication however, and

based on layer composition rather than sequential composition. The side conditions (in [JPZ91]) for the CCL law are that actions from $P_{i,j}$ must commute with actions from $P_{k,l}$ unless $i = k$ or $j = l$. Under these conditions the programs S_L and S_D are semantically identical, that is, they have identical sets of computations. This certainly *implies* that S_L and S_D have the same input-output relation, but it is a stronger property than just that: it also implies that S_L can replace S_D, or vice versa, *in arbitrary contexts,* including contexts where processes run concurrently with S_L or S_D. That is, the equivalence between S_L and S_D is actually a *congruence*. On the other hand, when one replaces layer composition in S_L and S_D by sequential composition, then, under the same side conditions, input-output equivalence is still guaranteed, but this equivalence is not preserved when processes run concurrently. That is, S_L can be replaced by S_D within sequential contexts only. One does not expect the correctness of such algorithms to be preserved when run in an arbitrary concurrent context that could read and modify the shared variables used by the algorithm, so this seems to be not a very severe restriction. When an algorithm like "set partitioning" would be incorporated into a larger concurrent program, then one would take care that no interference would be possible between the algorithm and the rest of the system, for instance by declaring the shared variables of the algorithm to be "private" variables [MP92] of the algorithm. For programs like the two-phase commit protocol the situation is more complex. A protocol like this one is usually only a small part of more elaborate protocols, for instance a protocol dealing with reliable atomic transactions in distributed databases [BHG87]. In such cases the final (distributed) algorithm has to be combined with other protocols that interact in a non-trivial way. In such situations one needs either a congruence between layered and distributed version of the protocol, or else one must be careful when combining protocols. Mechanisms for combining protocols that do not depend on compositional program operators, and therefore do not depend on congruence properties, are discussed in [BS92, CM88, Kat93]

Overview In section 2 the syntax is introduced and the semantics is given in section 3. The small fragment of temporal logic which is needed for formulating the communication closed principle is discussed in section 4. In sections 5 and 6 several versions of the CCL laws are formulated. A generalization of the CCL laws for loops is given in section 6.1. An examples of program transformation is given in section 7, where a well known set partitioning algorithm is discussed.

2 Syntax and informal meaning of programs

We introduce a small programming language which is is in essence a variation of the well known guarded command languages for shared variable concurrency. A novel point here is that guarded assignment actions are named by means of a (unique) label in front of them. We use variables x, expressions e, boolean expressions b, channel names c, and for action names we use a. The syntax is given in table 1 below.

Table 1. Syntax of the Programming Language

Actions	$act ::= \langle b \to \bar{x} := \bar{e} \rangle \mid \mathbf{send}(c, e) \mid \mathbf{receive}(c, x)$
Programs	$S ::= a : act \mid S_1 ; S_2 \mid \mathbf{if}\ \ b_1 \to S_1 \ [] \ b_n \to S_n\ \ \mathbf{fi} \mid$ $\mathbf{do}\ b_1 \to S_1 \ [] \ b_n \to S_n\ \mathbf{od} \mid [S_1 \parallel S_2]$
Closed programs	$Sys ::= \langle S \rangle$

We use assignments boolean guards "b" and assignments without a guard of the form "$\bar{x} := \bar{e}$" as straightforward abbreviations of guarded assignments. We require that all action names a within a program S are *unique*. In practice we suppress most of these labels and we put only labels in front of those actions that are referred to in formulae that are used to specify and verify the program. Formally speaking we assume in such cases that, implicitly, some labeling scheme is used that assigns unique names to all unlabeled actions. We refer to [Lam83a] for an example of such a labeling scheme.
We assume that communication channels are unidirectional and point-to-point channels. The **send** and **receive** commands model *asynchronous send* and *receive* commands for a channel with a one-place buffer. Informally, $\mathbf{send}(c, e)$ evaluates expression e and sends the result along channel c, where it is temporarily stored in a one-place buffer. In case the buffer is filled up already, the send command waits until it is emptied. The receive action $\mathbf{receive}(c, x)$ waits until a message is put in the buffer associated with the channel, retrieves it, and stores the message in a variable x. The **send** and **receive** commands are therefore abbreviations:

$$\mathbf{send}(c, e) \stackrel{\text{def}}{=} \langle \neg c.full \to c.full, c.buf := true, e \rangle$$

$$\mathbf{receive}(c, x) \stackrel{\text{def}}{=} \langle c.full \to c.full, x := false, c.buf \rangle.$$

For closed programs we assume that initially all "semaphores" of the form $c.full$ associated with communication channels c are set to "$false$", denoting that channels are empty initially. Formally speaking, we take care of this by assuming that the preconditions of Hoare formulae for closed programs implicitly a conjunct of the form $\neg c.full$ for all relevant channels c.

3 The semantics of programs

A reactive sequence models a computation of a system which takes into account possible interactions by parallel components. This is modeled using "gaps" whenever two subsequent computation steps have non-identical final and initial states. So we use reactive sequences θ of the form:

$$\langle \sigma_0 \stackrel{a_0}{\to} \sigma_0' \rangle \langle \sigma_1 \stackrel{a_1}{\to} \sigma_1' \rangle \ldots \langle \sigma_{i-1} \stackrel{a_{i-1}}{\to} \sigma_{i-1}' \rangle \langle \sigma_i \stackrel{a_i}{\to} \sigma_i' \rangle \langle \sigma_{i+1} \stackrel{a_{i+1}}{\to} \sigma_{i+1}' \rangle \ldots$$

Here, σ_i and σ_i' are the initial and final state of the $i-th$ step, whereas a_i is the name (i.e. the label) of the action being executed for the $i-th$ step

We provide here a semantics $\mathcal{RA}[\![S]\!]$ which defines the reactive sequences for terminating computations of S. It is possible to define deadlock behavior and divergent computations in a similar style, but since we are aiming here at partial correctness only, we won't need those here.

Definition 1 (Reactive event sequence semantics).

- $\mathcal{RA}[\![a : \langle b \to \bar{x} := \bar{e}\rangle]\!] \stackrel{\text{def}}{=} \{\langle\sigma \stackrel{a}{\to} \sigma'\rangle \mid \sigma \models b \ \wedge \ \sigma' = f(\sigma)\}$, where the state transformation f is the meaning of the assignment $\bar{x} := \bar{e}$.
- $\mathcal{RA}[\![S_1\,;\,S_2]\!] \stackrel{\text{def}}{=} \mathcal{RA}[\![S_1]\!] \frown \mathcal{RA}[\![S_2]\!]$, where $\frown$ is the operation of concatenation of sequences, pointwise extended to sets of reactive sequences.
- $\mathcal{RA}[\![\,\mathbf{if}\quad b_1 \to S_1 \,[]\, b_n \to S_n\ \mathbf{fi}\ ,]\!] \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \mathcal{RA}[\![b_i]\!] \frown \mathcal{RA}[\![S_i]\!]$.
- Let $R^{(0)} = \mathcal{RA}[\![\neg b_1 \wedge \cdots \wedge \neg b_n]\!]$, and
  let $R^{(i+1)} = \mathcal{RA}[\![\,\mathbf{if}\quad b_1 \to S_1 \,[]\, b_n \to S_n\ \mathbf{fi}\ ]\!] \frown R^{(i)}$, for $i \geq 0$.
  Then $\mathcal{RA}[\![\,\mathbf{do}\ b_1 \to S_1 \,[]\, b_n \to S_n \mathbf{od}\,]\!] \stackrel{\text{def}}{=} \bigcup\{R^{(i)} \mid i \geq 0\}$
- $\mathcal{RA}[\![S_1 \parallel S_2]\!] \stackrel{\text{def}}{=} \mathcal{RA}[\![S_1]\!] \,\tilde{\|}\, \mathcal{RA}[\![S_2]\!]$, where $\tilde{\|}$ denotes the operation of interleaving of reactive sequences.

Finally, we define the semantics of *closed* systems. We say that a reactive sequence θ of the form

$$\langle\sigma_0 \stackrel{a_0}{\to} \sigma_0'\rangle\langle\sigma_1 \stackrel{a_1}{\to} \sigma_1'\rangle \ldots \langle\sigma_i \stackrel{a_i}{\to} \sigma_i'\rangle\langle\sigma_{i+1} \stackrel{a_{i+1}}{\to} \sigma_{i+1}'\rangle \ldots$$

is *connected* iff $\sigma_i' = \sigma_{i+1}$ for all indices i such that both σ_i' and σ_{i+1} belong to θ. For a closed system $\langle S\rangle$ no interaction with parallel components from outside of S is assumed, hence we require that for such closed systems the reactive sequences don't contain "gaps", i.e. are connected.

- The reactive sequence semantics $\mathcal{RA}[\![\langle S\rangle]\!]$ of a closed system $\langle S\rangle$ is defined as $\mathcal{RA}[\![\langle S\rangle]\!] \stackrel{\text{def}}{=} \{\theta \in \mathcal{RA}[\![S]\!] \mid \theta \text{ is connected}\}$.

Finally, we define the input-output semantics $\mathcal{O}[\![S]\!]$, based on the reactive sequences semantics. First we define an auxiliary function $\mathcal{IO}(\eta)$ for reactive sequences η, that constructs the initial-final state pair from a (finite) reactive sequence:

$$\mathcal{IO}(\langle\sigma_0 \stackrel{a_0}{\to} \sigma_0'\rangle \ldots \langle\sigma_n \stackrel{a_i}{\to} \sigma_n'\rangle) = (\sigma_0, \sigma_n')$$

The $\mathcal{O}[\![S]\!]$ semantics of systems is then easily determined from the reactive sequences for the corresponding *closed* system $\langle S\rangle$:

$$\mathcal{O}[\![S]\!] = \{\mathcal{IO}(\eta) \mid \eta \in \mathcal{RA}[\![\langle S\rangle]\!]\}$$

Table 2. Syntax of the Temporal Logic fragment TL^-

TL formulae $\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \neg\phi \mid \Box\phi \mid \Diamond\phi \mid \blacksquare\phi \mid \blacklozenge\phi$

4 Partial orders and temporal logic

Programs are specified here within a subset of temporal logic. [Pnu77, Lam83b, MP84, MP92, MP95]. In this paper we use a variant of so called past time temporal logic, to specify partial order constraints among actions. The syntax of the fragment of temporal logic that we use is given in table 2. In this table, a denotes an action name. We use standard abbreviations, such as $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$. Moreover, we use $\phi_1 \Rightarrow \phi_2$ for the temporal formula $\Box(\phi_1 \rightarrow \phi_2)$. We define a type of correctness formulae of the form S **psat** ϕ, where S denotes a program. The formula S **psat** ϕ denotes that all reactive sequences of S satisfy temporal logic formula ϕ. Similarly, $\langle S \rangle$ **psat** ϕ denotes that all reactive sequences of $\langle S \rangle$, that is, all *connected* reactive sequences of S, satisfy the temporal formula ϕ. We remark that in the literature on temporal logic of reactive systems one uses a different type of correctness formulae of the form "S **sat** ϕ". (Often one simply specifies the temporal formula ϕ, and leaves the system S implicit). Such formulae require a temporal logic formula ϕ to hold for all finite and infinite computations of system S. Our " **psat** " relation requires ϕ to hold for *terminating* computations only, and therefore is suited only for partial correctness properties, i.e. properties that should hold *provided that a computation terminates.*
Let $\theta = \langle \sigma_0 \stackrel{a_0}{\rightarrow} \sigma_0' \rangle \ldots \langle \sigma_n \stackrel{a_n}{\rightarrow} \sigma_n' \rangle \in \mathcal{RA}[\![S]\!]$ be a reactive sequence of a program S. We define the relation $(\theta, j) \models \phi$ (" ϕ holds at position j in sequence θ") for all $j, 0 \leq j \leq n$, as follows:

