

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Teodor Rus (Ed.)

Algebraic Methodology and Software Technology

8th International Conference, AMAST 2000
Iowa City, Iowa, USA, May 20-27, 2000
Proceedings



Springer

Series Editors

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

Volume Editor

Teodor Rus
The University of Iowa, Department of Computer Science
Iowa City, IA 52242, USA
E-mail: rus@cs.uiowa.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algebraic methodology and software technology : 8th international
conference ; proceedings / AMAST 2000, Iowa City, Iowa, USA, May 20 -
27, 2000. Teodor Rus (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 2000
(Lecture notes in computer science ; Vol. 1816)
ISBN 3-540-67530-2

CR Subject Classification (1998): F.3-4, D.2, C.3, D.1.6, I.2.3, I.1.3

ISSN 0302-9743

ISBN 3-540-67530-2 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 is a company in the BertelsmannSpringer publishing group.
© Springer-Verlag Berlin Heidelberg 2000
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10720319 06/3142 5 4 3 2 1 0

Preface

The AMAST movement was initiated in 1989 with the First International Conference on Algebraic Methodology and Software Technology (AMAST), held on May 21–23 in Iowa City, Iowa, and aimed at setting the development of software technology on a mathematical basis. The virtue of the software technology envisioned by AMAST is the capability to produce software that has the following properties: (a) it is correct and its correctness can be proved mathematically, (b) it is safe, such that it can be used in the implementation of critical systems, (c) it is portable, i.e., it is independent of computing platforms and language generations, and (d) it is evolutionary, i.e., it is self-adaptable and evolves with the problem domain. Ten years later a myriad of workshops, conferences, and research programs that share the goals of the AMAST movement have occurred. This can be taken as proof that the AMAST vision is right. However, often the myriad of workshops, conferences, and research programs lack the clear objectives and the coordination of their goals towards the software technology envisioned by AMAST. This can be taken as a proof that AMAST is still necessary.

The approach of software development promoted by AMAST is based on the integration of the two fundamental mechanisms for problem solving, *specification* and *computation*, into a general problem-solving model by a machine-independent methodology where: (1) specification allows one to develop the mathematical model of the problem at hand and to represent this model in a mathematical language; (2) computation allows one to apply specific *calculi* on problem representation and its input data to deduce the solution. A major aspect of the AMAST interpretation of the algebraic methodology is that software environments designed to help develop software systems should be populated by tools, components, and patterns. Tools are efficient implementations of universal algorithms; components are stand-alone software automatically generated by tools from finite specifications; patterns are domain-oriented composition schemes mapping components into software systems. This ensures the development of a software factory capable of producing software systems having the qualities required by the AMAST movement.

Ten years after its initiation AMAST is no longer a simple grass-roots movement. Instead, AMAST has become a professional movement whose goals are embraced by a large international following. As long as AMAST remains a movement of world-wide computer science researchers determined by philosophical and intellectual conviction rather than by academic or business opportunism, AMAST can provide the balance and the common sense needed by the myriad of formal-methods trends, that are often short sighted by short-term goals of for-profit-organizations. This makes AMAST further necessary as the unique worldwide movement with the goal of *creating the mathematical methodology for the development of software technology*. The transformation of algebraic methodology into the software technology is a dynamic process that involves both static

and evolutionary knowledge. Since the more one discovers the more remains to be discovered this process never ends. Hence, AMAST will always be needed to balance the process of building the software factory.

In response to the call for papers, 53 papers were submitted to AMAST 2000 and 29 have been selected by the program committee. The relatively small number of papers submitted tells us that the recent advances in software system design and application requires new forms of research dissemination. It seems that the classical form of paper presentation cannot catch the excitement raised by the current stage of the software technology. Therefore AMAST 2000 takes a step ahead on research dissemination by mixing paper presentations with open panel discussions. The discussion of the software worker education is guided by David Lorge Parnas, who introduces us to “A software engineering program of lasting value” and Jeannette M. Wing, who presents “Thoughts on integrating formal methods into a computer science curriculum”. The technical meetings are organized around the following invited talks: Egidio Astesiano, “Plugging data constructs into paradigm-specific languages: towards an application to UML”; Yuri Gurevich, “ASM formalware in the software engineering cycle”; Michael Healy, “Applying category theory to derive engineering software from encoded knowledge”; Oege de Moor, “Pointwise relational programming”; David Lorge Parnas, “Making mathematical methods more practical for the software developer”; and Martin Wirsings, “Algebraic state machines”. The excitement is however provided by the three open-panel discussions organized by Joseph Goguen, “New computing logics”; Douglas Smith, “Automatic program generation”; and Roger Shultz, “From software model to commercial product”. We thank all these invited speakers for sharing their expertise with us.

Here I thank those whose generosity made possible the organization of this AMAST conference. Ralph Wachter, Office of Naval Research, helped us to initiate AMAST, to lead it towards its maturity, and finally to celebrate its 10th anniversary. Thanks are due to the University of Iowa, Dean Linda Maxson, Associate Provost Lee Anna Clark, and Vice President for Research David Skorton, for their financial support, and particularly Steve Bruell, Chair of Computer Science Department, for his continual support. I also thank Arthur Fleck and the AMAST steering committee for the guidance and all the program committee members for their effort during paper reviewing and selection. Finally, I thank all those who submitted papers to this edition of AMAST. I believe that the result of a peer-evaluation process is rather subjective and therefore I congratulate both those whose papers were accepted as well those whose papers were not accepted. Last (of course, not least) my thanks go to the organizing committee, particularly to Robert Kooima, and to Springer-Verlag, particularly to Alfred Hofmann, Anna Kramer, and Antje Endemann, whose excellent cooperation and advice made these proceedings feasible.

Organization

Conference chair: Maurice Nivat

Program chair: Teodor Rus

Local organization chair: Steve Bruell

Program Committee

Andre Arnold, France	Egidio Astesiano, Italy
Gabriel Baum, Argentina	Didier Begay, France
Robert Berwick, USA	Michel Bidoit, France
Val Tannen, USA	Gregor Bochmann, Canada
Chris Brink, South Africa	Manfred Broy, Germany
Christian Calude, New Zealand	Christine Choppy, France
Philippe Darondeau, France	Jim Davies, UK
Rocco De Nicola, Italy	Ruy de Queiroz, Brazil
Arthur Fleck, USA	Marcelo Frias, Argentina
Kokichi Futatsugi, Japan	Dov Gabbay, UK
Harald Ganzinger, Germany	Radu Grosu, USA
Yuri Gurevich, USA	Armando Haeberer, Brazil
Nicolas Halbwachs, France	Michael Healy, USA
Peter Henderson, UK	Yoshi Inagaki, Japan
Paola Inverardi, Italy	Dan Ionescu, Canada
Ryszard Janicki, Canada	Jarkko Kari, USA
Michael Johnson, Australia	Helene Kirchner, France
Gary Leavens, USA	Luigi Logrippo, Canada
Thomas Maibaum, UK	Zohar Manna, USA
Chris Marlin, Australia	Michael Mislove, USA
Peter Mosses, Denmark	George Nelson, USA
Anton Nijholt, The Netherlands	Maurice Nivat, France
Michael O'Donnell, USA	Fernando Orejas, Spain
David Lorge Parnas, Canada	Sriram Pemmaraju, India
Don Pigozzi, USA	Jacques Printz, France
Charles Rattray, UK	Teodor Rus, USA
Giuseppe Scollo, The Netherlands	Stephen Seidman, USA
Roger Shultz, USA	Ken Slonneger, USA
Douglas Smith, USA	John Staples, Australia
Carolyn Talcott, USA	Andrzej Tarlecki, Poland
Alagar Vangalur, Canada	Rob van Glabbeek, USA
Paulo Veloso, Brazil	Brian Warboys, UK
Jeannette Wing, USA	Martin Wirsing, Germany
Hantao Zhang, USA	

Steering Committee

E. Astesiano	R. Berwick	Z. Manna
M. Mislove	A. Nijholt	M. Nivat
J. Printzs	C. Rattray	T. Rus
G. Scollo	J. Staples	J. Wing
M. Wirsing		

Organizing Committee

A. Fleck	R. Kooima	T. Rus

Referees

V.S. Alagar	A. Arnold	G. Baum
M. Bidoit	M. Boreale	M. Breitling
C. Calude	A. Cerone	P. R. D'Argenio
P. Darondeau	J. Davies	D. Eichmann
C. Fidge	A. Fleck	M. Frias
H. Ganzinger	R. Grosu	N. Halbwachs
M. Healy	P. Henderson	P. Inverardi
D. Ionescu	R. Janicki	M. Johnson
J. Kari	C. Kirchner	M. Koutny
I. Kreuger	G. Leavens	L. Logrippo
M. Loreti	S. Maharaj	F. Maraninchi
C. Marlin	P.E. Martinez Lopez	M. Mislove
L. Moss	P. Mosses	G. Nelson
A. Nijholt	M. O'Donnell	F. Orejas
D. Pigozzi	R. Pugliese	C. Rattray
G. Rosolini	T. Rus	P. Schnoebelen
G. Scollo	S. Seidman	H. Sipma
K. Slonneger	G. Smith	C. Talcott
A. Tarlecki	R. van Glabbeek	B. Warboys
M. Wirsing	H. Zhang	

Sponsoring Institutions

The University of Iowa
Office of Naval Research

Contents

Education Day

Invited Talk: A Software Engineering Program of Lasting Value	1
<i>David L. Parnas</i>	
Invited Talk: Weaving Formal Methods into the Undergraduate Computer Science Curriculum	2
<i>Jeannette M. Wing</i>	

Technical Meetings

Session 1

Invited Talk: Making Mathematical Methods More Practical for the Software Developers	9
<i>David L. Parnas</i>	
Step by Step to Histories	11
<i>Max Breitling, Jan Philipps</i>	
Distance Functions for Defaults in Reactive Systems	26
<i>Sofia Guerra</i>	
Generalizing the Modal and Temporal Logic of Linear Time	41
<i>Bernhard Heinemann</i>	
Process Algebra versus Axiomatic Specification of a Real-Time Protocol . .	57
<i>Antonio Cerone</i>	
Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software	73
<i>Mark Lawford, Jeff McDougall, Peter Froebel, Greg Moun</i>	

Session 2

Invited Talk: Algebraic State Machines	89
<i>Manfred Broy, Martin Wirsing</i>	
Meta Languages in Algebraic Compilers	119
<i>Eric Van Wyk</i>	
Random Access to Abstract Data Types	135
<i>Martin Erwig</i>	

A Monad for Basic Java Semantics	150
<i>Bart Jacobs, Erik Poll</i>	
A Global Semantics for Views	165
<i>Christine Choppy, Pascal Poizat, Jean-Claude Royer</i>	
Analysis of Downward Closed Properties of Logic Programs	181
<i>Patricia M. Hill, Fausto Spoto</i>	

Session 3

Invited Talk: ASM Formalware in the Software Engineering Cycle	197
<i>Yuri Gurevich</i>	
Process Calculi for Coordination: From Linda to JavaSpaces	198
<i>Nadia Busi, Roberto Gorrieri, Gianluigi Zavattaro</i>	
The Algebra of Multi-tasking	213
<i>Colin J. Fidge</i>	
A Causal Semantics for Timed Default Concurrent Constraint Programming	228
<i>Simone Tini, Andrea Maggiolo-Schettini</i>	
CASL-CHART: A Combination of Statecharts and of the Algebraic Specification Language CASL	243
<i>Gianna Reggio, Lorenzo Repetto</i>	
Message Authentication through <i>Non Interference</i>	258
<i>Riccardo Focardi, Roberto Gorrieri, Fabio Martinelli</i>	

Session 4

Invited Talk: Plugging Data Constructs into Paradigm-Specific Languages: Towards an Application to UML	273
<i>Egidio Astesiano, Maura Cerioli, Gianna Reggio</i>	
An ASM Semantics for UML Activity Diagrams	293
<i>Egon Börger, Alessandra Cavarra, Elvinia Riccobene</i>	
Approximate Bisimilarity	309
<i>Mingsheng Ying, Martin Wirsing</i>	
Time and Probability in Process Algebra	323
<i>Suzana Andova</i>	
A Modal Logic for KLAIM	339
<i>Rocco De Nicola, Michele Loreti</i>	

Kleene under a Demonic Star	355
<i>Jules Desharnais, Bernhard Möller, Fairouz Tchier</i>	

Session 5

Invited Talk: Pointwise Relational Programming	371
<i>Oege de Moor, Jeremy Gibbons</i>	
Towards a Toolkit for Actor System Specification	391
<i>Carolyn L. Talcott</i>	
Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic	407
<i>Christiano de O. Braga, E. Hermann Haeusler, José Meseguer, Peter D. Mosses</i>	
The Extensibility of Maude's Module Algebra	422
<i>Francisco Durán</i>	
A Reuse-Oriented Framework for Hierarchical Specifications	438
<i>Sophie Coudert, Pascale Le Gall</i>	
MIX(FL): A Kernel Language of Mixin Modules	454
<i>Davide Ancona</i>	
Behavioural Subtyping Relations for Object-Oriented Formalisms	469
<i>Clemens Fischer, Heike Wehrheim</i>	

Session 6

Invited Talk: Applying Category Theory to Derive Engineering Software from Encoded Knowledge	484
<i>Michael Healy, Keith Williamson</i>	
A New Logic for Electronic Commerce Protocols	499
<i>Kamel Adi, Mourad Debbabi, Mohamed Mejri</i>	
Extended Institutions for Testing	514
<i>Marielle Doche, Virginie Wiels</i>	
Testing from Structured Algebraic Specifications	529
<i>Patrícia D. L. Machado</i>	
Author Index	545

Invited Talk: A Software Engineering Program of Lasting Value (Abstract)

David L. Parnas, P.Eng

NSERC/Bell Industrial Research Chair in Software Engineering
Director of the Software Engineering Programme
Department of Computing and Software
Faculty of Engineering, McMaster University
Hamilton, Ontario Canada L8S 4L7

Engineering educators have long recognised that it is their obligation to prepare students for a professional career that may last 40 years in rapidly changing fields. Good engineering educators know that they must focus on fundamental ideas and teach students how to apply those ideas. Thus, although I studied Electrical Engineering at a time when semiconductor research was considered useless theory by many of my teachers, most of my textbooks are still valid and still useful. In contrast, most of the computer books on my shelves are out-of-date and irrelevant.

It is clear that a Software Engineering program cannot attempt to keep up with the latest. If we teach today's newest ideas, much of what we teach will be considered out of date before the students graduate. On the other hand, students who end up working in environments that use older tools will not be prepared for that job.

It should also be clear that we must teach students things that they will be able to apply and show them how to apply what they have learned. Otherwise they will ignore their education when they leave the University.

In designing McMaster University's Software Engineering program, we concluded we should not teach anything that would not have been useful (if known) 20 years ago and will not be useful 40 years from now. We also took the attitude that we should not teach theory unless we could show how to use it and we should not teach any pragmatic techniques unless we could show that "theory" proved that the technique was sound.

The result of applying these principles is a program that is very different from most Computer Science programs. While students use the latest technology in the laboratories, the lectures, homework, and examinations focus on more fundamental material. Classical mathematics, material that is certainly of lasting value, is at the heart of the program. The other courses build on that mathematical foundation. There are many practical courses in which student teams build well-structured and well-documented software but mathematics and science are applied in each one.

The talk will provide a description of our new program including some details about the key courses.

Invited Talk: Weaving Formal Methods into the Undergraduate Computer Science Curriculum (Extended Abstract)

Jeannette M. Wing

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA
wing@cs.cmu.edu
<http://www.cs.cmu.edu/~wing/>

Abstract. We can integrate formal methods into an existing undergraduate curriculum by focusing on teaching their common conceptual elements and by using state of the art formal methods tools. Common elements include state machines, invariants, abstraction mappings, composition, induction, specification, and verification. Tools include model checkers and specification checkers. By introducing and regularly revisiting the concepts throughout the entire curriculum and by using the tools for homework assignments and class projects, we may be able to attain the ideal goal of having computer scientists use formal methods without their even realizing it.

1 Philosophy

Rather than treat formal methods solely as a separate subject to study, we should weave their use into the existing infrastructure of an undergraduate computer science curriculum. In so doing, we would be teaching formal methods alongside other mathematical, scientific, and engineering methods already taught. Formal methods would simply be additional weapons in a computer scientist's arsenal of ways to think when attacking and solving problems.

My ideal is to get to the point where computer scientists use formal methods *without* even thinking about it. Just as we use simple mathematics in our daily life, computer scientists would use formal methods routinely.

By formal methods I mean the specification and verification of hardware and software systems. Some methods will be accessible to undergraduates—these are the ones I hope computer scientists will use without realizing it. Some methods are more advanced, requiring either more mathematical sophistication or domain knowledge—those can be taught in upper-level electives, in graduate courses, or through independent undergraduate research projects. In this paper I focus on the former.

Dating back to the Dijkstra-Gries predicate transformer approach of program development [Gr81], we already have a long history of inculcating undergraduates

with notions of program specification and verification. While there are varying degrees of success in teaching programming using this approach, the method is not used by programmers in practice. Moreover, specifying and verifying small, simple programs does not address the problems of scale and complexity faced by software engineers in industry. What should we do differently or why should we be more optimistic now?

First, we should focus on teaching the common elements of all (or most) methods, rather than on the specific notation or stylistic requirements of the method itself. Students writing large programs are not easily going to be able to do a stepwise refinement of design to code following the Dijkstra-Gries approach, but they can certainly learn and apply the notions of program specification, loop invariants, and termination functions. Programming-in-the-small and programming-in-the-large are inherently creative problem solving activities. Thinking in terms of formal methods concepts, e.g., invariants, forces the designer to take a more abstract perspective of a system than that taken with an algorithmic or operational approach. This more abstract thinking invariably provides the designer with new insights and a deeper understanding of the system's desired behavior.

Second, tools are essential. Without sufficient tool support, a method will not scale to practice. Model checking [CGP99] is a successful formal method because it addresses scale in two ways: it is applicable to a narrow problem domain (control aspects of hardware and protocols) and we do not have to specify the whole system before we can do some interesting verification. Furthermore, without appropriate tool support, typical computer science students have no incentive to use them. While mathematics students may be happy to do pencil and paper proofs, computer science students grow up using compilers, interpreters, operating systems, databases, graphical user interfaces, editors, electronic mail systems, spreadsheets, document preparation packages, web browsers, search engines, and so on. Formal methods tools have to be packaged in a way to fit into the way computer scientists work on a daily basis.

What follows is first, a list of the common elements and tools which we can teach to undergraduates, and second, specific suggestions on where to teach them with respect to existing courses found in a typical undergraduate computer science curriculum.

2 What We Can Teach

2.1 Common Elements

Below is a list of the elements that transcend the specific syntax and semantics of most formal methods. A firm understanding of these concepts goes a long way.

- *State machines.* The notion of a *state machine* as a tuple of a set of states, a set of initial states, and a transition relation between states; variations include accepting states, nondeterministic transition relations, finite state machines, labelled states, and labelled transitions. The notion of a *state* as a

mapping from variables to values; enrichments include using typed variables and values, and modeling both the environment and store as needed for imperative programming languages. The notion of *executions* as sequences of interleaved states and transitions; various projections on executions, for example, on just states, transitions, objects, or processes (these projections are often termed *traces* or *histories*). The notion of *behavior* and *observable behavior* of a state machine as a set of executions (or traces or histories).

- *Invariants*. The notion of state invariants; variations include abstract and representation type invariants in the context of abstract data types. The notion of loop invariants for statement-level reasoning.
- *Abstraction mappings*. The notion of abstraction functions for reasoning about abstract data types. The more general notion of simulation relations, for example, in relating states (or transitions or executions) of one machine to those of another.
- *Composition*. The notion of composition as a way to build larger machines (systems). Basic functional composition as in sequential composition of statements and nested and recursive procedure calls. The use of interfaces to compose modules, as already manifest in programming languages like Java and ML. Process composition for concurrent and distributed systems. Problems due to interference when composing concurrent processes.
- *Induction*. The basic notions of mathematical and complete (strong) induction. Structural induction for reasoning abstract data types. Computational induction for reasoning more generally about state machines.
- *Specification*. The notion of writing a formal description of *what* the system is supposed to do, not *how*; i.e., the difference between a specification and code. The notion of a type as a “weak” (in terms of expressibility), but extremely powerful (in terms of practicality) specification. Going further, pre-conditions and post-conditions and other predicates (e.g., Larch’s **modifies** clause [GH93]). Going even further, the use of rely and guarantee predicates for reasoning about concurrent programs.
- *Verification*. The notions of correctness and termination of a program, and more generally, notions of safety and liveness properties of concurrent and distributed systems. Proof techniques for showing a system satisfies its specification. Termination functions and well-founded orderings for proofs of termination.

There are clearly mathematical prerequisites or corequisites for understanding these concepts. They are (1) discrete mathematics, minimally, algebraic structures and their properties; and (2) mathematical logic, minimally, first-order predicate logic, and proof techniques.

2.2 Tools

There are two classes of tools that we can use at the undergraduate level: model checkers and specification checkers.

There is no excuse not to be using *model checkers* in our undergraduate courses today. With a verification tool, we can more easily teach that verification complements the testing and simulation activities of practicing hardware and software engineers. Model checkers verify temporal properties of finite state machines. They are fast, completely automatic, and relatively easy to learn. There are industrial-strength, commercial model checkers available on the market. If the trend of using them in the hardware industry continues, then it behooves us as educators to ensure that our students are well-versed in the state of the art verification technology.

Specification checkers are less common and are still making their transition from research environments to industry. One promising kind of specification checker is exemplified by LCLint [EGHT94] and ESC/Java [CSRC00], which both support incremental specifications: as we add more to a specification, the tool can check more of the code. LCLint does much of the traditional lint checks of C programs, including unused declarations, type inconsistencies, and use-before-definition; additional source code annotations, in the form of pre-/post-conditions, modifies clauses, and representation invariants, enable more powerful checks such as determining violations of information hiding, memory management errors, and dangerous data sharing or unexpected aliasing. ESC/Java (Extended Static Checking [DLNS98] for Java) also relies on annotated source code and can catch many common programming mistakes such as array index bounds errors, null dereference errors, type-cast errors, and deadlocks and race conditions in multi-threaded programs. The checker uses an automatic theorem prover to reason about the semantics of conditional statements, loops, procedure and method calls, exceptions, and mutex locks.

As an aside, I leave for the more advanced student, the upper-level elective courses, and the undergraduate researcher two other classes of formal methods tools: design checkers such as Nitpick [JD96] and Alcoa [Ja00], which are still in the research incubator; and theorem provers, which still require sophisticated users. Design checkers have much promise in their use in upper-level software engineering courses, but need more time to mature. Theorem provers require more expertise than we can expect our students to acquire in one semester, all the while learning other course material.

3 Specific Undergraduate Courses

Introduction to Programming. Here we can teach the concepts of specification and verification but likely only informally and at a high-level. Still, acclimating students to the difference between a specification and code and to the idea of verification in addition to testing is a good first step. Students should get in the habit of writing informal specifications, loop invariants, and termination arguments in their comments.

Data Structures and Algorithms. This course lends itself naturally to introducing and exercising notions of abstraction, representation invariants, inductive proofs, and state machines.

Programming Principles. This is the traditional course that many schools use to teach the concepts of program specification and verification. It may make sense to revisit this course if some of the material is distributed across the others. At Carnegie Mellon we use this course to teach the functional programming language paradigm (we use ML) with a heavy emphasis on types (as weak specifications), modules (interfaces versus implementation; composition and abstraction techniques), and the course mantra “code with proof in mind” (recursive programs lend themselves to inductive proofs).

Programming Languages. This course provides the opportunity to revisit more formally the concepts perhaps learned only informally during the students’ first year. For example, we can give semantics for imperative and object-oriented programming languages in terms of state machines. We can use logic programming languages to illustrate advantages and disadvantages of using executable specifications, i.e., where specifications are code and vice versa.

Compilers. Translators and interpreters, by definition, provide rich examples of abstraction mappings (defining or simulating one machine in terms of another). Correctness preserving transformations require statements of invariants (formal or not) and soundness arguments (formal or not). Target machines (compiler back-ends) are just state machines. This course comes close to the ideal, where students are using some elements of formal methods without realizing it.