- For action names a, we define:

$$(\theta, j) \models a \text{ iff } a_j = a.$$

- For boolean connectives we define:

$$(\theta, j) \models \phi_1 \wedge \phi_2 \text{ iff } (\theta, j) \models \phi_1 \text{ and } (\theta, j) \models \phi_2,$$

$$(\theta, j) \models \phi_1 \rightarrow \phi_2 \text{ iff } (\theta, j) \models \phi_1 \text{ implies } (\theta, j) \models \phi_2,$$

$$(\theta, j) \models \neg\phi \text{ iff not } (\theta, j) \models \phi,$$

- For temporal operators we define:

$$(\theta, j) \models \Box\phi \text{ iff } (\theta, k) \models \phi \text{ for all } k,\ j \leq k \leq n$$

$$(\theta, j) \models \Diamond\phi \text{ iff } (\theta, k) \models \phi \text{ for some } k,\ j \leq k \leq n$$

$$(\theta, j) \models \blacksquare\phi \text{ iff } (\theta, k) \models \phi \text{ for all } k,\ 0 \leq k \leq j.$$

$$(\theta, j) \models \blacklozenge\phi \text{ iff } (\theta, k) \models \phi \text{ for some } k,\ 0 \leq k \leq j.$$

Definition 2.
We define the meaning of a correctness formula as follows:

$$S \;\; \textbf{psat} \;\; \phi \text{ iff for all } \theta \in \mathcal{RA}[\![S]\!], (\theta, 0) \models \phi.$$
$$\langle S \rangle \;\; \textbf{psat} \;\; \phi \text{ iff for all } \eta \in \mathcal{RA}[\![\langle S \rangle]\!], (\eta, 0) \models \phi.$$

□

According to the semantics, every reactive sequence of $\langle S \rangle$ is also a possible reactive sequences of S. This fact, combined with the definition of the **psat** relation, then implies the following result:

Theorem 3 (From open to closed programs).
If S **psat** ϕ *is valid, then* $\langle S \rangle$ **psat** ϕ *is valid.*

We introduce an *ordering relation* on actions, that play a major role in the program transformations we discuss later in this paper:

Definition 4 (Action ordering).
We define the *weak precedence relation* $a_1 \twoheadrightarrow a_2$:

$$a_1 \twoheadrightarrow a_2 \stackrel{\text{def}}{=} a_1 \Rightarrow \neg\blacklozenge a_2.$$

□

(Note that according to our conventions, $a_1 \twoheadrightarrow a_2$ is the formula $\Box(a_1 \rightarrow \neg\blacklozenge a_2)$.) Informally, $a_1 \twoheadrightarrow a_2$ requires that when both a_1 and a_2 occur in one execution sequence, then all a_1 occurrences precede all a_2 occurrences within the execution sequence. Note that the relation does not require that a_1 or a_2 to be executed at all; for instance, an execution sequence with some a_2 occurrences but no occurrences of a_1 does satisfy "$a_1 \twoheadrightarrow a_2$". (This explains why the ordering is called "weak".) Weak precedence is expressed by a temporal logic formula. The question is: how does one *verify* such formulae , and in particular, how does one verify $a_1 \Rightarrow \neg\blacklozenge a_2$ within the Owicki-Gries style proof method? Let us assume that we have a program S with actions a_1 and a_2, and we would like to verify the property $a_1 \twoheadrightarrow a_2$. Intuitively, one would like to construct a proof outline for S where the precondition for action a_1, say $pre(a_1)$, would imply the formula $\neg\blacklozenge a_2$. Formally speaking this wouldn't be possible since $pre(a_1)$ is a state formula, whereas a temporal formula like $\neg\blacklozenge a_2$ cannot be interpreted in a single state. Fortunately, the formula that we are interested in is expressed by means of past time operators only, and therefore we can transform it into a state formula by introducing auxiliary variables. In this case we introduce a boolean auxiliary variable "$a_2.occ$" that records whether a_2 did occur at least once thus far, or not. Let action a_2 be of the form $\langle b_2 \rightarrow \bar{x_2} := e_2 \rangle$. Then we modify the program S and construct a proof outline for S, as follows:

- The action a_2 in S is replaced by $\langle b_2 \rightarrow \bar{x_2}, a_2.occ := e_2, true \rangle$. No other assignments to $a_2.occ$ are added within S.
- Construct a proof outline $\{p\}A(S)\{q\}$, of S such that $p \rightarrow \neg a_2.occ$ and $pre(a_1) \rightarrow \neg a_2.occ$.

Then S satisfies the condition $a_1 \twoheadrightarrow a_2$.

Theorem 5 (Verifying action ordering).
Let $pf \stackrel{\text{def}}{=} \{p\}A(S)\{q\}$ *be a proof outline for a program* S*, augmented with an auxiliary variable* $a_2.occ$ *as described above, such that* $p \rightarrow \neg a_2.occ$*. Let the precondition for action* a_1 *in this proof outline be denoted by* $pre(a_1)$ *and assume that* $pre(a_1) \rightarrow \neg a_2.occ$ *is a valid formula. Then* $\langle S \rangle$ **psat** $a_1 \twoheadrightarrow a_2$ *is a valid correctness formula.*

The last theorem begs the question why one would use weak precedence relations to verify programs. After all, the theorem suggests that you *already* must have a proof outline in order to show that certain precedence relations hold. However, the strategy that we propose is as follows:

- First, set up a proof outline that is used *only* to verify weak precedence properties. That is, this proof outline does not show the correctness of the program with respect its specification by means of the pre- and postconditions p and q. Therefore, this proof outline can usually be fairly simple.
- Second, after having verified the necessary precedence properties, forget the first proof outline, and use the precedence properties in combination with one of the communication closed layers laws, that is discussed later on in this paper, to transform the original distributed program S_D into a simplified layered version S_L.
- Finally, use new proof outlines to show the correctness of the layered version S_L with respect to the original specification of the program.

In practice the proof outlines needed to verify precedence properties follow a few predefined patterns.

Theorem 6 (Ordering caused by sequential composition).
Assume that S_1 *is a program containing action* a_1 *and* S_2 *is a program containing* a_2*. Let* $C[\cdot]$ *be a program context such that within a program of the form* $C[S]$*, the* S *component is not inside the body of a loop construct. Then:*

$$\langle C[S_1 \,;\, S_2] \rangle \quad \textbf{psat} \quad a_1 \twoheadrightarrow a_2.$$

Weak precedence is *not* a transitive relation, that is, from $a_1 \twoheadrightarrow a_2$ and $a_2 \twoheadrightarrow a_3$ it does *not* follow that $a_1 \twoheadrightarrow a_3$. To see this, consider some execution sequence with no a_2 occurrences at all. Then $a_1 \twoheadrightarrow a_2$ and $a_2 \twoheadrightarrow a_3$ both hold, trivially. But of course $a_1 \twoheadrightarrow a_3$ need not hold. This "counterexample" suggests a useful theorem. Assume that the system satisfies the formula $\Diamond a_2$, so we know that in every execution sequence at least one a_2 event occurs. Informally, we say that "a_2 is *unavoidable*". Now if in this situation the relations $a_1 \twoheadrightarrow a_2$ and $a_2 \twoheadrightarrow a_3$ are valid, then we may conclude that $a_1 \twoheadrightarrow a_3$ is valid too. To prove this, assume to the contrary that there would exist an execution sequence where some a_1 event occurs that is preceded by some a_3 event. We know that some a_2 event is bound to occur also. By the $a_1 \twoheadrightarrow a_2$ and $a_2 \twoheadrightarrow a_3$ relations, it follows that this a_2 event should follow the a_1 event and precede the a_3 event. This contradicts

the assumption that the a_3 event would precede the a_1 event, and so, by reductio ad absurdum, we conclude that $a_1 \twoheadrightarrow a_3$ must be valid. We formulate a theorem that is a simple extension of what we have shown:

Theorem 7 (Properties of action ordering).
*Let a_1 and a_2 be action names and let $\tilde{a}_i$ where $i \in I$ for some finite index set I be action names. If S **psat** $(a_1 \twoheadrightarrow \tilde{a}_i$, S **psat** $\tilde{a}_i \twoheadrightarrow a_2$, for all $i \in I$, and S **psat** $(\Diamond a_1 \wedge \Diamond a_2) \rightarrow \bigvee_{i\in I} \Diamond \tilde{a}_i$, then S **psat** $a_1 \twoheadrightarrow a_2$ holds.*

We do not aim at formulating a complete proof system for temporal logic here. Rather, we provide a few simple rules that we need, mainly in combination with rules for order constraints as above.

Theorem 8 (Properties of $\Diamond$).
- $-\ a : act$ **psat** $\Diamond a$
- *If* S_1 **psat** $\Diamond a$ *then, for any* S_2, $S_1 \,;\, S_2$ **psat** $\Diamond a$ *and* $S_2 \,;\, S_1$ **psat** $\Diamond a$
- *If* S_1 **psat** $\Diamond a$ *then, for any* S_2, $S_1 \parallel S_2$ **psat** $\Diamond a$ *and* $S_2 \parallel S_1$ **psat** $\Diamond a$
- *If* S_i **psat** $\Diamond a_i$ *for* $i = 1..n$, *then* **if** $[]_{i=1}^{n} b_i \rightarrow S_i$ **fi** **psat** $\bigvee_{i=1..n} \Diamond a_i$.

5 The Communication Closed Layers laws

We discuss a group of related program transformations, collectively referred to as "communication closed layers laws" (CCL-laws). The phrase "communication closed" stems from a paper by T. Elrad and N. Francez [EF82], where communication closedness of CSP programs was introduced. We refer here to a more general setting where we ask under which conditions an arbitrary program S_L of the form $[S_{0,0} \parallel S_{0,1}] \,;\, [S_{1,0} \parallel S_{1,1}]$ can be considered equivalent to a program S_D of the form $[S_{0,0}; S_{1,0} \parallel S_{0,1}; S_{1,1}]$. The interest lies in program development where one starts off with *sequentially layered* programs S_L, which are considered simpler to design and verify than *distributed programs* S_D. After S_L has been verified, it is then *transformed* into the form S_D. There are a number of related transformation laws, where the variation originates from the following factors:

- The variation in communication mechanisms: CSP style communication or shared variable based communication.
- The side conditions under which the equivalence holds: there is a choice between simple syntactic conditions and more complex verification conditions, where the latter are applicable in more general situations than the simple ones.
- The notion of equivalence that we are interested in: We require that "equivalent" programs have the same input-output relation when we consider them as state transformers. This implies that equivalent programs satisfy the same Hoare formulae for partial correctness. A much stronger requirement is that programs are considered equivalent if they have the same reactive sequence semantics. If S_1 and S_2 are equivalent in this stronger sense then any temporal logic formula satisfied by S_1 would also be satisfied by S_2 and vice

versa. Moreover, we know that semantic equality yields not only an equivalence but even a congruence. Despite the obvious advantages of program congruence there is also a serious disadvantage: if "equivalent" programs are required to have identical computations, then not much room for interesting transformations remains. In this paper we focus on equivalence in the sense of identical input-output behavior.

Definition 9 (IO-equivalence).
We define IO-equivalence, denoted by $S_1 \stackrel{IO}{=} S_2$, iff $\mathcal{O}[\![S_1]\!] = \mathcal{O}[\![S_2]\!]$.

The soundness of various transformation laws can be shown by considering action labeled computations $\mathcal{RA}[\![S]\!]$ of programs. From the definition of IO-equivalence and the definitions of the semantic functions it follows that IO-equivalence between S_1 and S_2 holds iff the following is true: For any computation $\eta \in \mathcal{RA}[\![\langle S_1 \rangle]\!]$, respectively, $(\mathcal{RA}[\![\langle S_2 \rangle]\!])$ of the form

$$\langle \sigma_0 \stackrel{a_0}{\to} \sigma_1 \rangle \langle \sigma_1 \stackrel{a_1}{\to} \sigma_2 \rangle \ldots \langle \sigma_{n-1} \stackrel{a_{n-1}}{\to} \sigma_n \rangle,$$

there is a computation $\eta' \in \mathcal{RA}[\![\langle S_2 \rangle]\!]$, respectively, $(\mathcal{RA}[\![\langle S_1 \rangle]\!])$ of the form

$$\langle \sigma_0 \stackrel{a'_0}{\to} \sigma'_1 \rangle \langle \sigma'_1 \stackrel{a'_1}{\to} \sigma'_2 \rangle \ldots \langle \sigma'_{n-1} \stackrel{a'_{n-1}}{\to} \sigma_n \rangle.$$

That is, for any computation η starting with an initial state σ_0 and terminating in a state σ_n that is possible for one of the two programs, there is an *IO-equivalent* computation η', i.e. with the same initial and final state, for the other program. Note that IO-equivalent computations need not go through the same intermediate states, and that they need not be of the same length.