Software Engineering. Students can complement the use of informal CASE tools and semi-formal design methods such as UML with the use of formal ones, e.g., model checkers and specification checkers. Here would be the place to introduce design checkers such as Nitpick and Alcoa.

Computer Architecture. Students can use model checkers such as SMV to verify properties of simple circuits, simple processor designs, bus protocols, and cache coherence protocols.

Operating Systems. Students can use model checkers to check safety properties, e.g., freedom from deadlock, of various mutual exclusion algorithms (e.g., Peterson’s tie-breaker algorithm or Lamport’s bakery algorithm), and with various synchronization primitives (e.g., semaphores, mutex locks, condition variables).

Networking. Students can use model checkers to check properties of simple network protocols. (A Carnegie Mellon undergraduate did an honors thesis using Nitpick to discover a flaw in the Mobile IPv6 protocol [JNW00].)

Databases. We can use relational databases and other data models to discuss all flavors of invariants. Transactional systems require understanding executions, observable behavior, consistency (correctness) constraints, and interference due to concurrency.

User Interfaces. Modeling the user, environment, and system as a set of interacting concurrent processes can provide the foundation for usage scenarios. Using model checkers such as FDR makes sense here.

Undergraduate upper-level electives such as Artificial Intelligence and Graphics presumably offer other opportunities as well.

4 Future Work

All the real work is future work. The ideas sketched in this paper are just ideas of what might be possible. We are faced with working out the details. The biggest obstacle is getting “buy-in” from our colleagues: convincing co-instructors, curricula committees, and administrators that integrating formal methods unintrusively is a good thing to do.

Also, while philosophically in Section 1 we argued to emphasize concepts, not notation, concrete notation is the conveyor of abstract ideas. To effectively weave in the teaching of elemental concepts with existing courses means adapting notations and methods to the languages already in use. For example, using ESC/Java makes sense to use in a data structures and algorithms course taught in Java; but using Z tools for that same course may require too much additional overhead.

The nitty-gritty hard future work is in thinking of the examples to use in lectures, in designing appropriate homework and exam problems, and in making learning these concepts and tools enjoyable.

We do not have to do everything, and we do not have to do everything all at once. We can begin, for example, by discussing state machines in a programming languages course, and by introducing model checkers in a homework assignment or project of a computer architecture course. The main thing is to start doing something!

References

- [CGP99] Clarke, E.M., O. Grumberg, and D.A. Peled: *Model Checking*, MIT Press, 1999.
- [CSRC00] Compaq Systems Research Center,
<http://www.research.compaq.com/SRC/esc/Esc.html>
- [DLNS98] Detlefs, D., K. Rustan M. Leino, G. Nelson, and J.B. Saxe: Extended Static Checking, Compaq SRC Research Report 159, 1998.
- [EGHT94] Evans, D., J. Gutttag, J.J. Horning, and Y.M. Tan: LCLint: A Tool for Using Specifications to Check Code, *SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.
- [Gr81] Gries, D.: *The Science of Programming*, Springer-Verlag, 1981.
- [GH93] Gutttag, J.V. and J.J. Horning, editors: *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
- [JD96] Jackson, D. and C. Damon: “Nitpick Reference Manual,” Carnegie Mellon University Technical Report CMU-CS-96-109, Computer Science Department, Pittsburgh, PA, January 1996.
- [Ja00] Jackson, D.: “Alloy: A Lightweight Object Modelling Notation,” MIT Technical Report 797, February 2000.
- [JNW00] Jackson, D., Y. Ng, and J.M. Wing: “A Nitpick Analysis of Mobile IPv6,” to appear in *Formal Aspects of Computing*, accepted January 2000.

Invited Talk: Making Mathematical Methods More Practical for Software Developers (Abstract)

David L. Parnas, P.Eng

NSERC/Bell Industrial Research Chair in Software Engineering
Director of the Software Engineering Programme
Department of Computing and Software
Faculty of Engineering, McMaster University
Hamilton, Ontario Canada L8S 4L7

There is a startling contrast between classical engineering disciplines and Software Engineering. Electrical, Mechanical, and Civil Engineers learn a lot of mathematics and they actively use that mathematics when designing new products or processes. In contrast, most software developers see mathematics as nearly irrelevant to their work and some educational programs deliberately neglect traditional mathematics.

At the Software Engineering Research Group at McMaster University we believe that one of the reasons for the limited use of mathematics by software developers is the inappropriateness of what is often called "formal methods". Many advocates seem to view "formal methods" as an "add on" to programming. We view mathematics as an integral part of the work of the Engineer.

There are obvious problems in using such tools as Z and VDM. Complete specifications written in these languages are often longer, and more difficult to understand than the code itself. Writing specifications in these languages often looks suspiciously like programming and practitioners wonder why they should not just write a program and let it serve as its own documentation.

We believe that there are two problems with the best known methods:

1. They provide models of software rather than summary descriptions of behavior
2. They use complex expressions that are difficult to parse and understand.

Our response has been:

1. to focus on functional/relational methods that summarize the required behavior rather than show how it might be implemented.
2. to introduce and define the meaning of multi-dimensional mathematical expressions (mathematical tables).

These two ideas have allowed us to build a set of prototype tools. Using these tools we have been able to explore what mathematics can do for the program designer. The mathematics we use is old, and we use it in the same way that mathematics is used in classical engineering.

The talk will describe the fundamentals of our approach and then describe some of the prototype tools and how they would be used. We will demonstrate that classical mathematics is superior in many ways to newer "formal methods".

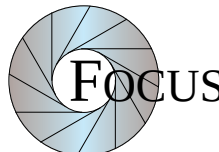
Step by Step to Histories^{*}

Max Breitling and Jan Philipps



Institut für Informatik
Technische Universität München
80290 München
Germany

`{max.breitling|jan.philipps}@in.tum.de`



Abstract. The behavior of reactive systems is typically specified by state machines. This results in an operational description of how a system produces its output. An alternative and more abstract approach is to just specify the relation between the input and output histories of a system. In this work, we propose a way to combine state-based and history-based specifications: Abstract communication history properties of system components can be derived from temporal logic properties of state machines. The history properties can then be used to deduce global properties of a complete system.

1 Introduction

To allow precise reasoning about a hard- or software system, a mathematical foundation for both systems and properties is a prerequisite. For some classes of systems—in particular, clocked hardware—temporal logics have been used successfully to formalize and to reason about their properties.

Temporal logic and model checking are less successful, however, when the dataflow between loosely coupled components that communicate asynchronously via communication channels is examined. For such systems, a black box view which just relates input and output is more useful than the state-based glass box view of a component. Black box properties of dataflow components and systems can be concisely formulated as relations over the communication history of components [7,8]; such properties are inherently modular and allow easy reasoning about the global system behavior.

For individual data flow components, however, a state-based glass box view is helpful. State machines are good design documents for a component’s implementation. Moreover, they provide an operational intuition that can aid in structuring proofs: Safety properties, for example, are typically shown using induction over the machine transitions.

In this paper we show —based on the ideas of Broy’s verification of the Alternating Bit Protocol [6]— how specifications of the black box view of a system or system component can be systematically derived from state machine specifications of the components. Thus we bridge the gap between techniques for

* This work is supported by the DFG within the Sonderforschungsbereich 342.

easy verification of dataflow properties and more operational descriptions that are close to efficient implementations of a system.

The paper is structured as follows: In the next section we introduce some mathematical concepts and notations. § 2 and § 3 describe history specifications for the black box view, and state machines for the glass box view of a component, respectively. In § 4 we present verification rules for temporal logic properties that are used in § 5 to relate the black box and glass box views of a component. In § 6 we demonstrate how the black box views support compositional reasoning about a system. The conclusion in § 7 gives an outlook on future work.

2 History Relations

A dataflow system is a network of components. Each component has input and output ports. Ports of different components are connected by directed channels. Communication over these channels is asynchronous, message buffers are assumed to be unbounded. The black box view of a dataflow system regards only the communication between components and abstracts from the internal workings inside the components.

Systems in the black box view are modeled as relations over communication histories. The relations are expressed using formulas in predicate logic where the formula's free variables range over *streams*. Each free variable represents the communication history over one of the component's input or output ports.

There is a rich mathematical basis for this system model [7,8]; this section contains only a short overview over the concepts used in the rest of the paper.

2.1 Streams

The communication history between components is modeled by *streams*. A stream is a finite or infinite sequences of messages. Finite streams can be enumerated, for example: $\langle 1, 2, 3, \dots, 10 \rangle$; the empty stream is denoted by $\langle \rangle$. For a set of messages Msg , the set of finite streams over Msg is denoted by Msg^* , that of infinite streams by Msg^∞ . By Msg^ω we denote $\text{Msg}^* \cup \text{Msg}^\infty$. Given two streams s, t and $j \in \mathbb{N}$, $\#s$ denotes the length of s . If s is finite, $\#s$ is the number of elements in s ; if s is infinite, $\#s = \infty$. We write $s \frown t$ for the concatenation of s and t . If s is infinite, $s \frown t = s$. We write $s \sqsubseteq t$, if s is a prefix of t , i.e. if $\exists u \in \text{Msg}^\omega \bullet s \frown u = t$. The j -th element of s is denoted by $s.j$, if $1 \leq j \leq \#s$; it is undefined otherwise. $\text{ft}.s$ denotes the first element of a stream, i.e. $\text{ft}.s = s.1$, if $s \neq \langle \rangle$.

The prefix relation $\sqsubseteq$ is a partial order. The set of streams Msg^ω together with $\sqsubseteq$ forms a complete partial order (CPO); the empty stream $\langle \rangle$ is the least element in this CPO. This means that for every chain $\{s_i \mid i \in \mathbb{N}\}$ of streams, where for each i : $s_i \sqsubseteq s_{i+1}$, there is a unique least upper bound $\bigsqcup \{s_i \mid i \in \mathbb{N}\}$. A predicate Φ where the free variables range over streams M^ω is *admissible*, if it holds for the limit of a chain of valuations for its variables, provided that it

holds for each element of the chain. We then write $\text{adm } \Phi$. Syntactical criteria for admissibility can be found in [12].

Stream concatenation and the prefix order can be extended pointwise to tuples of streams; continuity of functions and admissibility of prefix can also be defined for stream tuples.

2.2 Component Specification

Figure 1 shows the system structure of a bounded transmission system with three components: a sender, a receiver, and a buffer with a capacity for $N \geq 2$ data messages. For now, we just examine the sender.

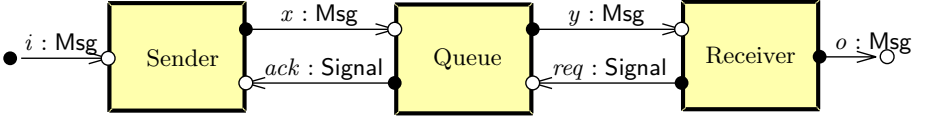


Fig. 1. Bounded Buffer

The black box view of the sender is specified by giving a set of input channel identifiers I and a set of output channel identifiers O (where $I \cap O = \emptyset$) to define its interface. The behavior is specified by a predicate with free variables from I and O . Each channel identifier has an assigned type that describes the set of messages allowed on that channel. Typically, we write the specification in the following style:

<i>Sender</i>
in $i : \text{Msg}, \text{ack} : \text{Signal}$ out $x : \text{Msg}$
$x \sqsubseteq i$ $\#x = \min(\#i, 1 + \#\text{ack})$

Intuitively, the sender behaves as follows: On channel x it forwards the messages it receives on channel i , in the same order, but possibly not all of them. This safety property is denoted by the first assertion. The second assertion contains both a safety and liveness part: For liveness, it demands the sender to send *at least* the number of messages it receives on i ; but only as long as each message is acknowledged; the safety part asserts that *at most* this number is received.

The specification pattern of the sender is typical for history specifications: The specification is a conjunction of prefix expressions which restrict the data values on the output channels, and (in-)equalities, which specify the length of the output histories in terms of the length of the input histories.

2.3 Component Composition

The history relation of a composed system can be derived from the history relations of its components. Components may share input channels, but each output channel must be controlled by only one single component. This is captured in the definition of compatibility: Two components $\mathcal{S}_1$ and $\mathcal{S}_2$ are *compatible* if they do not share output channels: $O_{\mathcal{S}_1} \cap O_{\mathcal{S}_2} = \emptyset$.

The result of the composition, noted as $\mathcal{S}_1 \otimes \mathcal{S}_2$, is again a system specification. Channels with identical names are connected, the output of the composition is the union of the two component's output channels, and the input of the composition consists of those input channels that remain unconnected.

$$I_{\mathcal{S}_1 \otimes \mathcal{S}_2} \stackrel{\text{df}}{=} (I_{\mathcal{S}_1} \cup I_{\mathcal{S}_2}) \setminus (O_{\mathcal{S}_1} \cup O_{\mathcal{S}_2}), \quad O_{\mathcal{S}_1 \otimes \mathcal{S}_2} \stackrel{\text{df}}{=} O_{\mathcal{S}_1} \cup O_{\mathcal{S}_2}$$

The behavior of the composed system is defined as the conjunction of the component behavior predicates.

3 State Machines

State machines are a more operational way to specify dataflow components than history relations. We use the term *state machine* both for the abstract syntax (state transition systems, § 3.2) and for the concrete graphical representation (state transition diagrams, § 3.4). The executions of state transition systems are defined in § 3.3.

First we give a formal definition of variable valuations for an assertion. Variable valuations allow us to talk about the validity of assertions in the different states of a state machine execution.

3.1 Variable Valuations

We assume an (infinite) set $\mathbf{Var}$ of variable names. A valuation α is a function that assigns to each variable in $\mathbf{Var}$ a value from the variable's type. By $\text{free}(\Phi)$ we denote the set of free variables in a logical formula Φ . If an assertion Φ evaluates to true when each variable $v \in \text{free}(\Phi)$ is replaced by $\alpha(v)$, we write $\alpha \models \Phi$.

Variable names can be *primed*: For example, v' is a new variable name that results from putting a prime behind v . We extend priming to sets $V' \stackrel{\text{df}}{=} \{ v' \mid v \in V \}$ and to valuations: Given a valuation α of variables in $\mathbf{Var}$, α' is a valuation of variables in V' with $\alpha'(v') = \alpha(v)$ for all variables $v \in \mathbf{Var}$. Priming can also be extended to predicates, functions and other expressions: If Ψ is an assertion with $\text{free}(\Psi) \subseteq V$, then Ψ' is the assertion that results from priming all free variables.

Note that an unprimed valuation α assigns values to all *unprimed* variables, while a primed valuation β' only assigns values to all *primed* variables. If an assertion Φ contains both primed and unprimed variables, we need two valuations to determine its truth. If Φ evaluates to true when each unprimed variable $v \in \text{free}(\Phi)$ is replaced by $\alpha(v)$ and each primed variable $v' \in \text{free}(\Phi)$ is replaced

by $\beta'(v)$, we write $\alpha, \beta' \models \Phi$. Two valuations *coincide* on a subset $V \subseteq \text{Var}$ if $\forall v \in V \bullet \alpha(v) = \beta(v)$. We then write $\alpha \stackrel{V}{=} \beta$.

3.2 State Transition Systems

A state transition system is a tuple $\mathcal{S} = (I, O, A, \mathcal{I}, \mathcal{T})$, where I, O, A are sets of variables. A state of our system is described by a valuation α , that assigns values to all variables in $V \stackrel{\text{def}}{=} I \cup O \cup A$. $\mathcal{I}$ is an assertion with $\text{free}(\mathcal{I}) \subseteq V$ that characterizes the initial states of the state transition system. $\mathcal{T}$ is a finite set of transitions; each transition $\tau \in \mathcal{T}$ is an assertion with $\text{free}(\mathcal{T}) \subseteq V \cup V'$. The tuple elements have to obey the following restrictions.

The sets I and O , with $I \cap O = \emptyset$, contain the input and output channel variables. The variables range over finite streams which represent the communication history to and from the component. The set A contains local state attributes, as e.g. a variable σ for a control state and variables for data states. Additionally, A contains for every $i \in I$ a variable i° . These variables hold the part of the external input stream i that has already been processed by $\mathcal{S}$. The restrictions on the initialization and transition assertions defined below ensure that $i^\circ \sqsubseteq i$ always holds. We can therefore define i^+ as the part of the message history that has not yet been processed by $i = i^\circ \frown i^+$.

The assertion $\mathcal{I}$ characterizes the initial states of the system. We require $\mathcal{I}$ to be satisfiable for arbitrary input streams

$$\exists \alpha \bullet \alpha \models \mathcal{I} \quad \wedge \quad \left(\forall \beta \bullet \beta \stackrel{O \cup A}{=} \alpha \Rightarrow \beta \models \mathcal{I} \right)$$

and to assert that initially no input has been processed and no output has yet been produced:

$$\mathcal{I} \Rightarrow \bigwedge_{i \in I} i^\circ = \langle \rangle \wedge \bigwedge_{o \in O} o = \langle \rangle$$

The set $\mathcal{T}$ contains the allowed transitions of $\mathcal{S}$. Every transition $\tau \in \mathcal{T}$ is an assertion over $V \cup V'$ and relates states with their successor states. Unprimed variables in τ are valuated in the current state, while primed variables are valuated in the successor state. All transitions must guarantee that the system does not take back messages it already sent, that it can not undo the processing of input messages, that it can only read messages that have been sent to the component and that it does not change the variables for input streams, since these are controlled by the environment:

$$\tau \Rightarrow \bigwedge_{o \in O} o \sqsubseteq o' \wedge \bigwedge_{i \in I} i^\circ \sqsubseteq i^{\circ'} \wedge \bigwedge_{i \in I} i^{\circ'} \sqsubseteq i \wedge \bigwedge_{i \in I} i = i'$$

In addition to the transitions in $\mathcal{T}$, there is an implicit *environment transition* τ_ϵ . This transition is defined to allow the environment to extend the input, while it leaves the controlled variables $v \in O \cup A$ unchanged:

$$\tau_\epsilon \Leftrightarrow \bigwedge_{v \in O \cup A} v = v' \wedge \bigwedge_{i \in I} i \sqsubseteq i'$$

A transition is *enabled* in a state α , written as $\alpha \models \text{En}(\tau)$, iff there is a state β such that $\alpha, \beta' \models \tau$.

3.3 Executions

An *execution* of a STS $\mathcal{S}$ is an infinite stream ξ of valuations that satisfies the following three requirements:

1. The first valuation in ξ satisfies the initialization assertion:

$$\xi.1 \models \mathcal{I}$$

2. Each pair of subsequent valuations $\xi.k$ and $\xi.(k+1)$ in ξ are related either by a transition in $\mathcal{T}$ or by the environment transition τ_ϵ :

$$\xi.k, \xi'.(k+1) \models \tau_\epsilon \vee \bigvee_{\tau \in \mathcal{T}} \tau$$

3. Each transition $\tau \in \mathcal{T}$ of the STS is taken infinitely often in an execution, unless it is disabled infinitely often (weak fairness):

$$(\forall k \bullet \exists l \geq k \bullet \xi.l \models \neg \text{En}(\tau)) \vee (\forall k \bullet \exists l \geq k \bullet \xi.l, \xi'.(l+1) \models \tau)$$

By $\langle\langle \mathcal{S} \rangle\rangle$ we denote the set of all executions of a system $\mathcal{S}$.

3.4 State Transition Diagrams

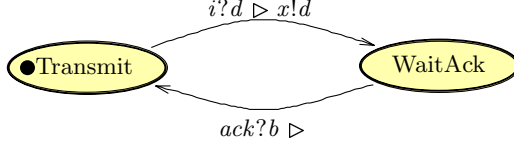
Typically, state transition systems are specified by *state transition diagrams* (STDs). We use a subset of the STD syntax from the CASE tool AUTOFOCUS [9]. STDs are directed graphs where the vertices represent (control) states and the edges represent transitions between states. One vertex is designated as *initial state*; graphically this vertex is marked by an opaque circle in its left half. Edges are labeled; each label consists of four parts, represented by the following schema:

$$\{Precondition\} Inputs \triangleright Outputs \{Postcondition\}$$

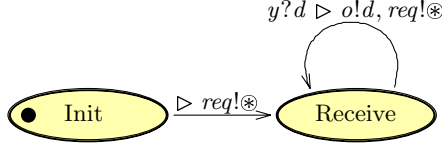
Inputs and *Outputs* stand for lists of expressions of the form $i?x$ and $o!exp$ ($i \in I$, $o \in O$) respectively, where x is a constant value or a (transition-local) variable of the type of i , and exp is an expression of the type of o . The *Precondition* is a boolean formula containing data state variables and transition-local variables as free variables, while *Postcondition* and *exp* may also contain primed variables. The distinction between pre- and postconditions does not increase the expressiveness, but improves readability. If the pre- or postconditions are equivalent to *true*, they can be omitted.

The informal meaning of a transition is as follows: If the available messages on the input channels can be matched with *Inputs*, the precondition is true and the postcondition can be made true by assigning proper values to the primed

Sender



Receiver



Queue

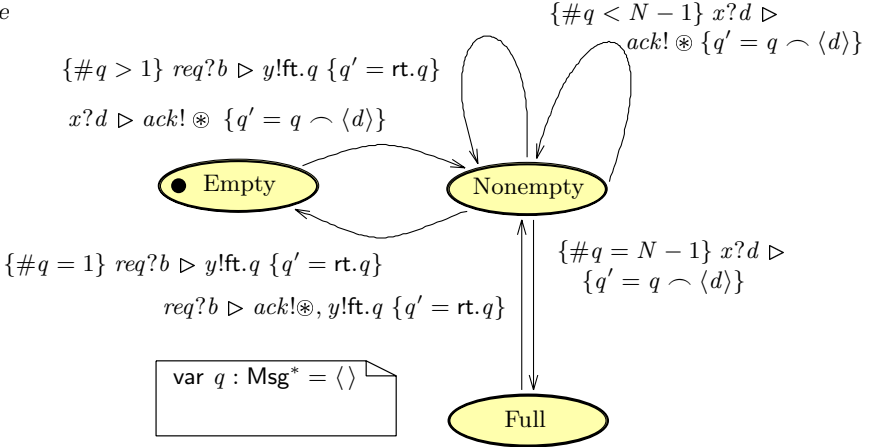


Fig. 2. Sender, Receiver and Queue STDs

variables, then the transition is enabled. If the transition is executed, the inputs are read, the outputs are written and the postcondition is made true.

Figure 2 shows the STDs of sender, queue and receiver of the transmission system (see Fig. 1). Again, we focus on the sender component: If the sender receives some data d on channel i , this message is immediately forwarded on x , and the system starts waiting for an acknowledgment message on channel ack . When the acknowledgment is received, the sender is ready to receive the next message from i .

State transition diagrams can be encoded schematically as state transition systems. For the sender component, the variable sets are defined as follows: $I = \{i, ack\}$, $O = \{x\}$ (see Fig. 1), $A = \{i^\circ, ack^\circ, \sigma\}$. The state attributes

consist of the processed message stream for each of the two input channels, and a variable σ to hold the current control state.

The initial assertion $\mathcal{I}$ of the sender is defined as:

$$\sigma = \text{Transmit} \wedge i^\circ = \langle \rangle \wedge \text{ack}^\circ = \langle \rangle \wedge x = \langle \rangle$$

The transition τ_1 from the state *Transmit* to the state *WaitAck* in the sender STD is encoded as the following assertion:

$\exists d. \quad \sigma = \text{Transmit}$ $\wedge \sigma' = \text{WaitAck}$ $\wedge \#i^\circ < \#i$ $\wedge \text{ft}.i^+ = d$ $\wedge i^{\circ'} = i^\circ \smallfrown \langle d \rangle$ $\wedge x' = x \smallfrown \langle d \rangle$ $\wedge \text{ack}^{\circ'} = \text{ack}^\circ$ $\wedge i = i' \wedge \text{ack} = \text{ack}'$	<i>We move from the source state to the target state.</i> <i>There are unread messages in channel i.</i> <i>Let d be the first of them,</i> <i>which we consume</i> <i>and send on channel x,</i> <i>whereas we don't read from channel ack,</i> <i>and leave the input channels unchanged.</i>

The second transition τ_2 of the sender can be encoded similarly. Note that the initialization and transition assertion obey the restrictions from § 3.2.