Theorem 10 (Sequential contexts).
Let S and S' be IO-equivalent programs and let $C[\cdot]$ be a context such that S is not a statement within a parallel component of $C[S]$, i.e. S is not within the scope of a parallel composition operator of $C[S]$. Then $C[S] \stackrel{IO}{=} C[S']$

Informally this means that although IO-equivalence is not a congruence, it can be treated as a congruence within sequential contexts.

5.1 CCL laws for shared variables

The CCL laws for shared variables are based on the fairly simple idea of *syntactically commuting actions*. First, let a be the name of some $\langle b \to \bar{x} := \bar{e} \rangle$ action. We define the set of read variables $R(a) \stackrel{\text{def}}{=} var(b) \cup var(\bar{e})$, and the set of write variables $W(a) \stackrel{\text{def}}{=} \{\bar{x}\}$. We say that two actions $a_1 \equiv \langle b_1 \to \bar{x}_1 := \bar{e}_1 \rangle$ and $a_2 \equiv \langle b_2 \to \bar{x}_2 := \bar{e}_2 \rangle$ are *syntactically commuting* if the following three conditions are satisfied:

1. $W(a_1) \cap R(a_2) = \emptyset$,
2. $W(a_2) \cap R(a_1) = \emptyset$.

3. $W(a_1) \cap W(a_2) = \emptyset$.

Actions which do not syntactically commute are said to be *in conflict* and, depending on which condition above is violated, we speak of read-write or write-write conflicts. When a_1 and a_2 are in conflict we denote this by $a_1 — a_2$ and otherwise, i.e. when they commute syntactically, we denote this by $a_1 \not— a_2$.

Definition 11 (Concurrent actions).
Two actions a_1 and and a_2 occurring in S are called *concurrent* actions if there are two different parallel components S_1 and S_2 of S such that a_1 occurs in S_1 and a_2 occurs in S_2. This means there exists statements S_1 and S_2 such that the statement $[S_1 \parallel S_2]$ occurs within S, a_1 is an action of S_1, and a_2 is an action of S_2. □

A principal property of commuting concurrent actions is the following:

Lemma 12 (Commuting actions).
Let a_1 and a_2 be concurrent actions of a closed program $\langle S \rangle$ and suppose that a_1 and a_2 are syntactically commuting actions, i.e. $a_1 \not— a_2$. Let $\eta \in \mathcal{RA}[\![\langle S \rangle]\!]$ be a computation of $\langle S \rangle$, of the form:

$$\langle \sigma_0 \stackrel{a_0}{\to} \sigma_1 \rangle \ldots \langle \sigma_{i-1} \stackrel{a_{i-1}}{\to} \sigma_i \rangle \langle \sigma_i \stackrel{a_i}{\to} \sigma_{i+1} \rangle \langle \sigma_{i+1} \stackrel{a_{i+1}}{\to} \sigma_{i+2} \rangle \langle \sigma_{i+2} \stackrel{a_{i+2}}{\to} \sigma_{i+3} \rangle \ldots$$

Assume that for some index i, event a_i is an occurrence of action a_1, and a_{i+1} is an occurrence of a_2. Let η' be defined as η with a_i and a_{i+1} exchanged, i.e. of the form:

$$\langle \sigma_0 \stackrel{a_0}{\to} \sigma_1 \rangle \ldots \langle \sigma_{i-1} \stackrel{a_{i-1}}{\to} \sigma_i \rangle \langle \sigma_i \stackrel{a_{i+1}}{\to} \sigma'_{i+1} \rangle \langle \sigma'_{i+1} \stackrel{a_i}{\to} \sigma_{i+2} \rangle \langle \sigma_{i+2} \stackrel{a_{i+2}}{\to} \sigma_{i+3} \rangle \ldots$$

(Note that η' determines the same sequence of states except for state σ'_{i+1} that in general will differ from σ_{i+1}.) We claim that η' too is a computation of $\langle S \rangle$, that is: $\eta' \in \mathcal{RA}[\![\langle S \rangle]\!]$.

The next theorem is the first example of a CCL law. It is based on independence of program fragments, where we define $S_1 \not— S_2$ as: for all actions a_1 occurring in S_1 and all actions a_2 occurring in S_2 we have that $a_1 \not— a_2$.

Theorem 13 Communication Closed Layers-1.
Let S_L and S_D be programs defined thus:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} [S_{0,0} \parallel \cdots \parallel S_{0,m}] \\ ; \\ \vdots \qquad \vdots \qquad \vdots \\ ; \\ [S_{n,0} \parallel \cdots \parallel S_{n,m}] \end{array}$$

and

$$S_D \stackrel{\text{def}}{=} \left[\begin{array}{c|c|c} S_{0,0} & \cdots & S_{0,m} \\ ; & \cdots & ; \\ \vdots & \vdots & \vdots \\ ; & \cdots & ; \\ S_{n,0} & \cdots & S_{n,m} \end{array}\right]$$

Assume that $S_{i,j} \not\frown S_{i',j'}$ *for* $i \neq i'$ *and* $j \neq j'$, *then* $S_L \stackrel{IO}{=} S_D$.

The intuitive justification of this law is simple: Consider the simple case

$$S_L \stackrel{\text{def}}{=} [S_{0,0} \parallel S_{0,1}] \,;\, [S_{1,0} \parallel S_{1,1}]$$

and

$$S_D \stackrel{\text{def}}{=} [S_{0,0} \,;\, S_{1,0} \parallel S_{0,1} \,;\, S_{1,1}]$$

It will be clear that any computation for $\langle S_L \rangle$ is also a possible computation for $\langle S_D \rangle$. The reverse does in general not hold, since, for instance, $S_{1,0}$ actions can occur as soon as the $S_{0,0}$ part has terminated, so possibly *before* the $S_{0,1}$ part has terminated. However, since it is assumed in the above theorem that $S_{0,1}$ actions and $S_{1,0}$ actions commute syntactically, such computations are equivalent to computations where $S_{1,0}$ actions occur only *after* all $S_{0,1}$ actions, that is, such $\langle S_D \rangle$ computations are equivalent to some $\langle S_L \rangle$ computation. The theorem follows as a simple corollary from our second CCL law, that we discuss next. Assume that we have programs S_L and S_D as above, but this time assume, for instance, the condition $S_{1,0} \not\frown S_{0,1}$ does *not* hold. That is, there are actions a_1 in $S_{1,0}$ and a_2 in $S_{0,1}$ such that $a_1 — a_2$. In general, an S_D computation where an instance of a_2 precedes an instance of a_1 will not be equivalent to any S_L computation. Let us assume however that (for all such pairs of conflicting actions) we can show that S_D satisfies the formula $a_1 \twoheadrightarrow a_2$; that is, a_2 instances simply do not precede a_1 instances. Then reasoning as above shows again that S_D computations can be shown to be equivalent to S_L computations via the technique of permuting occurrences of commuting actions.

We introduce some extra notation in order to formulate our second version of the CCL law. For actions a_1 and a_2 let $a_1 \stackrel{C}{\twoheadrightarrow} a_2$ abbreviate the formula $a_1 \twoheadrightarrow a_2$ if $a_1 — a_2$, and let it denote "*true*" otherwise. We extend the notation to programs: $S_1 \stackrel{C}{\twoheadrightarrow} S_2$ abbreviates the following formula:

$$\bigwedge \{a_1 \stackrel{C}{\twoheadrightarrow} a_2 \mid a_1 \text{ occurs in } S_1, a_2 \text{ occurs in } S_2\}$$

Informally, $S_1 \stackrel{C}{\twoheadrightarrow} S_2$ expresses that a_1 events precede those a_2 events that are conflicting with them.

Theorem 14 (Communication Closed Layers-2).
Let S_L and S_D be programs defined thus:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} [S_{0,0} \parallel \cdots \parallel S_{0,m}] \\ ; \\ \vdots \quad\quad \vdots \quad\quad \vdots \\ ; \\ [S_{n,0} \parallel \cdots \parallel S_{n,m}] \end{array}$$

and

$$S_D \stackrel{\text{def}}{=} \left[\begin{array}{c|c|c|c} S_{0,0} & \cdots & S_{0,m} \\ ; & \cdots & ; \\ \vdots & \vdots & \vdots \\ ; & \cdots & ; \\ S_{n,0} & \cdots & S_{n,m} \end{array}\right]$$

Assume that S_D **psat** *$(S_{i,j} \overset{C}{\twoheadrightarrow} S_{i',j'})$ for all $i < i'$ and $j \neq j'$. Then $S_L \overset{IO}{=} S_D$.*

Example 1 (Communication closed layers).
Consider the program S given by

$$S \stackrel{\text{def}}{=} \left[\begin{array}{c|c} z := 2 & x := 2 \\ ; & ; \\ P_1 : P(s) & V_1 : V(s) \\ ; & ; \\ w := 1 & v := 1 \\ ; & ; \\ P_2 : P(s) & V_2 : V(s) \\ ; & ; \\ z := x + 1 & v := w + 1 \end{array}\right]$$

One can show that, under the implicit assumption that initially s holds, S satisfies the ordering:

$$\langle S \rangle \quad \textbf{psat} \quad P_1 \twoheadrightarrow V_1 \twoheadrightarrow P_2 \twoheadrightarrow V_2.$$

Moreover using the theorems above one can deduce

$$\langle S \rangle \quad \textbf{psat} \quad \Diamond P_1 \wedge \Diamond V_1 \wedge \Diamond P_2 \wedge \Diamond V_2$$

Introducing action labels for conflicting events and the ordering and eventualities, cf. theorem 6 and 7, one can easily deduce that S is IO-equivalent with the layered program $S_1 \,;\, S_2$ with

$$S_1 \stackrel{\text{def}}{=} \left[\begin{array}{c|c} z := 2 & x := 2 \\ ; & ; \\ P_1 : P(s) & V_1 : V(s) \\ ; & ; \\ w := 1 & v := 1 \end{array}\right]$$

and

$$S_2 \stackrel{\text{def}}{=} \left[\begin{array}{c} P_2 : P(s) \\ ; \\ z := x+1 \end{array} \middle\| \begin{array}{c} V_2 : V(s) \\ ; \\ v := w+1 \end{array}\right].$$

This IO-equivalent layered version readily implies that S satisfies

$$\models \{s\} S \{z = 3 \ \wedge \ v = 2\}$$

5.2 CCL law for communication based programs

In this section we consider distributed programs in which the components only communicate with each other through send and receive actions. The CCL law for such communication based programs is based on the notion of communication closed layers. Assume that we have programs

$$S_L \stackrel{\text{def}}{=} [S_{0,0} \parallel S_{0,1}] \,;\, [S_{1,0} \parallel S_{1,1}],$$

and

$$S_D \stackrel{\text{def}}{=} [S_{0,0} \,;\, S_{1,0} \parallel S_{0,1} \,;\, S_{1,1}].$$

The CCL law states that $S_L \stackrel{IO}{=} S_D$, provided that S_D **psat** $(S_{0,0} \stackrel{C}{\twoheadrightarrow} S_{1,1}) \wedge (S_{0,1} \stackrel{C}{\twoheadrightarrow} S_{1,0})$. In general, the conditions on S_D for this property to hold are not always so easy to verify. However, assume that communication between $(S_{0,0} \,;\, S_{1,0})$ and $(S_{0,1} \,;\, S_{1,1})$ is by means of explicit **send** and **receive** commands only, i.e. there are no shared variables except for variables associated with communication channels. We call a single layer, such as $L_0 \stackrel{\text{def}}{=} [S_{0,0} \parallel S_{0,1}]$, *communication closed* if it has the following properties:

- For any channel c, one of $S_{0,0}$ and $S_{0,1}$ contains all the **send** actions, and the other one contains all the **receive** actions.
- For each **send** event for some channel c, there is a "matching" **receive** event.

By "matching" we refer to a simple counting argument: the number of **send** events for channel c is the same as the number of **receive** events for c. We claim that when both $L_0 \stackrel{\text{def}}{=} [S_{0,0} \parallel S_{0,1}]$ and $L_1 \stackrel{\text{def}}{=} [S_{1,0} \parallel S_{1,1}]$ are communication closed layers, then the side conditions of the form $(S_{0,0} \stackrel{C}{\twoheadrightarrow} S_{1,1})$ and $(S_{0,1} \stackrel{C}{\twoheadrightarrow} S_{1,0})$ are satisfied. In general, the number of **send** or **receive** actions actually executed might depend on the initial state for a particular layer. Before we can deal with this extra complication we must discuss IO-equivalence and transformations that rely on preconditions.