The queue and receiver components lead to similar transition assertions. In case of the queue component, there is an additional variable q in A . Initially, $q = \langle \rangle$; the transitions change q according to the queue STD. A more detailed explanation of the translation of STDs to STS assertions can be found in [2].

4 Verification Rules

A common technique for formalizing and verifying properties of state transition system executions is temporal logic [11]. For the state machines of § 3 we are not interested in general temporal logic properties, but only in two special cases: invariants for *safety* properties and *leadsto* properties for liveness. This section introduces verification rules for these two property classes. Soundness proofs of these and other rules —expressed in a UNITY-like formalism— can be found in [2].

Note that both invariance and leadsto properties relate single states in an STS execution; in § 5 these properties are used to express properties about the complete communication history of executions.

4.1 Invariance Properties

To show that a STS $\mathcal{S}$ fulfills a safety property, we use invariants. For a system $\mathcal{S} = (I, O, A, \mathcal{I}, \mathcal{T})$, an assertion Φ with $\text{free}(\Phi) \subseteq I \cup O \cup A$ is an invariant, written as $\mathcal{S} \models \Box\Phi$, if Φ evaluates to true for each state in all executions of $\mathcal{S}$:

$$\mathcal{S} \models \Box\Phi \quad \Leftrightarrow \quad \forall \xi \in \langle\langle \mathcal{S} \rangle\rangle \bullet \forall k \bullet \xi.k \models \Phi$$

To prove Φ to be an invariant, we have to show that Φ holds initially, and remains true under each transition $\tau \in \mathcal{T}$ as well as under the environment transition τ_ϵ :

$$\frac{\begin{array}{l} \mathcal{I} \Rightarrow \Phi \\ \Phi \wedge \tau \Rightarrow \Phi' \quad \text{for all } \tau \in \mathcal{T} \\ \Phi \wedge \tau_\epsilon \Rightarrow \Phi' \end{array}}{\mathcal{S} \models \Box \Phi}$$

Example. For the sender, the output on channel x is always equal to the sequence of messages from i that have already been consumed:

$$\text{Sender} \models \Box x = i^\circ$$

The first condition of the invariant rule is fulfilled, since for the sender initially both x and i° are empty (see § 3.4). The other two premises are fulfilled since the sender transition τ_1 appends a single message to both x and i° ; for transitions τ_2 and τ_ϵ we observe that both x and i° remain unchanged.

4.2 Leadsto Properties

Progress of a system can be expressed using the leadsto operator $\Phi \leadsto \Psi$, which states that whenever Φ is **true** for a state in an execution, then Ψ will be **true** in the same or in a subsequent state in the execution. Usually, the leadsto operator is defined in temporal logic as $\Box(\Phi \Rightarrow \Diamond\Psi)$, but for our purposes the following semantic definition of $\mathcal{S} \models \Phi \leadsto \Psi$ is sufficient:

$$\mathcal{S} \models \Phi \leadsto \Psi \Leftrightarrow \forall k \bullet (\xi.k \models \Phi) \Rightarrow (\exists l \geq k \bullet \xi.l \models \Psi)$$

For the leadsto operator, too, there are verification rules:

$$\frac{\begin{array}{l} \text{For all transitions } \tau \in \mathcal{T} \cup \{\tau_\epsilon\}: \\ \quad \Phi \wedge \neg \Psi \wedge \tau \Rightarrow \Phi' \vee \Psi' \\ \text{For a transition } \tau \in \mathcal{T}: \\ \quad \Phi \wedge \neg \Psi \Rightarrow \text{En}(\tau) \\ \quad \text{and} \\ \quad \Phi \wedge \neg \Psi \wedge \tau \Rightarrow \Psi' \end{array}}{\mathcal{S} \models \Phi \leadsto \Psi} \quad \frac{\begin{array}{l} \text{For a transition } \tau \in \mathcal{T}: \\ \quad \#o = k \wedge k < \mathcal{L} \Rightarrow \text{En}(\tau) \\ \quad \text{and} \\ \quad \#o = k \wedge k < \mathcal{L} \wedge \tau \Rightarrow \#o' > k \end{array}}{\mathcal{S} \models \#o = k \wedge k < \mathcal{L} \leadsto \#o > k}$$

The first rule is a standard verification rule for liveness under weak fairness [10,11]: There is a helpful transition $\tau \in \mathcal{T}$ which is enabled in all states where Φ holds, and which leads into a state where Ψ holds (second premise). The other transitions are not harmful in that they leave Φ invariant. Thus, the helpful

transition remains enabled until it is, by weak fairness, executed. The second rule, the *output extension rule*, is a specialization of the first rule. It is used to prove that an output stream exceeds a certain length k provided that sufficient input is available. This can be described by an $\mathbb{N}$ -valued length expression $\mathcal{L}$ with $\text{free}(\mathcal{L}) \subseteq I$ which is monotonic in its free variables. The main difference to the first rule is that it is not necessary to show the safety premises of the first rule: For this special case they hold trivially, since channel valuations are monotonic with respect to $\sqsubseteq$, and due to its monotonicity the length expression $\mathcal{L}$ can be proven to be nondecreasing [2]. The left hand side of the output extension's conclusion rule can be strengthened by an arbitrary predicate Ψ , if the left hand sides of the premises are also strengthened by Ψ .

Besides the two rules above, there are a number of additional rules for the leadsto operator: transitivity, weakening of the right hand side, strengthening of the left hand side. The *disjunction rule* combines two leadsto properties: If $\mathcal{S} \models \Phi_1 \rightsquigarrow \Psi$ and $\mathcal{S} \models \Phi_2 \rightsquigarrow \Psi$, then also $\mathcal{S} \models (\Phi_1 \vee \Phi_2) \rightsquigarrow \Psi$. Moreover, invariants can be introduced and eliminated on both sides of the operator.

Example. Again regarding the sender, we want to show

$$\text{Sender} \models \#x = k \wedge k < \min(\#i, 1 + \#ack) \rightsquigarrow \#x > k$$

which expresses that the output on x is extended, provided there is sufficient input on i and ack expressing that the length of the output on x is reaching at least the limit $\min(\#i, 1 + \#ack)$.

For $\sigma = \text{Transmit}$, we use the output extension rule with τ_1 as the helpful transition, since it produces output on x . The last condition of the rule is easy to prove, since τ_1 implies the extension of x by $x' = x \smallfrown \langle d \rangle$, so that $\#x = k \wedge \tau_1 \rightsquigarrow \#x' > k$ is trivial. For the second condition we have to prove that τ_1 is enabled. If we assume $\sigma = \text{Transmit}$, it is enabled iff there is some message on the channel i , i.e. iff i is longer than its consumed part i° . Using the safety invariant from above, this can be derived as follows:

$$\#i \geq \min(\#i, 1 + \#ack) > k = \#x = \#i^\circ$$

For $\sigma = \text{WaitAck}$, transition τ_1 is not enabled. Instead, we use the standard weak fairness rule to show that by transition τ_2 state *WaitAck* is entered. The two results can be combined with the transitivity and disjunction rules to derive the property

$$\begin{aligned} \text{Sender} \models & ((\sigma = \text{WaitAck} \vee \sigma = \text{Transmit}) \\ & \wedge \#x = k \wedge k < \min(\#i, 1 + \#ack)) \rightsquigarrow \#x > k \end{aligned}$$

It can be shown that $\sigma = \text{WaitAck} \vee \sigma = \text{Transmit}$ is an invariant; its elimination results in the property above [2].

5 History Properties

We introduced two ways to specify reactive systems: history relations and state machines. The two views describe quite different views on a system: Using the black box views of history relations, we model the I/O behavior with streams; the relations do not refer to any internals of the components and do not describe how this behavior is achieved. Using state machines we concentrate on single steps of the system, referring to the component internals. In this section, we close the gap between state machines and black box views.

Within a state machine execution ξ , changes in the valuations for the input and output variables in $I \cup O$ are restricted to extensions. Thus the valuations of each input and output variable within an execution form a chain, and for each execution and each variable $v \in I \cup O$ there is a least upper bound

$$\lceil \xi \rceil(v) \stackrel{\text{df}}{=} \bigsqcup \{ (\xi.k)(v) \mid k \in \mathbb{N} \}$$

Note that $\lceil \xi \rceil(v)$ is only defined for the input and output variables, not for the attribute variables A of a state machine.

The *black box view* of a state machine is a set of valuations for the variables $I \cup O$. It is denoted by $\llbracket \mathcal{S} \rrbracket$ and defined via the least upper bounds of the input and output histories of the machine's executions. For each execution ξ in $\langle\langle \mathcal{S} \rangle\rangle$, there is a valuation α in $\llbracket \mathcal{S} \rrbracket$ which assigns to the channel variables in $I \cup O$ the limits of the channel variable valuations of ξ :

$$\llbracket \mathcal{S} \rrbracket \stackrel{\text{df}}{=} \{ \alpha \mid \exists \xi \in \langle\langle \mathcal{S} \rangle\rangle \bullet \bigwedge_{i \in I} \alpha(i) = \lceil \xi \rceil(i) \wedge \bigwedge_{o \in O} \alpha(o) = \lceil \xi \rceil(o) \}$$

Since both the proper transitions $\tau \in \mathcal{T}$ and the environment transition τ_ϵ of a state machine allow arbitrary extension of the input variable valuations, it is possible to successively approximate an arbitrary input history. This means that the black box view $\llbracket \mathcal{S} \rrbracket$ is total with respect to the input variables of $\mathcal{S}$: For an arbitrary input there is always some reaction of the system. Formally, this reads as: For each valuation α for the variables $I \cup O$ there exists a valuation β for $I \cup O$ such that

$$\alpha \stackrel{I}{\leq} \beta \quad \text{and} \quad \beta \in \llbracket \mathcal{S} \rrbracket$$

5.1 Safety Properties

In practice, it is difficult to directly use the black box semantics $\llbracket \mathcal{S} \rrbracket$ of a state machine. Instead, we derive properties of the black box view from properties of the state machine. Technically, a property of the black box view $\llbracket \mathcal{S} \rrbracket$ is a predicate Φ with $\text{free}(\Phi) \subseteq I \cup O$ which is valid for each valuation in a system's black box view:

$$\forall \alpha \in \llbracket \mathcal{S} \rrbracket \bullet \alpha \models \Phi$$

We then write $\llbracket \mathcal{S} \rrbracket \Rightarrow \Phi$.

If Φ is an admissible invariance property of a state machine, it holds not only in every state of a system run, but also for the complete communication history:

$$\frac{\begin{array}{l} \text{free}(\Phi) \subseteq I \cup O \\ \text{adm } \Phi \\ \mathcal{S} \models \Box \Phi \end{array}}{\llbracket \mathcal{S} \rrbracket \Rightarrow \Phi}$$

The validity of the rule follows from the fact that the valuations of the channel variables I and O form a chain. Because it is invariant, Φ holds for every element of the chain. Because of admissibility, it also holds in the limit.

Example. In § 4.1 we showed that $x = i^\circ$ is an invariant of the sender. Moreover, $x \sqsubseteq i$ is also an invariant since $i^\circ \sqsubseteq i$. This predicate is also admissible [12], and thus we can directly conclude

$$\llbracket \text{Sender} \rrbracket \Rightarrow x \sqsubseteq i$$

This means that the sender STD implies the first half of the sender's history specification in § 2.2. Similarly, we can show $\llbracket \text{Sender} \rrbracket \Rightarrow \#x \leq 1 + \#ack$.

5.2 Progress Properties

In general, progress properties expressed with the leadsto operator $\leadsto$ cannot be lifted to complete executions. However, output extension properties (§ 4.2) can be used to derive liveness properties of a state machine's black box view. In the following rule, $\mathcal{L}$ is a monotonic $\mathbb{N}$ -valued expression with $\text{free}(\mathcal{L}) \subseteq I$, as used in the output extension rule.

$$\frac{\mathcal{S} \models \#o = k \wedge k < \mathcal{L} \leadsto \#o > k}{\llbracket \mathcal{S} \rrbracket \Rightarrow \#o \geq \mathcal{L}}$$

To see the validity of the rule, assume that the premise holds, but not the conclusion. Thus, there is an execution ξ of $\mathcal{S}$ such that the length of the limit of the channel valuations for o is strictly less than the limit of the valuations of $\mathcal{L}$; in particular, it is equal to a natural number k . This means that there is an earliest state $\xi.n$ in the execution where the length of the output valuation for o reaches k . Moreover, there is a state $\xi.m$ where $\mathcal{L}$ is larger than k . Since channel valuations cannot become shorter, and $\mathcal{L}$ is monotonic, this means that in all states $\xi.p$, where $p \geq \max(n, m)$ the left hand side of the premise is fulfilled, but the right hand side never holds. This violates the assumption that the premise is valid.