6 Assertion based program transformations

We introduce generalizations of the communication closed layers laws that depend on side conditions that are based on *assertions*. To be precise, we introduce laws that are of the form "S_1 is equivalent to S_2, provided these programs start in an initial state satisfying precondition *pre*". Let us assume that we have a program of the form $C[S_1]$, where $C[\cdot]$ is the program context of S_1. If we want to transform this program into $C[S_2]$ by applying the transformation law, then it will be clear that one has to show that, within the given context $C[S_1]$, the required precondition *pre* holds whenever S_1 starts executing. We first introduce a generalized version of the operation semantics, which is in essence captures the semantics of a program provided that it starts in a state satisfying a given precondition. Formally speaking, we define a semantic function $\mathcal{O}[\![(pre, S)]\!]$ that maps pairs consisting of a precondition and a program to sets of state pairs. We use the notation $\mathcal{O}[\![\{pre\}S]\!]$, rather than $\mathcal{O}[\![(pre, S)]\!]$:

Definition 15 (Precondition based semantics).

$$\mathcal{O}[\![\{pre\}S]\!] = \{(\sigma, \sigma') \mid \sigma \models pre \wedge \sigma' = val(\eta) \ \wedge \ \eta \in \mathcal{RA}[\![\langle S \rangle]\!]\,\sigma\}$$

□

Definition 16 (Precondition based IO-equivalence).
We define precondition based IO-equivalence between programs, denoted by $\{p_1\}S_1 \stackrel{IO}{=} \{p_2\}S_2$ iff $\mathcal{O}[\![\{p_1\}S_1]\!] = \mathcal{O}[\![\{p_2\}S_2]\!]$. □

For the next series of definitions we assume that extra auxiliary variables "*c.sent*" and "*c.received*" have been introduced for all relevant channels c, acting as counters for the number of times that **send** and **receive** actions have been executed. The **send** and **receive** abbreviations incorporating updates to these auxiliary variables then become as follows:

$$\mathbf{send}(c, e) \stackrel{\text{def}}{=} \langle \neg c.full \rightarrow c.full, c.buf, c.sent := true, e, c.sent + 1 \rangle$$

$$\mathbf{receive}(c, x) \stackrel{\text{def}}{=} \langle c.full \rightarrow c.full, x, c.received := false, c.buf, c.received + 1 \rangle.$$

We assume that the preconditions of Hoare formulae for closed programs implicitly conjuncts of the form $\neg c.full$ and $c.sent = c.received = 0$, denoting that initially all channels are empty, and no messages have been sent.
In previous sections we have dealt with communication closedness for shared variable programs. We now reformulate this notion for programs that rely solely on **send** and **receive** commands for communication between parallel components. Formally speaking we have defined **send** and **receive** as abbreviations of shared variable actions; our definition below is consistent with the definition of communication closedness that we gave *for shared variables*. A new aspect, that was not discussed for shared variables, is the introduction of preconditions.

Definition 17 (Communication closedness).
A program S with precondition pre is called communication closed for channel c iff the following Hoare formula is valid:

$$\{pre \wedge c.sent = c.received\}\ S\ \{c.sent = c.received\}.$$

A program or layer with precondition pre is called communication closed if it is communication closed for all of its channels. □

Theorem 18 (Communication Closed Layers-3).
Let S_L and S_D be communication based programs, and let S_L be annotated thus:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} \{p_0\} \\ [S_{0,0} \parallel \cdots \parallel S_{0,m}] \\ \{p_1\} \\ ; \\ \vdots \qquad \vdots \qquad \vdots \\ ; \\ \{p_n\} \\ [S_{n,0} \parallel \cdots \parallel S_{n,m}] \end{array}$$

and

$$S_D \stackrel{\text{def}}{=} \left[\begin{array}{c|c|c} S_{0,0} & \cdots & S_{0,m} \\ ; & \cdots & ; \\ \vdots & \vdots & \vdots \\ ; & \cdots & ; \\ S_{n,0} & \cdots & S_{n,m} \end{array}\right]$$

Assume that each layer with precondition

$$L_i \stackrel{\text{def}}{=} \{p_i\}\,[S_{i,0} \parallel \cdots \parallel S_{i,m}]$$

is communication closed. Then $\{p_0\}\,S_L \stackrel{IO}{=} \{p_0\}\,S_D$.

6.1 Loop distribution

In this section we will derive certain laws for distributing loops over parallel composition. Conditions will be given under which a loop of the form

$$\textbf{do}\ b \rightarrow [S_1 \parallel S_2]\ \textbf{od}$$

can be considered equivalent to a distributed loop of the form:

$$[\textbf{do}\ b_1 \rightarrow S_1\ \textbf{od} \parallel \textbf{do}\ b_2 \rightarrow S_2\ \textbf{od}]$$

Such a transformation is not valid in general, but depends on the relation between the guard b and the guards b_1 and b_2, and an appropriate loop invariant. The loop bodies S_1 and S_2 should satisfy the CCL-condition. And finally variables of the guard b_i should be local variables of the loop body S_i.

Theorem 19 (Loop transformations).
Consider a program $\mathbf{do}\ b\ \rightarrow\ [S_1\ \|\ S_2]\ \mathbf{od}$*, guards* b_1, b_2*, and assertions* p, q, I *with the following properties.*

1. *I is a loop invariant, i.e.*

$$\models \{I\ \wedge\ b\}[S_1\ \|\ S_2]\{I\},$$

2. $\models p\ \Rightarrow\ I$,
3. *The variables of b_i are contained in the local variables of S_i, $i = 1, 2$, and moreover the following is valid:*

$$\models I\ \Rightarrow\ (b\ \leftrightarrow\ b_1)\ \wedge\ (b\ \leftrightarrow\ b_1).$$

4. $\{I\}[S_1\ \|\ S_2]$ *is communication closed. (In particular we assume that no shared variables are used except for those implementing the channels.)*

Then

$$\{p\}\ \mathbf{do}\ b\ \rightarrow\ [S_1\ \|\ S_2]\ \mathbf{od}$$
$$\overset{IO}{=}$$
$$\{p\}\ [\mathbf{do}\ b_1\ \rightarrow\ S_1\ \mathbf{od}\ \|\ \mathbf{do}\ b_2\ \rightarrow\ S_2\ \mathbf{od}]$$

and $\{p\}\ [\mathbf{do}\ b_1\ \rightarrow\ S_1\ \mathbf{od}\ \|\ \mathbf{do}\ b_2\ \rightarrow\ S_2\ \mathbf{od}]$ *is communication closed.*

7 Set partitioning revisited

In this section a variation of the well known set partitioning algorithm is developed. We start with a provably correct top-level layered system, and afterwards transform this layered system, meanwhile preserving correctness, to a distributed version. The pre- and post-specification of the set partioning algorithm is given by

$$\{S = S_0 \neq \emptyset\ \wedge\ T = T_0 \neq \emptyset\ \wedge\ S\ \cap\ T = \emptyset\}$$
$$P$$
$$\{|\ S\ |=|\ S_0\ |\ \wedge\ |\ T\ |=|\ T_0\ |\ \wedge S\ \cup\ T = S_0\ \cup\ T_0\ \wedge\ max(S) < min(T)\}$$

The initial version of this algorithm is based on a shared variables:

$$P_{sv} \overset{\text{def}}{=} \left\{ \begin{array}{lll} [max := \max(S) & \| & min := \min(T)] \\ ; & & \\ \mathbf{do}\ max > min\ \rightarrow & & \\ \quad [S := (S - \{max\}) \cup \{min\} & \| & T := (T - \{min\}) \cup \{max\}] \\ ; & & \\ [max := \max(S) & \| & min := \min(T)] \\ \mathbf{od} & & \end{array} \right.$$

One can easily show that the above program P_{sv} is correct with respect to the desired pre- and post-specification. Since we are aiming at a distributed version,

where *sharing* of variables like min and max is not possible, we introduce some extra variables mn and mx that, in the distributed version, will be used for maintaining local copies of min and max. This leads to the following program, which will be the starting point of the design and transformation strategy.

$$
P_{init} \stackrel{\text{def}}{=} \left\{ \begin{array}{l}
\begin{array}{ccc} [max := \max(S) & \| & min := \min(T)] \\ ; & & \\ [[\mathbf{send}(C, max) \ \| \ \mathbf{receive}(D, mn)] & \| & [\mathbf{receive}(C, mx) \ \| \ \mathbf{send}(D, min)]] \end{array} \\
\mathbf{do}\ max > min \ \rightarrow \\
\left[\begin{array}{ccc} [S := (S - \{max\}) \cup \{mn\} & \| & T := (T - \{min\}) \cup \{mx\}] \\ ; & & \\ [max := \max(S) & \| & min := \min(T)] \\ ; & & \\ [[\mathbf{send}(C, max) \ \| \ \mathbf{receive}(D, mn)] & \| & [\mathbf{receive}(C, mx) \ \| \ \mathbf{send}(D, min)]] \end{array} \right] \\
\mathbf{od}
\end{array} \right.
$$

The partial correctness of the above program can be shown using the following loop invariant:

$$
I \stackrel{\text{def}}{=} \mid S \mid = \mid S_0 \mid \ \wedge \ \mid T \mid = \mid T_0 \mid \ \wedge S \cup T = S_0 \cup T_0 \ \wedge \\
mn = min = \min(T) \ \wedge \ mx = max = \max(S).
$$

The next step is that we we transform the initialization phase, preceding the loop.

$$
\begin{array}{ccc} [max := \max(S) & \| & min := \min(T)] \\ ; & & \\ [[\mathbf{send}(C, max) \ \| \ \mathbf{receive}(D, mn)] & \| & [\mathbf{receive}(C, mx) \ \| \ \mathbf{send}(D, min)]] \end{array}.
$$

One easily checks that the premises of the communication based CCL law, are satisfied. Hence this initialization phase can be transformed into the following IO-equivalent program:

$$
\left[\begin{array}{c} max := \max(S) \\ ; \\ [\mathbf{send}(C, max) \ \| \ \mathbf{receive}(D, mn)] \end{array} \Big\| \begin{array}{c} min := \min(T) \\ ; \\ [\mathbf{receive}(C, mx) \ \| \ \mathbf{send}(D, min)] \end{array} \right]
$$

Next we will transform the loop body

$$
\left[\begin{array}{ccc} [S := (S - \{max\}) \cup \{mn\} & \| & T := (T - \{min\}) \cup \{mx\}] \\ ; & & \\ [max := \max(S) & \| & min := \min(T)] \\ ; & & \\ [[\mathbf{send}(C, max) \ \| \ \mathbf{receive}(D, mn)] & \| & [\mathbf{receive}(C, mx) \ \| \ \mathbf{send}(D, min)]] \end{array} \right]
$$

to a distributed version. This loop body also satisfies the premises of the communication based CCL law, theorem 18. Therefore this loop body can be trans-

formed into the following IO-equivalent program $[B_1 \parallel B_2]$, where:

$$B_1 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} S := (S - \{max\}) \cup \{mn\} \\ ; \\ max := \max(S) \\ ; \\ [\mathbf{send}(C, max) \parallel \mathbf{receive}(D, mn)] \end{array} \right.$$

and

$$B_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} T := (T - \{min\}) \cup \{mx\} \\ ; \\ min := \min(T) \\ ; \\ [\mathbf{send}(D, min) \parallel \mathbf{receive}(C, mx)] \end{array} \right.$$

Finally we want to distribute the guard $\mathbf{do}\ max < min\ \rightarrow [B_1 \parallel B_2]\ \mathbf{od}$ over the parallel composition. As explained in the section on loop distribution, this can be done if we are able to express for each parallel component the global guard in terms of local variables, and if the layer $B_1 \parallel B_2$ is communication closed. In this particular case we can use the auxiliary variables mn, a copy of min, and mx, a copy of max, cf. the loop invariant I. Hence we have at the start of the loop

$$I \ \rightarrow\ (max < min \leftrightarrow mx < min) \ \wedge\ (max < min \leftrightarrow max < mn)$$