Example. In § 4.2 we showed

$$Sender \models \#x = k \wedge k < \min(\#i, 1 + \#ack) \rightsquigarrow \#x > k$$

We can now directly use the above rule to derive

$$\llbracket Sender \rrbracket \Rightarrow \#x \geq \min(\#i, 1 + \#ack)$$

Together with the safety properties shown above, this implies the second part of the sender's history specification.

6 Black Box Composition

We now have a closer look on the complete transmission system of Fig. 1. The sender pushes data to the queue and waits for acknowledgments and the receiver requests data from the queue; the queue itself stores up to N ($N \geq 2$) data messages.

The behavior of the three components is defined in Fig. 2 by STDs. Using the techniques of this paper, we can show that the receiver and the queue imply the following history relations:

<i>Queue(N)</i>	<i>Receiver</i>
$\begin{array}{l} \text{in } x : \text{Msg}, \text{req} : \text{Signal} \\ \text{out } \text{ack} : \text{Signal}, y : \text{Msg} \end{array}$	$\begin{array}{l} \text{in } y : \text{Msg} \\ \text{out } \text{req} : \text{Signal}, o : \text{Signal} \end{array}$
$\begin{array}{l} y \sqsubseteq x \\ \#y \geq \min(\#x, \#req) \\ \#ack = \min(\#x, \#req + N - 1) \end{array}$	$\begin{array}{l} o \sqsubseteq y \\ \#o \geq \#y \\ \#req = 1 + \#y \end{array}$

By black box composition, the history relation of the complete system is specified as follows. The behaviour is simply described by the conjunction of the component properties.

<i>System(N)</i>
$\begin{array}{l} \text{in } i : \text{Msg} \\ \text{out } o : \text{Signal}, x : \text{Msg}, \text{ack} : \text{Signal}, y : \text{Msg}, \text{req} : \text{Signal} \end{array}$
$\begin{array}{l} x \sqsubseteq i \\ y \sqsubseteq x \\ o \sqsubseteq y \\ \#x = \min(\#i, 1 + \#ack) \\ \#y \geq \min(\#x, \#req) \\ \#ack = \min(\#x, \#req + N - 1) \\ \#o \geq \#y \\ \#req = 1 + \#y \end{array}$

From the specification of $System(N)$ above, we can immediately see that the output is a prefix of the input: $o \sqsubseteq y \sqsubseteq x \sqsubseteq i$. Using the inequalities it can also be shown by some case analysis that the length of the output equals the length of the input. Together, this implies

$$o = i$$

for all input streams i . As expected, the system implements the identity relation.

The same result could have been obtained by first composing the three component state machines, and then deriving $o \sqsubseteq i$ and $\#o \geq \#i$; the number of verification conditions for the invariance and leadsto properties would have been much higher, however. For the composition of dataflow properties, history relations seem to be the more adequate abstraction level.

7 Conclusion

In this paper we showed how state-based and history-based specification and verification techniques for safety and liveness properties of distributed systems can be combined. State machine properties are expressed using a standard linear temporal logic; history properties are expressed as relations between input and output streams.

In a related technical report [2] we also allow composition at the level of state machines; properties proven for the combined system are shown to hold also for the black box composition of a system. That our system is compositional is due to the dataflow nature of our systems: Components cannot disable transitions of other components, thus the system is interference free. This is quite useful in practice, since it is often hard to find suitable history predicates for each component, although the complete system behavior can be succinctly specified in this way. State machine composition also helps to circumvent the mismatch between purely relational dataflow specifications and the operational intuition that was discovered by Brock and Ackermann [3].

Proofs for larger systems, especially for leadsto properties, are often quite complex. A solution might be to use verification diagrams along the lines of [4,11], which reduce temporal reasoning to simple first-order verification conditions. Since the number of verification conditions for concrete systems can be quite large, some kind of tool support is needed. As an experiment, the safety properties of the communication system example have been verified using the STeP [1] proof environment; currently, we are formalizing our approach in Isabelle/HOL [13].

Our specification and proof techniques are so far only suited for time independent systems. The extension of history-based specifications raises some interesting questions [5]. A straightforward solution might be to explicitly include “time ticks” in the message streams. Such time ticks can also be used to ensure progress of a state machine. But also without explicit time, progress is not restricted to the weak fairness condition of § 3.3. An alternative would

be to just demand that some transition is taken whenever at least one transition is persistently enabled; some classes of components, in particular *fair merge* components would then require additional oracle inputs.

Acknowledgments This report benefited from many stimulating discussions with Manfred Broy. We thank Katharina Spies for comments on a draft version of this report, and an anonymous referee for his very detailed remarks.

References

1. N. Bjørner, A. Browne, E. 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 *CAV'96. Lecture Notes in Computer Science 1102*, pages 415–418, 1996.
2. M. Breitling and J. Philipps. Black Box Views of State Machines. Technical Report TUM-I9916, Institut für Informatik, Technische Universität München, 1999.
3. J. D. Brock and W. B. Ackermann. Scenarios: A model of nondeterministic computation. In J. Diaz and I. Ramos, editors, *Lecture Notes in Computer Science 107*, pages 225–259, 1981.
4. I. A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. In *Lecture Notes in Computer Science 1026*, pages 484–498, 1995.
5. M. Broy. Functional specification of time sensitive communicating systems. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Models, Formalism, Correctness. Lecture Notes in Computer Science 430*, pages 153–179. Springer, 1990.
6. M. Broy. From states to histories. In *Engineering Theories of Software Construction*. NATO Science Series F, Marktoberdorf Summer School, 2000. To be published.
7. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus—Revised Version. Technical Report TUM-I9202-2, Institut für Informatik, Technische Universität München, 1993.
8. M. Broy, F. Huber, B. Paech, B. Rumpe, and K. Spies. Software and system modeling based on a unified formal semantics. In M. Broy and B. Rumpe, editors, *Requirements Targeting Software and Systems Engineering, International Workshop RTSE'97. Lecture Notes in Computer Science 1526*. Springer, 1998.
9. F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus—a tool for distributed systems specification. In *Proceedings FTRTFT'96 — Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science 1135*, 1996.
10. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages*, 6(3):872–923, May 1994.
11. Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.
12. L. C. Paulson. *Logic and Computation*. Cambridge University Press, 1987.
13. L. C. Paulson. *Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science 828*. Springer, 1994.

Distance Functions for Defaults in Reactive Systems

Sofia Guerra

Department of Computer Science
University College London
London WC1E 6BT, UK
fax: +44 (0)20 7387 1397
S.Guerra@cs.ucl.ac.uk

Abstract. Default reasoning has become an important topic in software engineering. In particular, defaults can be used to revise specifications, to enhance reusability of existing systems, and to allow a more economic description of systems. In this paper we develop a framework for default specifications of reactive systems.

We present a formalisation of non-monotonicity in temporal logic based on the notion of default institution. Default institutions were defined as an extension of institutions in order to allow partial reuse of existing modules. The semantics of defaults is given by a (generalised) distance between interpretations. In this way, by defining a pre-order between temporal morphisms and using temporal logic as a specification language, we get a way of handling defaults in specifications of reactive systems. We illustrate the developed formalism with an example in which a specification is reused, but where the new behaviour contradicts the initial specification. In this example, the initial specification is seen as a default to which exceptions are added.

1 Introduction

Although default reasoning first appeared as a field within artificial intelligence, it has become an important topic in software engineering. It concerns the reasoning based on assumptions held to be true unless there is specific evidence to the contrary. Several issues show the benefits of defaults in specifications. In particular, defaults can be used to revise specifications, they enhance reusability of existing systems, and they allow a more economic description of systems. Defaults can also be used to handle inconsistencies resulting from the combination of different perspectives and views of people developing a large software system, i.e. in the viewpoints framework [3].

Many important computer programs such as operating systems, network communication protocols, and air traffic control systems exhibit ongoing behaviour which is ideally non-terminating, and thus infinite, reflecting their continuously operating nature. These systems maintain an ongoing interaction with their environment, and intermediate outputs of the program can influence subsequent intermediate inputs of the program. Such systems are called *reactive*

systems. Following the seminal work by Pnueli [9], temporal logics have proved to be suitable for modelling the reactive aspects of systems. Therefore, developing a formalism for default reasoning based on temporal logic will allow for the description of reactive software systems where we can directly talk about defaults.

The work of Goguen and Burstall on institutions [4] has shown that many aspects of specifications in the large, namely the ability to put together small specifications to form the specification of a complex system, depend only on some properties of the underlying logic. Institutions formalise the notion of ‘a logical system’. In addition, they provide a way of ‘gluing’ together theories and hence a way of structuring specification. Although institutions provide a way of structuring specifications, the existing specifications can be enriched but not modified. *Default institutions* [10] were proposed as an extension to the notion of institution in order to enable partial re-use of specifications. They are a generic framework for the treatment of exceptions to a norm, based on a logic and a generalised distance between its interpretations. By using this generalisation of distances, they give semantics to the combination of default modules with more specific information that can override the defaults. In this way, in the same way the theory of algebraic specifications is parameterised by institutions, theory of structuring with exceptions will be parameterised by default institutions.

In this paper we develop a formalism for non-monotonicity in temporal logic based on the concept of default institution. In this framework, the semantics of a default with an exception is given by selecting the models of the exception that are as close as possible to the models of the default, according to the given notion of distance function. The temporal instantiation of default institutions allows for the development of structured default specifications of reactive systems.

Although several mechanisms have been proposed for handling defaults in common-sense reasoning, not so many have been proposed for specifications of reactive systems. The motivations and intended models of frameworks for non-monotonic temporal reasoning about software systems or common-sense examples are different. Hence, the systems intended for these different purposes have to be dissimilar. Non-monotonic reasoning is invisible and virtually non-existent in industry. Morgenstern [8] tries to understand why non-monotonic reasoning and industry are so far apart. The reasons given are related to the fact that research has been focussed almost exclusively on problems of common-sense reasoning, while industry is primarily concerned with problems which appear to have very little to do with common-sense reasoning. Although industry offers fertile ground for non-monotonic researchers, it remains uncharted territory for the non-monotonic community [8]. In the strategy for integrating non-monotonic reasoning in industry, researchers should familiarise themselves with problems in industry, select a set to which non-monotonic reasoning appears to be relevant, and focus on those problems in their research. In this paper we focus on problems of non-monotonicity in specifications, and the decisions made are influenced accordingly.

Most of the work done in temporal defaults has been developed with common-sense examples in mind. However, [1,6] propose an extension to the Object Specification Logic (OSL) [12] with defaults, to arrive at what they call Object Specification Logic with Defaults (OSD). OSD is based on temporal logic and the interpretations are organised in a pre-order, hence constituting a preferential model [7]. OSD is based on the temporal prioritisation originated from Shoham's chronological ignorance [11], where earlier defaults are implicitly given higher priority. This approach has some problems which arise from chronological minimisation. The authors themselves highlighted these difficulties, and dealt with them by adding specific defaults with different levels of priority. However, when specifying a system, if using these ideas we would have to know exactly which defaults to add, and which levels of priority to assign to those defaults. These obstacles were considered here when developing the distances between temporal interpretations, and they were directly solved through the use of this framework; the specifier does not have to consider these problems when using the framework developed here. Default institutions are more powerful than preferential models [5] and hence, if the right instantiation is chosen, they allow us to deal with these obstacles directly within the framework.

This paper is structured as follows. Section 2 briefly presents default institutions and the idea of handling defaults by distances between interpretations. Section 3 is the main part of the paper. It begins by presenting the temporal logic that will be used. A simple example is then described, which will illustrate the framework being developed. Finally, we present the formalisation of non-monotonicity in temporal logic that is used to handle defaults in the specification of reactive systems. We conclude in section 4 by summarising the main points and sketching further work.