The second and fourth inequality are expressed using the local variables of B_2 and B_1, respectively. Moreover one can easily check the communication closedness of the layer $B_1 \parallel B_2$. So we can distribute the guard over the parallel composition. This yields the following IO-equivalent, distributed and correct version of the loop $\mathbf{do}\ max < min\ \rightarrow [B_1 \parallel B_2]\ \mathbf{od}$:

$$\left[\begin{array}{l} \mathbf{do}\ max < mn\ \rightarrow \\ \quad B_1 \\ \mathbf{od} \end{array} \right\| \left[\begin{array}{l} \mathbf{do}\ mx < min\ \rightarrow \\ \quad B_2 \\ \mathbf{od} \end{array} \right]$$

As an intermediate result we have obtained the following program

$$P_{inter_1} \stackrel{\text{def}}{=} \left\{ \begin{array}{cc} [Init_1 & \| \quad Init_2] \\ & ; \\ \left[\begin{array}{l} \mathbf{do}\ max < mn\ \rightarrow \\ \quad B_1 \\ \mathbf{od} \end{array} \right. & \left\| \left[\begin{array}{l} \mathbf{do}\ mx < min\ \rightarrow \\ \quad B_2 \\ \mathbf{od} \end{array} \right] \right. \end{array} \right.$$

with:

$$Init_1 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} max := \max(S) \\ ; \\ [\mathbf{send}(C, max) \parallel \mathbf{receive}(D, mn)] \end{array} \right.$$

and

$$Init_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} min := \min(T) \\ ; \\ [\mathbf{send}(D, min) \parallel \mathbf{receive}(C, mx)] \end{array} \right.$$

Again by theorem 19 the distributed loop (layer)

$$\{I\}[\mathbf{do}\ max < mn \ \rightarrow\ B_1\ \mathbf{od} \parallel \mathbf{do}\ mx < min \ \rightarrow\ B_2\ \mathbf{od}]$$

is communication closed. So we can apply the precondition based Communication Closed Layers transformation, theorem 18, to the the distributed initialization phases and the distributed loops yielding the following distributed program

$$P_{impl} \stackrel{\text{def}}{=} \left[\begin{array}{c} Init_1 \\ ; \\ \mathbf{do}\ max < mn \ \rightarrow \\ B_1 \\ \mathbf{od} \end{array} \middle\| \begin{array}{c} Init_2 \\ ; \\ \mathbf{do}\ mx < min \ \rightarrow \\ B_2 \\ \mathbf{od} \end{array} \right]$$

Invoking the definition of I_1, I_2, B_1 and B_2 results in

$$P_{impl} \stackrel{\text{def}}{=} \left[\begin{array}{c} max := \max(S) \\ ; \\ [\mathbf{send}(C, max) \parallel \mathbf{receive}(D, mn)] \\ ; \\ \mathbf{do}\ max < mn \ \rightarrow \\ S := (S - \{max\}) \cup \{mn\} \\ ; \\ max := \max(S) \\ ; \\ [\mathbf{send}(C, max) \parallel \mathbf{receive}(D, mn)] \\ \mathbf{od} \end{array} \middle\| \begin{array}{c} min := \min(T) \\ ; \\ [\mathbf{send}(D, min) \parallel \mathbf{receive}(C, mx)] \\ ; \\ \mathbf{do}\ mx < min \ \rightarrow \\ T := (T - \{min\}) \cup \{mx\} \\ ; \\ min := \min(T) \\ ; \\ [\mathbf{send}(D, mn) \parallel \mathbf{receive}(C, mx)] \\ \mathbf{od} \end{array} \right]$$

And we can conclude that the above program P_{itimpl} is a partial correct implementation of the set partioning algorithm.

References

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[BS92] R. Back and K. Sere. Superposition refinement of parallel algorithms. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques IV.* IFIP Transaction C-2, North Holland, 1992.

[CG88] C. Chou and E. Gafni. Understanding and verifying distributed algorithms using stratified decomposition. In *Proceeding 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[CM88] R. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[EF82] T. Elrad and N. Francez. Decomposition of distributed programs into communication closed layers. *Science of Computer Programming*, 2:155–173, 1982.

[FPZ93] M. Fokkinga, M. Poel, and J. Zwiers. Modular completeness for communication closed layers. In Eike Best, editor, *Proceedings CONCUR '93, LNCS 715*, pages 50–65. Springer-Verlag, 1993.

[GHS83] R. Gallager, P. Humblet, and P. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM TOPLAS*, 5(1):66–77, Jan 1983.

[GS86] R. Gerth and L. Shira. On proving communication closedness of distributed layers. In *Proceedings 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1986.

[JPXZ94] W. Janssen, M. Poel, Q. Xu, and J. Zwiers. Layering of real-time distributed processes. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proceedings Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863*, pages 393–417. Springer-Verlag, 1994.

[JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *Proceedings of CONCUR '91, LNCS 527*, pages 298–316. Springer-Verlag, 1991.

[JZ92a] W. Janssen and J. Zwiers. From sequential layers to distributed processes, deriving a distributed minimum weight spanning tree algorithm, (extended abstract). In *Proceedings 11th ACM Symposium on Principles of Distributed Computing*, pages 215–227. ACM, 1992.

[JZ92b] W. Janssen and J. Zwiers. Protocol design by layered decomposition, a compositional approach. In J. Vytopil, editor, *Proceedings Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 571*, pages 307–326. Springer-Verlag, 1992.

[Kat93] S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2), 1993.

[KP87] S. Katz and D. Peled. Interleaving set temporal logic. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing, Vancouver*, pages 178–190, 1987.

[KP92] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6(2), 1992.

[Lam83a] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5(2), 1983.

[Lam83b] L. Lamport. What good is temporal logic. In *Proc. IFIP 9th World Congress*, pages 657–668, 1983.

[Maz89] A. Mazurkiewicz. Basic notions of trace theory. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX workshop on Linear Time, Branching Time and Partial order in Logics and Models for Concurrency, LNCS 354*, pages 285–363. Springer-Verlag, 1989.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties. *Science of Computer Programming*, 4, 1984.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer-Verlag, 1992.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems – Safety.* Springer-Verlag, 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. on Foundations of Comp. Sci.*, pages 46–57, 1977.

[Pra86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 1(15):33–71, 1986.

[RW94] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In *Proceedings CONCUR '94, LNCS 836*, pages 226–241. Springer-Verlag, 1994.

[SdR87] F. Stomp and W.-P. de Roever. A correctness proof of a distributed minimum-weight spanning tree algorithm (extended abstract). In *Proceedings of the 7th ICDCS*, 1987.

[SdR89] F. Stomp and W.-P. de Roever. Designing distributed algorithms by means of formal sequentially phased reasoning. In J.-C. Bermond and M. Raynal, editors, *Proceedings of the 3rd International Workshop on Distributed Algorithms, Nice, LNCS 392*, pages 242–253. Springer-Verlag, 1989.

[SdR94] F. Stomp and W.-P. de Roever. A principle for sequential reasoning about distributed systems. *Formal Aspects of Computing*, 6(6):716–737, 1994.

[Sto89] F. Stomp. *Design and Verification of Distributed Network Algorithms: Foundations and Applications.* PhD thesis, Eindhoven University of Techology, 1989.

[Sto90] F. Stomp. A derivation of a broadcasting protocol using sequentially phased reasoning. In *Stepwise refinement of distributed systems, LNCS 430*, pages 696–730. Springer-Verlag, 1990.

[ZJ94] J. Zwiers and W. Janssen. Partial order based design of concurrent systems. In J. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX School/Symposium "A Decade of Concurreny", Noordwijkerhout, 1993, LNCS 803*, pages 622–684. Springer-Verlag, 1994.

[Zwi91] J. Zwiers. Layering and action refinement for timed systems. In de Bakker, Huizing, de Roever, and Rozenberg, editors, *Real-Time: Theory in Practice, LNCS 600*, pages 687–723. Springer-Verlag, 1991.

Compositional proof methods for concurrency: A semantic approach

F.S. de Boer[1] and W.-P. de Roever[2]

[1] Utrecht University, The Netherlands, email: frankb@cs.uu.nl,
[2] University of Kiel, Germany, email: wpr@informatik.uni-kiel.de

1 Introduction

This paper focusses on the mathematical theory of state-based reasoning about program constructs solely through specifications of their parts, without any reliance on their implementation mechanism. That is, the semantic foundations of compositional state-based reasoning about concurrency. The main advantages of a purely semantic approach are that:

- it highlights the very concept of compositional state-based reasoning about concurrency without any syntactic overhead, and
- it serves as a basis for the encoding of the program semantics and corresponding proof rules inside tools such as PVS which support program verifcation.

Referring to [4] for the full theory, the present paper illustrates the semantic approach for a particular case, namely that of synchronous communication. Our own motivation for developing this theory derives from three sources:

1. The dramatic simplification which such a semantical theory represents over earlier, syntactically formulated, theories for the same concepts, such as, for instance, those of Job Zwiers [5]; this simplification was a prerequisite for writing [4].
2. Confronted with the many tool-based theories for compositional reasoning about concurrency, and their applications, such as the PVS-based ones which Jozef Hooman contributed to [4] and the present volume, that of Leslie Lamport and Bob Kurshan's [3], or those mentioned in the contribution by Werner Damm, Amir Pnueli and al. to this volume, made us wonder which compositional theories these authors actually implemented inside their tools.
3. More generally, the relationship between operational and axiomatic semantics of programming languages, specifically, the construction of programming logics from a compositional semantics; this line of research was pioneered by Samson Abramski [1].

The approach which is followed in this paper is based on the inductive assertion method [2] which is a methodology for proving state-based transition diagrams correct. It consists of the construction of an *assertion network* by associating with each location of a transition diagram a (state) predicate and with each transition a *verification condition* on the predicates associated with the locations involved. Thus it reduces a statement of correctness of a transition

W.-P. de Roever, H. Langmaack, and A. Pnueli (Eds.): COMPOS'97, LNCS 1536, pp. 632-646, 1998.
Springer-Verlag Berlin Heidelberg 1998

diagram, which consists of a finite number of locations, to a corresponding finite number of verification conditions on predicates.

The inductive assertion method can be trivially generalized to concurrent transition diagrams by viewing a concurrent system as the product of its components and thus reducing it to a sequential system. However this global proof method leads to a number of verification conditions which is exponential in the number of components.

Compositional proof methods in general provide a reduction in the complexity of the verification conditions. In this paper we investigate the semantic foundations of a compositional proof method for concurrent transition diagrams based on synchronous message passing. Technically, we combine so-called *compositionally* inductive assertion networks for reasoning about the sequential parts of concurrent systems with compositional proof rules for deducing properties of the whole system, giving rigorous soundness and completeness proofs for the resulting theory. The basic idea of a compositionally inductive assertion network is the definition of the verification conditions in terms of a *logical* history variable which records the sequence of communications generated by the component (logical variables only occur in assertions, never in programs). The parallel combination of these compositionally inductive assertion networks is defined in terms of a simple semantic characterization of the variables and channels *involved in* a predicate. (The semantic notion "involved in" of a variable approximates the corresponding syntactic notion of occurrence.) More specifically, the notion of the channels involved in a predicate is defined in terms of a natural generalization of a *projection* operation on histories to predicates. These predicates themselves are viewed semantically as sets of states.

The introduction of a history variable as a logical variable is the main difference between noncompositional and compositional forms of reasoning (about concurrency). In noncompositional proof methods the program is annotated with assignments to so-called *auxiliary* variables which do not influence the control flow but only serve to enhance the expressive power of the assertions associated with the locations of a program. However, as analysis of compositional proof methods shows, this is not true for compositional proof methods. There, logical variables are introduced, which do not occur in any form inside a program, and only occur inside assertions. What happens is that the assignments to auxiliary variables are simulated at the logical level by appropriately defined verification conditions for the (inductive) assertion networks for the sequential components of a program. As a result, these logical variables simulate the local (communication) histories of those sequential components. Consequently, the main challenge in defining a compositional proof method for concurrent programs is the formulation of a sound (and complete) proof rule for parallel composition! For this to succeed it is mandatory that the specification for a to-be composed component only *involves* the variables and channels which are associated with that component, because such a rule combines the only locally verified information about the component into global information about the whole program, since, otherwise, the local specification could be invalidated by the other parallel components.

This is accomplished by *projecting* the (communication) histories encoded in these logical variables on the actions of that component. This explains also why compositional proof methods always involve some form of projection mechanism.

2 Synchronous transition diagrams

We consider (top-level) parallel systems the components of which communicate by means of synchronous message passing along unidirectional one-to-one channels. The control structure of a sequential component is described by means of a synchronous transition diagram.

Formally such a diagram is a quadruple (L, T, s, t), where L is a finite set of locations l, T is a finite set of transitions (l, a, l'), with a an instruction, and s and t are the entry and exit location, respectively. The precise nature of an instruction a is explained next.

Instructions involve either a guarded state-transformation or a guarded communication. Given an infinite set of variables VAR, with typical elements $x, y, z, \ldots$, the set of states Σ, with typical element σ, is given by $VAR \to VAL$, where VAL denotes the given underlying domain of values. Furthermore, let $CHAN$ be a set of channel names, with typical elements $c, d \ldots$. For $c \in CHAN$, e a semantic expression, i.e., $e \in \Sigma \to VAL$, execution of an *output* statement $c!e$ has to wait for execution of a corresponding *input* statement $c?x$, and, similarly, execution of an input statement has to wait for execution of a corresponding output statement. If there exists a computation in which both an input statement $c?x$ and an output statement $c!e$ are reached, this implies that communication can take place and the value of e in the current state is assigned to x. We often refer to an input or output statement as an *i/o statement*. In general an instruction a can have the following form:

1. A boolean condition $b \in \mathcal{P}(\Sigma)$ followed by a state transformation $f \in \Sigma \to \Sigma$, notation: $b \to f$. Transitions of this form are called *internal* transitions.
2. A guarded i/o-statement. There are two possibilities:
   (a) A guarded output statement $c!e$, notation: $b \to c!e$ $(b \in \mathcal{P}(\Sigma),\ e \in \Sigma \to VAL)$.
   (b) A guarded input statement $c?x$, notation: $b \to c?x$ $(b \in \mathcal{P}(\Sigma))$.
   These transitions are called *i/o transitions*.

Some terminology: In the sequel sets of states often will be called *predicates*. We have the following semantic characterization of the variables *involved* in a predicate and a state transformation. This characterization is an approximation of the corresponding syntactic notion of *occurrence* of a variable.

Definition 1. A predicate $\phi \in \mathcal{P}(\Sigma)$ *involves* the variables $\bar{x}$ if

- $\forall \sigma, \sigma' \in \Sigma.\ \sigma(\bar{x}) = \sigma'(\bar{x}) \Rightarrow (\sigma \in \phi \Leftrightarrow \sigma' \in \phi)$.

This condition expresses that the outcome of ϕ only depends on the variables $\bar{x}$. Similarly, a function $f \in \Sigma \to \Sigma$ involves the variables $\bar{x}$ if

- $\forall \sigma, \sigma' \in \Sigma.\ \sigma(\bar{x}) = \sigma'(\bar{x}) \Rightarrow f(\sigma)(\bar{x}) = f(\sigma')(\bar{x})$
- $\forall \sigma \in \Sigma, y \notin \bar{x}.\ f(\sigma)(y) = \sigma(y)$

The first condition expresses that if two states σ and σ' agree with respect to the variables $\bar{x}$, then so do their images under f. The second condition expresses that any other variable is not changed by f.

For $\bar{x}$ we will use the notation $f(\bar{x})$ and $\phi(\bar{x})$ to indicate that f and ϕ involves the variables $\bar{x}$. Note that for any function f which involves $\bar{x}$ we also have that f involves $\bar{y}$, for any $\bar{x} \subseteq \bar{y}$ (and a similar remark applies to predicates). Moreover we have that if f involves the variables $\bar{x}$ and $\bar{y}$ then f involves $\bar{x} \cap \bar{y}$ (and similarly for predicates). This we can prove as follows: Let σ and σ' be such that $\sigma(\bar{x} \cap \bar{y}) = \sigma'(\bar{x} \cap \bar{y})$. Let $\bar{z} = \bar{y} \setminus \bar{x}$ and $\sigma'' = (\sigma : \bar{z} \mapsto \sigma'(\bar{z}))$. So we have that $\sigma(\bar{x}) = \sigma''(\bar{x})$ and $\sigma''(\bar{y}) = \sigma'(\bar{y})$. Since f involve the variables $\bar{x}$ and $\bar{y}$ it then follows that $f(\sigma)(u) = f(\sigma'')(u) = f(\sigma')(u)$, for $u \in \bar{x} \cap \bar{y}$. In other words, $f(\sigma)(\bar{x} \cap \bar{y}) = f(\sigma')(\bar{x} \cap \bar{y})$. Next let $u \notin \bar{x} \cap \bar{y}$, that is, $u \notin \bar{x}$ or $z \notin \bar{y}$. Since f involves the variables $\bar{x}$ and $\bar{y}$ it thus follows that $f(\sigma)(u) = \sigma(u)$. A similar argument applies to predicates. Although this proves that sets of variables involved by f and φ are closed under finite intersection, this does not necessarily imply that they are closed under infinite intersection, as the following example shows.

Example 1. Let $Var = \{x_1, \ldots, x_n, \ldots\}$ and the predicate ϕ be defined as follows:

$$\sigma \in \phi \text{ if and only if } \exists i. \forall j, k \geq i. \sigma(x_j) = \sigma(x_k)$$

It follows that ϕ involves $Var \setminus \{x_1, \ldots, x_n\}$, for any n. But the infinite intersection of the sets $Var \setminus \{x_1, \ldots, x_n\}$ is empty and clearly it is not the case that ϕ involves the empty set. (A similar counter example applies to functions.)

Consequently we restrict ourselves to functions f and predicates ϕ for which there exists a *finite* set of variables which are involved by f and ϕ. Since any intersection with a finite set can be reduced to a finite intersection, the *smallest* set of variables involved by f (respectively ϕ) is well-defined. From now on we will call this smallest set the set of variables involved by f (respectively ϕ), also denoted by $VAR(f)$ and $VAR(\phi)$. We will use the phrase 'the variable x occurs in the state transformation f (predicate ϕ)' for $x \in VAR(f)$ ($x \in VAR(\phi)$). The definition of involvement of a variable in a predicate is extended in Def. 12 to involvement of a channel.

By $VAR(P)$, for P a synchronous transition diagram, we denote the variables occurring in its state transformations and boolean conditions. We call synchronous transition diagrams $P_1, \ldots, P_n$ *disjoint* if their associated sets of variables are mutually disjoint, and every channel occurring in $P_1, \ldots, P_n$ is unidirectional and connects at most two different diagrams. In the sequel we shall assume that only disjoint diagrams are composed in parallel. Formally, a parallel system P is inductively defined as a parallel composition $P_1 \parallel P_2$, where P_i is either a sequential process, i.e. a synchronous diagram, or again a parallel system. By $VAR(P)$, for P a parallel system, we denote the variables occurring in

the state transformations and boolean conditions of its sequential components. For parallel systems then we can also use the phrase 'the variable x occurs in P' for $x \in VAR(P)$. By $CHAN(P)$ we denote the set of channel names occurring in P.

3 Compositional semantics of synchronous transition diagrams

Given a synchronous transition diagram $P = (L, T, s, t)$ we define a *labeled* transition relation between *configurations* $\langle l; \sigma \rangle$, where $l \in L$. The labels are sequences of communications, denoted by θ. A communication itself is a pair (c, v), with c a channel name and $v \in VAL$. It indicates that the value v has been communicated along channel c. We assume the following operations on sequences: the append operation, denoted by $\cdot$, the operation of concatenation, denoted by $\circ$. The projection operation is denoted by $\downarrow$, i.e. $\theta \downarrow C$, with C a set of channels, denotes the subsequence of θ consisting of those communications invoving a channel $c \in C$. The empty sequence we denote by ϵ.

In the definition of the labeled transition relation we will make use of the notion of a *variant* of a state.

Definition 2. We define

$$\sigma\{v/x\}(y) = \begin{cases} \sigma(y) & \text{if } x \text{ and } y \text{ are distinct,} \\ v & \text{otherwise.} \end{cases}$$

Definition 3. Let $P = (L, T, s, t)$ be a synchronous transition diagram.

- In case of an internal transition $l \xrightarrow{a} l' \in T$, $a = b \to f$, we have
$$\langle l; \sigma \rangle \overset{\epsilon}{\Leftrightarrow} \langle l'; \sigma' \rangle,$$
if $\sigma \in b$ and where $\sigma' = f(\sigma)$.
- In case of an output transition $l \xrightarrow{a} l' \in T$, $a = b \to c!e$, we have, for $v = e(\sigma)$,
$$\langle l; \sigma \rangle \overset{(c,v)}{\Leftrightarrow} \langle l'; \sigma \rangle,$$
if $\sigma \in b$.
- In case of an input transition $l \xrightarrow{a} l' \in T$, with $a = b \to c?x$, we define, for an *arbitrary* value $v \in VAL$,
$$\langle l; \sigma \rangle \overset{(c,v)}{\Leftrightarrow} \langle l'; \sigma\{v/x\} \rangle,$$
if $\sigma \in b$.

Furthermore, we have the following rules for computing the reflexive, transitive closure:

$$\langle l; \sigma \rangle \overset{\epsilon}{\Leftrightarrow} \langle l; \sigma \rangle \text{ and } \frac{\langle l; \sigma \rangle \overset{\theta}{\Leftrightarrow} \langle l'; \sigma' \rangle \quad \langle l'; \sigma' \rangle \overset{\theta'}{\Leftrightarrow} \langle l''; \sigma'' \rangle}{\langle l; \sigma \rangle \overset{\theta \circ \theta'}{\Leftrightarrow} \langle l''; \sigma'' \rangle}$$

Using the above transition relation we can now define the semantics of a synchronous transition diagram, in which the value received in an input transition is selected by local guessing.

Definition 4. Let $P = (T, L, s, t)$ be a synchronous transition diagram and $l \in L$. We define

$$\mathcal{O}_l(P) = \{(\sigma, \sigma', \theta) \mid \langle s; \sigma\rangle \overset{\theta}{\Leftrightarrow} \langle l; \sigma'\rangle\}.$$

Note that we now can also define the initial/final state semantics of P as $\mathcal{O}_t(P)$, which we also denote by $\mathcal{O}(P)$.

In the following definition we extend the above semantics in a compositional manner to parallel systems.

Definition 5. For P a parallel system $P_1 \parallel P_2$, we define

$$\mathcal{O}(P) = \{(\sigma, \sigma', \theta) \mid \ (\sigma, \sigma'_i, \theta_i) \in \mathcal{O}(P_i),\ i = 1, 2\}.$$

Here σ'_i is obtained from σ' by assigning to the variables not belonging to process P_i their corresponding values in σ (note that P_i changes only its own local variables). The history θ_i denotes the projection $\theta \downarrow CHAN(P_i)$ of θ along the channels of P_i.

It is easy to see that the semantic operation of parallel composition defined above is commutative and associative.

We observe that in the above definition the requirement that the local histories θ_i can be obtained as the projection of one global history θ guarantees that an input on a channel indeed can be synchronized with a corresponding output. Also worthwhile to observe is the difference between the semantics of a parallel system and a sequential process: The histories θ in $\mathcal{O}(P)$, for P a synchronous transition diagram, contain only the communications of P itself whereas in $\mathcal{O}(P)$, with P a parallel system, they may also contain other communications.

Let us next explain the role of our basic assumption that channels are uni-directional and one-to-one. The above compositional semantics would generate, for example, for the network $c?x \parallel c?y \parallel c!0$ (abstracting from the locations) in which c connects two consumers with one producer, a global history which in fact models a multiparty communication interaction, i.e. the data produced is consumed at the same time by both consumers. However, our intention is to describe a communication mechanism such that any communication involves one sender and one receiver only (as it is in CSP, for example). Concerning the condition of uni-directionality of the channels, consider a network which connects two processes P_1 and P_2 via a *bi-directional* channel c such that both P_1 and P_2 first want to perform an input-statement on c and then an output-statement on c . Thus both processes P_1 and P_2 act as producer and consumer with respect to c. It is easy to see that $\mathcal{O}(P_1 \parallel P_2)$ is non-empty, whereas operationally this diagram clearly deadlocks. This is due to the fact that the communication history does not indicate the direction of the communications, and consequently does not capture the different rôles of the i/o statements $c?x$ and $c!e$.

We conclude this section with the following simple closure property of $\mathcal{O}$.