2 The Meaning of but

The algebraic specification school proposes a strong construct where an existing specification can be enriched, but not modified. In [10] default institutions are proposed in order to allow partial reuse of existing specification modules. They extend the general notion of a logic, of *institutions* [4], by including a notion of distance between interpretations. The modification of a specification D with an exception E is denoted by D **but** E , representing that a default D can be overridden by more specific properties E . The semantics of D **but** E is given by selecting the models of the exception E that are as close as possible to the models of the default D according to the given notion of distance between interpretations.

In general, we want to compare interpretations that may be very different in nature. Therefore, we need a way to relate elements of different nature that play a similar role. This is already done for the framework of algebraic specifications through the use of morphisms between interpretations. The notion of distance is then generalised, and we compare pairs of interpretations linked by a morphism; distances are the particular case when there is only one morphism between each

pair of interpretations. Interpretation morphisms (i.e. pairs of interpretations linked by an indication about which elements play similar roles) are compared by a pre-order $\leq_\Sigma$ among morphisms (here, and in the following, Σ is a signature). This pre-order has to verify some constraints in order to capture the motivations for handling defaults. We explain some of the ideas underlying this pre-order and the notion of default institution. The formal definition of default institution and further explanation can be found in [10,5].

If m, n, m' , and n' are interpretations for a given signature Σ , and $h : m \rightarrow n$ and $h' : m' \rightarrow n'$ are interpretation morphisms, intuitively $h : m \rightarrow n \leq_\Sigma h' : m' \rightarrow n'$ means that m is ‘closer’ to n (according to h) than m' to n' (according to h'). The minimal morphisms for each of these orderings $\leq_\Sigma$ are called *agreements*. Note that identity morphisms should always be agreements, since what can be closer to any interpretation than itself? We want interpretations linked by a minimal morphism (an agreement) to behave similarly, since the fact that a morphism $h : m \rightarrow n$ is minimal represents that m is as close to n as possible. To guarantee that it is the case, we impose that if two interpretations m and n are linked by an interpretation morphism $h : m \rightarrow n$ then the properties of m are also properties of n , i.e. if $\mathcal{L}_\Sigma$ is the set of all formulae with signature Σ then $\{\phi \in \mathcal{L}_\Sigma \mid m \models \phi\} \subseteq \{\phi \in \mathcal{L}_\Sigma \mid n \models \phi\}$. This condition is called *weak abstractness* [10]. Intuitively, this means that our logic does not allow us to look at more details of the interpretations than the morphisms do. In addition, the usual condition of symmetry on distance functions is weakened towards *0-symmetry* [10]: if there is an agreement from m to n , then there is also one from n to m . These two conditions of 0-symmetry and weak abstractness together imply that two interpretations linked by an agreement satisfy exactly the same formulae. An extra restriction on the pre-order is the requirement that agreements should be transparent with respect to comparisons. This condition is called 0-equivalence and it means that composing a morphism with an agreement is equivalent (in the pre-order) to the initial morphism itself.

The semantics of the combination of a default D and an exception E , noted $D \text{ but } E$, is defined using the concept of (generalised) distance given by the pre-order on interpretation morphisms. An interpretation m is a model of $D \text{ but } E$ if m is the domain of a minimal morphism among the morphisms whose domain satisfies E and whose codomain satisfies D . Note that models of $D \text{ but } E$ always satisfy the exception E . To express this formally, some notation is explained:

- If E and D are sets of formulae, $Mor(E, D)$ is the class of morphisms whose domain satisfy E and whose codomain satisfy D .
- $Min(E, D)$ is the class of minimal morphisms of $Mor(E, D)$.

Formally, the models of the **but** are the following:

Definition 1 (Semantics of but). *Let E and D be sets of formulae over a signature Σ , and m an interpretation. Then m is a model of $D \text{ but } E$, written $m \models D \text{ but } E$, iff there is a morphism $h \in Min(E, D)$ such that $m = dom(h)$.*

When using this machinery within a particular logic, the main choices we have to make are on the morphisms between interpretations and the pre-order

between these morphisms, i.e. we have to decide what is the pre-order $\leq_{\Sigma}$. This is exactly what we do in the next section for the temporal case.

3 The Temporal Setting

The structure of temporal argument and temporal discourse has had an interest in different fields over the years, probably as a result of the temporal facet of human reasoning. In Computer Science, the use of temporal logic as a formalism for specifying and verifying correctness of computer programs was stated by Pnueli in [9], and it has been widely explored since this landmark paper.

In the remainder of this section we briefly describe linear propositional temporal logic, and we define morphisms between temporal interpretations and a pre-order on them. The pre-order presented here is tailored by the applications at hand, namely specifications with defaults.

3.1 Propositional Temporal Logic

The language of propositional temporal logic is based on the language of propositional logic, but various operators or ‘modalities’ are provided to reason about the change of truth values of assertions over time. Examples of common temporal operators include **G** (always in the future), **F** (eventually), **W** (unless), **X** (next) and **U** (until). The propositional temporal language is defined starting from a set of proposition letters Σ (a signature). The set $\mathcal{L}_{\Sigma}$ of propositional temporal formulae is the least set such that:

- each $p \in \Sigma$ is a formula ($\Sigma \subset \mathcal{L}_{\Sigma}$);
- if A and B are in $\mathcal{L}_{\Sigma}$, then $\neg A$ and $(A \wedge B)$ are in $\mathcal{L}_{\Sigma}$;
- if A and B are in $\mathcal{L}_{\Sigma}$, then $\mathbf{X}A$ and $A\mathbf{U}B$ are in $\mathcal{L}_{\Sigma}$.

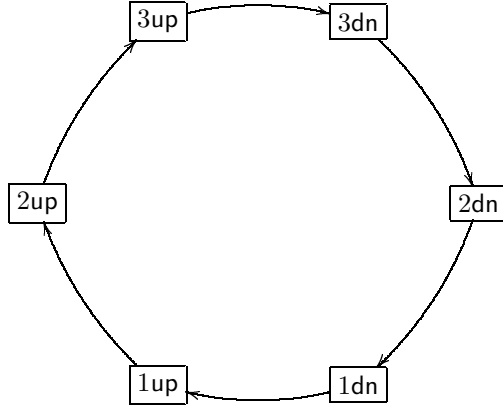
The semantics of the temporal language is given in terms of *frames*; a frame is an ordered pair $\mathcal{F} = (W, R)$, where W is the set of points in time ordered by a binary relation R of precedence between them: sRt is read ‘ t is after s ’. In the sequel we will use the frame $(\mathbb{N}, <)$. An *interpretation* is a triple $\mathcal{M} = (\mathbb{N}, <, V)$, where V is a function assigning to every timepoint t (every natural number) a set of proposition letters $V(t) \subseteq \Sigma$, namely the set of proposition letters that are true at the timepoint t . The semantics of a formula A over an interpretation m at timepoint t , written $m \Vdash_t A$, is recursively defined as follows:

$m \Vdash_t p, p \in \Sigma$	iff	$p \in V(t)$;
$m \Vdash_t \neg A$	iff	it is not the case that $m \Vdash_t A$;
$m \Vdash_t A \wedge B$	iff	$m \Vdash_t A$ and $m \Vdash_t B$;
$m \Vdash_t \mathbf{X}A$	iff	$m \Vdash_{t+1} A$;
$m \Vdash_t A\mathbf{U}B$	iff	there exists $t' \in \mathbb{N}$ with $t \leq t'$ and $m \Vdash_{t'} B$ and for every $t'' \in \mathbb{N}$, if $t \leq t'' < t'$ then $m \Vdash_{t''} A$.

We say that a formula A is *valid on the interpretation m* iff $m \models_0 A$. The semantics of the other temporal operators can be derived from these.

Now that we have defined the propositional temporal logic, the problem is how to define morphisms between interpretations and a pre-order on the morphisms in such a way that the defaults in specifications of reactive systems behave in the expected way. To motivate these definitions let us look at an example.

3.2 An Example: A Ferris Wheel



Consider a Ferris Wheel with six chairs, where each chair can be in one of six positions. As the relative position of the chairs does not change, from the position of one of the chairs we know the position of all the other five. Hence, we choose one of the chairs and from now on we always talk about the same chair. The chair can be in one of three levels 1, 2 or 3, and it moves always in the same direction (clockwise). The name of each position indicates the level of the chair and whether it is going up or down: e.g. 3up indicates that it is at level 3 and it is going up. Following this convention, the names of the positions of the chair are: 1up, 2up, 3up, 3down, 2down and 1down. The chair goes from 1up to 2up, and from this to 3up, then it goes to 3down and starts going down to 2down and 1down and then to 1up again, and it never stops. A specification of this Ferris Wheel can be seen below. The first formula states the fact that the chair is always in one of the six positions and it is not in more than one at each moment. The other formulae describe the movement of the chair.

$$\begin{aligned}
 & \mathbf{G}((1\text{up} \vee 2\text{up} \vee 3\text{up} \vee 3\text{down} \vee 2\text{down} \vee 1\text{down}) \wedge \\
 & \quad \neg(1\text{up} \wedge 2\text{up}) \wedge \neg(1\text{up} \wedge 3\text{up}) \wedge \neg(1\text{up} \wedge 3\text{down}) \wedge \dots) \\
 & \mathbf{G}(1\text{up} \rightarrow \mathbf{X}2\text{up}) \qquad \qquad \qquad \mathbf{G}(2\text{up} \rightarrow \mathbf{X}3\text{up}) \\
 & \mathbf{G}(3\text{up} \rightarrow \mathbf{X}3\text{down}) \qquad \qquad \mathbf{G}(3\text{down} \rightarrow \mathbf{X}2\text{down}) \\
 & \mathbf{G}(2\text{down} \rightarrow \mathbf{X}1\text{down}) \qquad \qquad \mathbf{G}(1\text{down} \rightarrow \mathbf{X}1\text{up})
 \end{aligned}$$

Suppose now that we want to add an emergency lever that makes the chair reach level 1 as quickly as possible. If the chair is in one of the down positions,

then the fastest way of getting to level 1 is to keep going clockwise. However, if the chair is in one of the **up** positions then the best thing is to reverse the direction, and move counter-clockwise. We suppose that the **lever** stays pushed until the chair gets to level 1 when it stops, and then the **lever** is released. The action of pushing the lever is called **push**, and **lever** represents the fact that the lever is pushed. The formulae below represent what we want to add to the specification.

$$\begin{aligned} & \mathbf{G}(\text{push} \rightarrow \text{lever}) & \mathbf{G}(((3\text{down} \vee 2\text{down}) \wedge \text{lever}) \rightarrow \mathbf{X}\text{lever}) \\ & \mathbf{G}((\text{lever} \wedge 3\text{up}) \rightarrow \mathbf{X}(2\text{up} \wedge \text{lever})) & \mathbf{G}((\text{lever} \wedge 1\text{down}) \rightarrow \mathbf{X}(1\text{down} \wedge \neg\text{lever})) \\ & \mathbf{G}((\text{lever} \wedge 2\text{up}) \rightarrow \mathbf{X}(1\text{up} \wedge \text{lever})) & \mathbf{G}((\text{lever} \wedge 1\text{up}) \rightarrow \mathbf{X}(1\text{up} \wedge \neg\text{lever})) \end{aligned}$$

Instead of rewriting the specification from the beginning, and considering explicitly whether the **lever** has been pushed or not, we want to add these formulae as exceptions to the specification. If D is the conjunction of the formulae in the specification of the Ferris Wheel, and E the conjunction of the formulae that describe the **lever**, then the models of the Ferris Wheel with the emergency lever would be the models of the state constraint that satisfy D **but** E . However, in order to give semantics to **but** we have to know what are the morphisms between interpretations and how they are ordered, i.e. what is the pre-order $\leq_\Sigma$.