Lemma 6. *Let P be a parallel system and $\theta \downarrow CHAN(P) = \theta' \downarrow CHAN(P)$. It follows that for every pair of states σ and σ',*

$$(\sigma, \sigma', \theta) \in \mathcal{O}(P) \text{ iff } (\sigma, \sigma', \theta') \in \mathcal{O}(P).$$

4 A compositional proof method

In this section we first introduce compositionally inductive assertion networks for reasoning about the sequential components of a parallel system. Then we introduce compositional proof rules for deducing properties of the whole system. The basic idea of a compositionally inductive assertion network is the definition of the verification conditions in terms of a *logical* history variable which records the sequence of communications generated by the component. The parallel combination of these compositionally inductive assertion networks is defined in terms of a simple semantic characterization of the variables and channels involved in a predicate. The notion of the channels involved in a predicate is defined in terms of a natural generalization of the projection operation on histories to predicates.

We assume given a set of *history variables* $HVAR \subseteq VAR$ and a distinguished history variable $h \in HVAR$. A state $\sigma \in \Sigma$ thus assigns to each history variable a sequence of communications (and to each other variable an element of the given domain VAL). The distinguished history variable h represents the sequence of communications of the given parallel system. For every synchronous transition diagram P we require that every state transformation f and boolean condition b of P satisfies that $VAR(f) \subseteq (VAR \setminus HVAR)$ and $VAR(b) \subseteq (VAR \setminus HVAR$. This requirement then formalizes the condition that history variables do not occur in any program.

In order to reason about an input statement $c?x$ which involves the assignment of an *arbitrary* value to x, we need the introduction of quantifiers involving variables of the set $VAR \setminus HVAR)$. We define for a predicate ϕ, $\sigma \in \exists x.\phi$ iff there exists $v \in VAL$ such that $\sigma\{v/x\} \models \phi$.

Definition 7. An *assertion network* Φ for a synchronous transition diagram $P = (L, T, s, t)$ assigns to each $l \in L$ a predicate Φ_l.

We have the following definition of a *compositionally inductive* assertion network. In this definition $f(\phi)$, with ϕ a predicate, i.e. a set of states, and f a state-transformation, denotes the set of states $\{f(\sigma) \mid \sigma \in \phi\}$, i.e., the image of ϕ under f.

Definition 8. A local assertion network Φ for a synchronous transition diagram P is called *compositionally* inductive if:

- For $l \xrightarrow{a} l'$ a local transition of P, i.e., $a = b \rightarrow f$ for some boolean b and state-transformation f one has

$$f(\Phi_l \cap b) \subseteq \Phi_{l'}.$$

- For $l \xrightarrow{a} l'$ an output transition of P, i.e. $a = b \to c!e$, for some boolean b, channel c and state-transformation f, one has

$$g(\Phi_l \cap b) \subseteq \Phi_{l'},$$

  where $g(\sigma) = \sigma\{\sigma(h) \cdot (c, \sigma(e))/h\}$.
- For $l \xrightarrow{a} l'$ an input transition of P, i.e. $a = b \to c?x$, for some boolean b, channel c and state-transformation f, one has

$$g(\exists x.\Phi_l \cap b) \subseteq \Phi_{l'},$$

  where $g(\sigma) = \sigma\{\sigma(h) \cdot (c, \sigma(x))/h\}$.

We denote by $P \vdash \Phi$ that Φ is a compositionally inductive assertion network for P.

Definition 9. A *partial correctness statement* is of the form $\{\phi\}P\{\psi\}$, where ϕ and ψ are predicates, also called the precondition and postcondition, and P is either a synchronous transition diagram or a parallel system.

Formally, validity of a partial correctness statement $\{\phi\}P\{\psi\}$, with P either a synchronous transition diagram or a parallel system, notation: $\models \{\phi\}P\{\psi\}$, is defined with respect to the semantics $\mathcal{O}$.

Definition 10. We define $\models \{\phi\}P\{\psi\}$, with $P = (L, T, s, t)$ a synchronous transition diagram, by

for every $(\sigma, \sigma', \theta) \in \mathcal{O}(P)$ such that $\sigma\{\epsilon/h\} \in \phi$, we have $\sigma'\{\theta/h\} \in \phi$.

(Note that $\mathcal{O}(P) = \mathcal{O}_t(P)$.) Similarly, we define $\models \{\phi\}P\{\psi\}$, for P a parallel system, in terms of $\mathcal{O}(P)$.

Note that thus the validity of a partial correctness statement $\{\phi\}P\{\psi\}$ is defined with respect to computations of P which start in a state in which the history variable h is initialized to the empty sequence. We can impose this restriction because we do not consider the sequential composition of parallel systems (that is, we consider only top-level parallelism). For a discussion on the consequences of such an operation we refer to the section on nested parallelism.

Definition 11. For $P = (L, T, s, t)$ a synchronous transition diagram, we have the following main rule:

$$\frac{P \vdash \Phi}{\{\Phi_s\}P\{\Phi_t\}}$$

Moreover, for synchronous transition diagrams we have the following initialization rule.

$$\frac{\{\phi \cap h = \epsilon\}P\{\psi\}}{\{\phi\}P\{\psi\}}$$

where $h = \epsilon$ denotes the set $\{\sigma \mid \sigma(h) = \epsilon\}$.

In order to define a rule for parallel composition we introduce the following restriction operation on predicates.

Definition 12. Let ϕ be a predicate and C a set of channels. We denote by $\phi \downarrow C$ the predicate

$$\{\sigma \mid \text{there exists } \sigma' \in \phi \text{ s.t. } \sigma(x) = \sigma'(x), \text{ for } x \in VAR \setminus \{h\}, \\ \text{and } \sigma(h) \downarrow C = \sigma'(h) \downarrow C\}.$$

Note that $\phi \downarrow C = \phi$ indicates that, as far as the dependency of the value of ϕ upon the value of h is concerned, the value of ϕ only depends on the *projection* of the global history h on the channels C. More formally, $\phi \downarrow C = \phi$ indicates that for every σ and σ' such that σ and σ' are the same but for the value of the history variable h, and $\sigma(h) \downarrow C = \sigma'(h) \downarrow C$, we have

$$\sigma \models \phi \text{ if and only if } \sigma' \models \phi.$$

If $\phi \downarrow C = \phi$ then we also say that 'ϕ only involves the channels of C'.

We can now formulate the following rule for parallel composition.

Definition 13. Let $P = P_1 \parallel P_2$ in the rule

$$\frac{\{\phi_1\}P_1\{\psi_1\},\ \phi_2\}P_2\{\psi_2\}}{\{\phi_1 \cap \phi_2\}P\{\psi_1 \cap \psi_2\}}$$

provided ψ_i does not involve the variables of P_j and $\psi_i \downarrow CHAN(P_i) = \psi_i$, $i \neq j$.

Note that the restriction on channels indeed is necessary: Consider for example a network $c!0 \parallel d!0$ (abstracting from the locations of the components). Locally, we can prove

$$\{h = \epsilon\}c!0\{h = \langle(c,0)\rangle\} \text{ and } \{h = \epsilon\}d!0\{h = \langle(d,0)\rangle\}$$

(here $h = \epsilon$, for example, denotes the predicate $\{\sigma \mid \sigma(h) = \epsilon\}$). Applying the above rule leads to

$$\{h = \epsilon\}c!0 \parallel d!0\{false\},$$

However, this gives rise to incorrect results when further composing the system $c!0 \parallel d!0$. Observe that, e.g., postcondition $h = \langle(c,0)\rangle$ also involves channel d, in fact, we have that $h = \langle(c,0)$ involves *all* channels, and, hence, the condition upon the postconditions ψ_i in the above rule for parallel composition are violated.

We conclude the exposition of the proof system with the following consequence and elimination rules.

Definition 14. For P a synchronous transition diagram or a parallel system we have the usual consequence rule:

$$\frac{\phi \subseteq \phi',\ \{\phi'\}P\{\psi'\},\ \psi' \subseteq \psi}{\{\phi\}P\{\psi\}}$$

For P a parallel system we have the elimination rule:

$$\frac{\{\phi\}P\{\psi\}}{\{\exists \bar{z}.\phi\}P\{\psi\}}$$

where $\bar{z}$ is a sequence of variables which do not occur in P or ϕ (i.e. $\bar{z} \cap (VAR(P) \cup VAR(\phi)) = \emptyset$).

Derivability of a partial correctness statement $\{\phi\}P\{\psi\}$ using the rules above, with P a synchronous transition diagram or a parallel system, we denote by $\vdash \{\phi\}P\{\psi\}$.

5 Soundness

We have the following soundness result for synchronous transition diagrams.

Theorem 15. *Let $P = (L, T, s, t)$ be a synchronous transition diagram. We have that $P \vdash \Phi$ implies $\models \{\Phi_s\}P\{\Phi_t\}$.*

Proof
It suffices to prove that for every computation

$$\langle s; \sigma\rangle \overset{\theta}{\Leftrightarrow} \langle l; \sigma'\rangle$$

of $P = (L, T, s, t)$, we have that $\sigma\{\epsilon/h\} \in \Phi_s$ implies $\sigma'\{\theta/h\} \in \Phi_l$.

We proceed by induction on the length of the computation. We treat the case that the last transition involves the execution of a guarded input $b \to c?x$. Let

$$\langle s; \sigma\rangle \overset{\theta}{\Leftrightarrow} \langle l'; \sigma'\rangle \text{ and } \langle l'; \sigma'\rangle \overset{(c,v)}{\Leftrightarrow} \langle l; \sigma'\{v/x\}\rangle,$$

with $\sigma' \in b$ and $\sigma\{\epsilon/h\} \in \Phi_s$. By the induction hypothesis we have that

$$\sigma'\{\theta/h\} \in \Phi_{l'}.$$

We are given that Φ is compositionally inductive, so we have that

$$g(\exists x.\Phi_{l'} \cap b) \subseteq \Phi_l,$$

where $g(\sigma) = \sigma\{\sigma(h) \cdot (c, \sigma(x))/h\}$, for every state σ. Now $\sigma'\{\theta/h\} \in \Phi_{l'} \cap b$ (note that $h \notin VAR(b)$), so

$$\sigma'\{v/x\}\{\theta/h\} \in \exists x.\Phi_{l'} \cap b.$$

Consequently,

$$g(\sigma'\{v/x\}\{\theta/h\}) = \sigma'\{v/x\}\{\theta \cdot (c, v)/h\} \in g(\exists x.\Phi_{l'} \cap b) \subseteq \Phi_l.$$

□

Next we prove soundness of the parallel composition rule (the proof of the soundness of the other rules is standard).

Theorem 16. *Let $P = P_1 \parallel P_2$. We have that $\models \{\phi_1\}P_1\{\psi_1\}$ and $\models \phi_2\}P_2\{\psi_2\}$ implies $\models \{\phi_1 \cap \phi_2\}P\{\psi_1 \cap \psi_2\}$, provided ψ_i does not involve the variables of P_j and $\psi_i \downarrow C_i = \psi_i$, $i \neq j$.*

Proof
Let $(\sigma, \sigma', \theta) \in \mathcal{O}(P)$ such that $\sigma\{\epsilon/h\} \in \phi_1 \cap \phi_2$. By the definition of $\mathcal{O}(P)$ we have that

$$(\sigma, \sigma'_i, \theta_i) \in \mathcal{O}(P_i),\ i = 1, 2\},$$

where σ'_i is obtained from σ' by assigning to the variables not belonging to process P_i their corresponding values in σ, and θ_i denotes the projection $\theta \downarrow CHAN(P_i)$ of θ along the channels of P_i.

By the induction hypothesis we have that $\sigma'_i\{\theta_i/h\} \in \psi_i$, $i = 1, 2$. Since ψ_i does not involve the variables of P_j and $\psi_i \downarrow C_i = \psi_i$, we conclude that $\sigma'\{\theta/h\} \in \psi_i$, $i = 1, 2$. □

6 Semantic completeness

We want to prove completeness, that is, we want to prove that every valid partial correctness statement of a parallel system P is derivable. To this end we introduce the following *strongest postcondition semantics*.

Definition 17. Given a synchronous transition diagram $P = (L, T, s, t)$, $l \in L$, and a precondition ϕ we define

$$SP_l(\phi, P) = \{\sigma\{\theta/h\} \mid \text{there exists } \sigma' \text{ s.t. } \sigma'\{\epsilon/h\} \in \phi \text{ and } (\sigma', \sigma, \theta) \in \mathcal{O}_l(P)\}.$$

By $SP(\phi, P)$, with $P = (L, T, s, t)$, we denote $SP_t(\phi, P) \downarrow CHAN(P)$. Similarly we define $SP(\phi, P)$, for P a parallel sytem, in terms of $\mathcal{O}(P)$.

It is easy to see that $\models \{\phi\}P\{SP_t(\phi, P)\}$, and that $SP_t(\phi, P) \subseteq \psi$ is implied by the validity of $\{\phi\}P\{\psi\}$, for P a synchronous transition diagram. Similarly, for P a parallel system, we have that $\models \{\phi\}P\{SP(\phi, P)\}$ and that the validity of $\{\phi\}P\{\psi\}$ implies $SP(\phi, P) \subseteq \psi$. Moreover, we have that $SP(\phi, P)$ only involves the channels of P.

Lemma 18. *For P a synchronous transition diagram or a parallel system we have that*

$$SP(\phi, P) \downarrow CHAN(P) = SP(\phi, P).$$

Proof
For P a synchronous transition diagram we have by definition that $SP(\phi, P) = SP_t(\phi, P) \downarrow CHAN(P)$. For P be a parallel system the statement follows from the definition of $SP(\phi, P)$ and lemma 6. □

We have the following completeness result for synchronous transition diagrams.

Theorem 19. *Let P be a synchronous transition diagram. We have*

$$\vdash \{\phi\}P\{SP(\phi, P)\}.$$

Proof
Let $P = (L, T, s, t)$ and let Φ be the assertion network which associates with each location l of P the predicate $SP_l(\phi, P)$. It is easy to prove that this assertion network is compositionally inductive. We treat the case of an input transition $l \xrightarrow{a} l' \in T$, with $a = b \to c?x$: We have to prove that

$$g(\exists x.SP_l(\phi, P) \cap b) \subseteq SP_{l'}(\phi, P),$$

where $g(\sigma) = \sigma\{\sigma(h) \cdot (c, \sigma(x))/h\}$, for every state σ. To this end let

$$\sigma \in g(\exists x.SP_l(\phi, P) \cap b).$$

So we have that

$$\sigma = \sigma'\{v/x\}\{\sigma'(h) \cdot (c, v)/h\},$$

for some value v and state $\sigma' \in SP_l(\phi, P) \cap b$. By definition of $SP_l(\phi, P)$ we thus derive that

$$\langle s; \sigma\rangle \overset{\theta}{\Leftrightarrow} \langle l; \sigma'\rangle,$$

for $\theta = \sigma'(h)$. By definition 3, note that $\sigma' \in b$, it follows that

$$\langle l; \sigma'\rangle \overset{(c,v)}{\Leftrightarrow} \langle l'; \sigma'\{v/x\}\rangle.$$

We conclude that $\sigma \in SP_{l'}(\phi, P)$. Coming back to our main argument, we derive that

$$\vdash \{SP_s(\phi, P)\}P\{SP_t(\phi, P).\}$$

Next we observe that $(\phi \cap h = \epsilon) \subseteq SP_s(\phi, P)$ and that $SP_t(\phi, P) \subseteq SP_t(\phi, P)\downarrow CHAN(P) (= SP(\phi, P))$. Thus an application of the consequence rule gives us

$$\vdash \{\phi \cap h = \epsilon\}P\{SP(\phi, P)\}.$$

We conclude with an application of the initialization rule that

$$\vdash \{\phi\}P\{SP(\phi, P)\}.$$

□

Now we want to prove $\vdash \{\phi\}P\{SP(\phi, P)\}$, for $P = P_1 \parallel P_2$. We first introduce the following set of variables and channels.

- $\bar{x} = VAR(\phi, \psi, P)$,
- $\bar{x}_i = VAR(P_i)$, $i = 1, 2$.

By the induction hypothesis and the above theorem we have

$$\vdash \{\phi'\}P_i\{SP(\phi', P_i)\},\ i = 1, 2,$$

where $\phi' = (\phi \cap \bar{z} = \bar{x})$, with $\bar{z}$ a sequence of 'fresh' variables corresponding to $\bar{x}$ ($\bar{z} = \bar{x}$ denotes the set of states σ such that $\sigma(z_i) = \sigma(x_i)$, for every $z_i \in \bar{z}$ and corresponding x_i). The variables $\bar{z}$ are used to *freeze* the initial values of $\bar{x}$: the predicate $\bar{z} = \bar{x}$ initializes the variables of $\bar{z}$ to the values of the corresponding $\bar{x}$.

Applying the consequence rule we obtain

$$\vdash \{\phi'\}P_i\{\exists \bar{x}_j SP(\phi', P_i)\},\ i, j \in \{1, 2\},\ i \neq j.$$

In order to apply the parallel composition rule we first observe that $\exists \bar{x}_j SP(\phi', P_i)\}$ does not involve the variables of P_j. Moreover it is easy to prove that

$$(\exists \bar{x}_j SP(\phi', P_i)) \downarrow C_i = \exists \bar{x}_j (SP(\phi', P_i) \downarrow C_i).$$

By lemma 18 we thus derive that

$$(\exists \bar{x}_j SP(\phi', P_i)) \downarrow C_i = \exists \bar{x}_j SP(\phi', P_i).$$

So the conditions of the parallel composition rule are satisfied. We obtain

$$\vdash \{\phi'\}P\{\exists \bar{x}_2 SP(\phi', P_1) \wedge \exists \bar{x}_1 SP(\phi', P_2)\}.$$

To proceed we need the following main theorem.

Theorem 20. *We have* $(\exists \bar{x}_2 SP(\phi', P_1) \cap \exists \bar{x}_1 SP(\phi', P_2)) \subseteq SP(\phi', P)$.

Proof
Let $\sigma \in \exists \bar{x}_2 SP(\phi', P_1) \cap \exists \bar{x}_1 SP(\phi', P_2)$. So there exist states σ_1 and σ_2 such that σ differs from σ_i only with respect to the variables $\bar{x}_j$ $(i \neq j)$, and $\sigma_i \in SP(\phi', P_2)$, $i, j \in \{1, 2\}$. By definition of the strongest postcondition and lemma 6 there exist states σ_1' and σ_2' such that

- $\sigma_1'\{\epsilon/h\}, \sigma_2'\{\epsilon/h\} \in \phi'$, and
- $(\sigma_i', \sigma_i, \theta_i) \in \mathcal{O}(P_i)$, where $\sigma(h) = \theta$ and $\theta_i = \theta \downarrow CHAN(P_i)$ $(i = 1, 2)$.

Since $\sigma_i'(\bar{z}) = \sigma_i'(\bar{x})$ and $\sigma(\bar{z}) = \sigma_i(\bar{z}) = \sigma_i'(\bar{z})$, it follows that σ_1' and σ_2' agree with respect to the variables $\bar{x}$. Moreover, for any other variable $y \in VAR \setminus \{h\}$ we have that $\sigma(y) = \sigma_i(y) = \sigma_i'(y)$. Summarizing, we have established that $\sigma_1'\{\epsilon/h\} = \sigma_2'\{\epsilon/h\}$. Let us call this state σ''.

We have that $\sigma_i(\bar{x}_j) = \sigma''(\bar{x}_j)$ $(i \neq j)$. In other words, σ_i can be obtained from σ by assigning to the variables not belonging to process P_i their corresponding values in σ''. So by definition of $\mathcal{O}(P)$ we derive that $(\sigma''\{\theta/h\}, \sigma, \theta) \in \mathcal{O}(P)$, from which we obtain that $\sigma \in SP(\phi', P)$. □

Applying the consequence rule to $\{\phi'\}P\{\exists \bar{x}_2 SP(\phi', P_1) \wedge \exists \bar{x}_1 SP(\phi', P_2)\}$ we thus obtain

$$\vdash \{\phi'\}P\{SP(\phi', P)\}\}.$$

Since $SP(\phi', P) \subseteq SP(\phi, P)$ ($\phi' \subseteq \phi$ and SP is monotonic in its first argument), we derive by another application of the consequence rule

$$\{\phi'\}P\{SP(\phi, P)\}.$$

An application of the elimination rule then gives us

$$\{\exists \bar{z}.\phi'\}P\{SP(\phi, P)\}.$$

Finally, we observe that $\phi = \exists \bar{z}.\phi'$, from which the result follows.

7 Nested parallelism

We have illustrated in this paper our general semantic approach to compositional proof methods for concurrent systems by means of a compositional proof method for parallel systems based on synchronous message passing. We refer to [4] for a detailed exposition of various generalizations. Here we only sketch the generalization to *nested parallelism*.

Nested parallelism arises when we introduce the operation of *sequential composition* of synchronous transition diagrams. As has been pointed out in [5] this operation requires the introduction in the proof method of the *prefix-invariance* axiom. The basic ideas of the approach described in this paper have also been applied smoothly in [4] to the semantic explanation of this "enigmatic" prefix-invariance axiom. Semantically, the prefix-invariance axiom derives from the following generalization of the validity of a partial correctness statement.

Definition 21. We define $\models \{\phi\}P\{\psi\}$ as follows:

for every $(\sigma, \sigma', \theta) \in \mathcal{O}(P)$ such that $\sigma \in \phi$, we have $\sigma'\{\sigma(h) \circ \theta/h\} \in \psi$.

The main difference with the notion of validity given in definition 10 is that we do not require the initial history to be empty. This is clearly necessary when we want to reason about sequential composition. However, the mere fact that thus we need to reason about a possibly non-empty initial history requires already the introduction of the prefix-invariance axiom, even in the context of top-level parallelism! This follows from the standard example. We cannot derive the valid correctness statement

$$\{\langle d, c\rangle = h\}d! \parallel c!\{\langle d, c\rangle \leq h\}$$

(we abstract both from locations and the values sent) which tells us that if the past history consists of first a communication on d followed by one on c, then *after* the communications $d!$ and $c!$ one has that $\langle d, c\rangle$ is a *prefix* (the prefix relation on sequences is denoted by $\leq$) of the new history h. It is not difficult to see that we cannot derive this correctness statement because of the restrictions on the postconditions in the rule for parallel composition, namely that they should involve only the channels of the components they describe.

In order to derive this correctness statement we have to introduce the following prefix-invariance axiom:

$$\{t = h\}P\{t \leq h\}.$$

We can now prove, along the lines of the above completeness proof, that any correctness statement $\models \{\phi\}P\{\psi\}$ about a top-level parallel system P, which is valid under the above new definition, is derivable (see [4]).

References

1. S. Abramsky. Domain Theory in Logical Form. *Proceedings, Annual Symposium on Logic in Computer Science, pp. 47-53, IEEE CS*, 1987. Extended version in Annals of Pure and Applied Logic, 51: 1-77, 1991.
2. R.W. Floyd. Assigning meanings to programs. In *Proceedings AMS Symp. Applied Mathematics*, volume 19, pages 19–31, Providence, R.I., 1967. American Mathematical Society.
3. R.P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. Proceedings of the 5th International Conference on Computer-Aided Verification (CAV94).
4. W.-P. de Roever, F.S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. Concurrency Verification: From Noncompositional to Compositional Proof Methods. Submitted for publication in 1998.
5. J. Zwiers. *Compositionality and Partial Correctness.* LNCS 321. Springer-Verlag, 1989.

Author Index

Alur, R. 23
Aubry, P. 61
Benveniste, A. 61
Berezin, S. 81
Boer de, F. 632
Bornot, S. 103
Broy, M. 130
Campos, S. 81
Clarke, E.M. 81
Dam, M. 150
Damm, W. 186
Dierks, H. 465
Finkbeiner, B. 239
Fredlund, L. 150
Gurov, D. 150
Hansen, M.R. 584
Henzinger, T.A. 23
Holenderski, L. 490
Hooman, J. 276
Hungar, H. 186
James, J. 301
Janssen, T. M.V. 327
Jonker, C.M. 350
Josko, B. 186
Kupferman, O. 23
Kupferman, O. 381
Lamport, L. 402
Le Guernic, P. 61
Manna, Z. 239
Maraninchi, F. 424
Moszkowski, B.C. 439
Olderog, E.-R. 465
Pnueli, A. 186
Poigné, A. 490
Rémond, Y. 424
Roever de, W.-P. 1, 632
Segala, R. 515
Shankar, N. 541
Sifakis, J. 103
Singh, A. 301
Sipma, H.B. 239
Swarup, M. 565
Treur, J. 350
Tripakis, S. 103
Vardi, M.Y. 381
Xu, Q. 565
Zhou, C. 584
Zwiers, J. 609