3.3 The Pre-order

When defining the pre-order, the first thing to note is that not every formula should be overridable. In particular, state constraints have to be rigid, otherwise we could satisfy the exceptions by allowing the chair to be in more than one position at the same time. These formulae should not be invalidated and we see them as axioms that cannot be overridden. The details of how to do that are omitted, but the basic idea is to consider hierarchic specifications. A hierarchic specification consists of two parts: a set of axioms that correspond to the facts that must hold, plus a part of defaults. The defaults are organised by priority levels, and they express properties that are likely to be true but can be overridden by other information, either by the axioms or by defaults with a higher priority. The models of a hierarchic specification are the models of the axioms that satisfy as much of the defaults as possible, taking into consideration the priority among them. The pre-order of morphisms is generalised towards a pre-order of families of morphisms. More details can be found in [5], but here it would correspond to considering for minimisation only the models of these axioms. In our example, the state constraint is expressed by the first formula of the specification of the Ferris Wheel.

If we think about how to order the morphisms, a first possibility which follows from the ideas in [10] would be the following: if $h : m \rightarrow n$ and $h' : m' \rightarrow n'$ are morphisms between temporal interpretations for a signature Σ , $h : m \rightarrow n \leq_\Sigma h' : m' \rightarrow n'$ iff for all $t \in \mathbb{N}$,

$$\text{if } m \text{ disagrees with } n \text{ at } t \quad \text{then} \quad m' \text{ disagrees with } n' \text{ at } t,$$

where by *disagree* we mean that they do not satisfy the same propositional symbols at that instant. This definition has two immediate problems. Firstly, if m disagrees with n at some point t then what happens afterwards might be meaningless. As an example consider the Ferris Wheel; let m be a model of the exceptions (the formulae that describe the lever) and n a model of the initial specification. Suppose that at some timepoint t , m was at 2up, the lever was not pushed and, as if by magic, in $t + 1$ it was at 2down. If n agreed with m at t (i.e. m was at 2up as well) then obviously they disagree at $t + 1$. Moreover, even if m behaves as expected after t it will disagree with n , and these disagreements are meaningless because they are the result of what happened at t . This suggests that we want to see, at some point t , if m were the interpretation n , whether it would behave in the same way. Therefore, morphisms between interpretations are functions that choose, for each timepoint of the domain interpretation, a timepoint in the codomain that satisfies exactly the same symbols as the domain at that point. This represents the idea of supposing that m was n ; and then we compare the satisfiability of the propositional symbols at the next points. Following this idea, morphisms between interpretations are monotonic functions $h : \mathbb{N} \rightarrow \mathbb{N}$ such that for every timepoint t and every symbol p , $m \models_t p$ iff $n \models_{h(t)} p$, and we check whether $m \models_{t+1} p$ iff $n \models_{h(t)+1} p$ in order to see how much m and n differ.

Another problem with that definition of the pre-order is that it blocks the occurrence of the actions that change the behaviour of the initial specification, like the action of pushing the lever. With that definition, the minimal models of the Ferris Wheel with the lever would be the ones where the lever is never pushed. This is highly undesirable, since we want to be able to push the lever whenever we feel like (whenever there is an emergency). This suggests that we only want to compare interpretations that have the same occurrences of actions at the same timepoints. However, this causes some problems when comparing interpretations with actions that have enabling conditions, as explained below. Nevertheless, we have to treat actions differently from the attributes in the minimisation process. Hence, we split each signature Σ in two disjoint sets: one for the actions Σ_{Act} , that cannot be minimised, and another set Σ_{Att} for the attributes. The language considered is built from the signature as explained before.

Definition 2. A signature is a pair of disjoint sets $\Sigma = (\Sigma_{Act}, \Sigma_{Att})$, where Σ_{Act} is the set of actions and Σ_{Att} is the set of attributes (or observations).

We impose that comparable interpretations agree at the first instant, i.e. we suppose that they satisfy the same symbols at the instant 0. The reason for this restriction is that when we have a system we know the initial state, and we are interested in the way it evolves; systems with different initial states will obviously be different and the meaning of the comparison is unclear.

We can now say what are the morphisms between interpretations. Firstly we introduce some notation. Let $h : \mathbb{N} \not\rightarrow \mathbb{N}$ be a partial function and $t \in \mathbb{N}$ be a natural number. Then

- $h(t) \uparrow$ means that h is undefined at t ;
- $h(t) \downarrow$ means that h is defined at t ;

- $\bar{h}(t) = \max\{h(t') \in \mathbb{N} : t' \leq t \text{ and } h(t') \downarrow\}$ is the image of the greatest timepoint less or equal than t at which the function is defined.

The idea of the morphisms is to relate two temporal interpretations m and n that have the same initial state (they satisfy exactly the same symbols at 0), in a way that, for each instant t_1 , a timepoint t_2 is chosen in a monotonic way, such that the interpretation domain m at t_1 satisfies exactly the same symbols as the interpretation codomain n at t_2 . This choice is partial to allow cases where such a t_2 does not exist. These conditions are formally expressed by the following definition:

Definition 3. Let Σ be a signature and m and n be two temporal interpretations over Σ . A morphism $h : m \rightarrow n$ is a partial function $h : \mathbb{N} \rightharpoonup \mathbb{N}$ such that:

- $h(0) = 0$;
- for all $t_1, t_2 \in \mathbb{N}$, if $t_1 < t_2$, $h(t_1) \downarrow$ and $h(t_2) \downarrow$ then $h(t_1) < h(t_2)$;
- for all $t \in \mathbb{N}$, if $h(t) \downarrow$ then for all $p \in \Sigma$, $m \models_t p$ iff $n \models_{h(t)} p$.

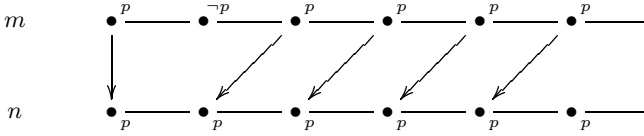
Let us look at an example of a morphism between two temporal interpretations.

Example 1. Let $\Sigma = (\Sigma_{Act}, \Sigma_{Att})$ be a signature where $\Sigma_{Act} = \emptyset$ and $\Sigma_{Att} = \{p\}$. Consider the expression $\mathbf{G}p \text{ but } \mathbf{X}\neg p$.

Let m be a Σ -interpretation such that $m \models_t p$, for all $t \neq 1$, and n the Σ -interpretation such that $n \models_t p$, for all $t \in \mathbb{N}$. Then $h : m \rightarrow n$ defined as follows is a temporal morphism:

- $h(0) = 0$;
- $h(1) \uparrow$; and
- $h(t) = t - 1$, for every $t > 1$.

This morphism can be represented by the diagram below.



This morphism belongs to the class of morphisms whose domain satisfies $\mathbf{X}\neg p$ and whose codomain satisfies $\mathbf{G}p$, i.e. $h \in \text{Mor}(\mathbf{X}\neg p, \mathbf{G}p)$. Note that at timepoint 1 the function h has to be undefined, as there is no point in n that satisfies $\neg p$. However, the choices for the other timepoints greater than 1 are arbitrary, as far as they are monotonic. We chose this morphism among several other possibilities as this will be minimal in $\text{Mor}(\mathbf{X}\neg p, \mathbf{G}p)$ when we define the pre-order on interpretation morphisms. In fact, this morphism will also belong to the class $\text{Min}(\mathbf{X}\neg p, \mathbf{G}p)$, and hence $m \models \mathbf{G}p \text{ but } \mathbf{X}\neg p$. We present the proof that this is a minimal morphism in the appendix.

Now that we have defined morphisms between temporal interpretations, we can return to considering the notion of ‘closeness’ in order to get the pre-order $\leq_\Sigma$.

Concordance on Initial States Although we cannot define the pre-order yet, taking into account the previous considerations, we know that we only want to compare interpretations with the same initial state. Hence, $h : m \rightarrow n \leq_{\Sigma} h' : m' \rightarrow n'$ implies:

$$1. \quad \forall p \in \Sigma. \quad m \Vdash_0 p \text{ iff } m' \Vdash_0 p.$$

Minimisation of Discrepancies Suppose $h : m \rightarrow n$ is a morphism between two temporal interpretations. If m was ‘near’ n , it would behave, at every time-point, in the same way as n , if n was at a similar state. Discrepancies are the cases when this does not happen.

Definition 4. Let $\Sigma = (\Sigma_{Act}, \Sigma_{Att})$ be a signature, $h : m \rightarrow n$ be an interpretation morphism and $t \in \mathbb{N}$ a timepoint. We say that there is a discrepancy at time t between m and n according to h , written $m \overset{h}{\leftrightarrow}_t n$, if $h(t)$ is defined and there is an attribute $r \in \Sigma_{Att}$ such that

$$(m \Vdash_{t+1} r \text{ and } n \nVdash_{h(t)+1} r) \quad \text{or} \quad (m \nVdash_{t+1} r \text{ and } n \Vdash_{h(t)+1} r).$$

This notion of discrepancy expresses the fact that, if the interpretation n was at the state that m is at some instant t , it would behave in a different way. This depends on the morphism h , that chooses from n an instant to compare with m at the instant t : by definition of morphism, if $h : m \rightarrow n$ is a morphism, then m satisfies at t exactly the same symbols as n at $h(t)$. There is a discrepancy at t if they progress in different ways: at the following instants, $t + 1$ for m and $h(t) + 1$ for n , they satisfy different attributes. This notion of discrepancy is going to be used to define the pre-order: two interpretations are ‘nearer’ if they have less discrepancies. It is the justification for unexpected differences between morphisms.

If $h : m \rightarrow n$ and $h' : m' \rightarrow n'$ are morphisms, we want to know if m is closer to n (according to h) than m' to n' (according to h'). The main idea is to minimise the discrepancies. We could say that if $h : m \rightarrow n \leq_{\Sigma} h' : m' \rightarrow n'$ and there is a discrepancy at t between m and n then there is also one between m' and n' at the same instant: $\forall t \in \mathbb{N}$, if $m \overset{h}{\leftrightarrow}_t n$ then $m' \overset{h'}{\leftrightarrow}_t n'$.

The problem with this definition is that if m' did not behave as expected before an instant t that allows it to agree with n' at t , but m did behave as expected until t , which makes it disagree with n at t , then we could not conclude the relation we wanted. Thinking again about the Ferris Wheel example, suppose $h : m \rightarrow n$ and $h' : m' \rightarrow n'$ are morphisms, m and m' satisfy **lever** at the same instants and they agree at 0. Suppose t is a timepoint where m is at **2up** and the lever is pushed at t , and this is the first time the lever is pushed. We can also suppose that before t , m satisfies all the formulae of the initial specification (without the lever). In these conditions there is a discrepancy between m and n at t . However, if m' is at a **down** point at t (because it did not follow the default formulae at some previous instant) then there is no discrepancy at t between m' and n' , and we would not prefer the morphism h to the morphism h' . This

is obviously undesirable, as the interpretation m had always behaved as desired but m' had not. To solve this we impose that if $h : m \rightarrow n \leq_{\Sigma} h' : m' \rightarrow n'$ and if there is a discrepancy between m and n at t that does not exist between m' and n' then there must be a reason for this: either because h' is undefined at t or because at a previous instant there was a discrepancy in h' that did not exist in h . This gives rise to the following condition that we add to the previous we already had: if $h : m \rightarrow n \leq_{\Sigma} h' : m' \rightarrow n'$ then

$$2. \forall t_1 \in \mathbb{N}. ((m \xleftrightarrow{h}_{t_1} n \wedge m' \not\xleftrightarrow{h'}_{t_1} n' \wedge h'(t_1) \downarrow) \Rightarrow (\exists t_2 < t_1. (m \not\xleftrightarrow{h}_{t_2} n \wedge h(t_2) \downarrow \wedge (m' \xleftrightarrow{h'}_{t_2} n' \vee h'(t_2) \uparrow))))).$$

Synchronisation of Actions We said that the occurrence of actions should not be blocked in the minimisation process. One way of doing it would be to restrict the comparisons between morphisms whose domain satisfies the same actions at the same instants. This does not cause any problem when the actions do not have enabling conditions, like in the Ferris Wheel example. However, this is not the case when we consider examples where there are enabling conditions on the defaults that are not overridden by exceptions. Since we would not block the occurrence of actions in the domains of morphisms, they would be able to occur even if they did not satisfy the enabling conditions of the defaults. The following example illustrates this problem.

Example 2. Consider a library with books. Each book of the library can be available to be taken by a user. The attribute **available** means that a book is available to be taken by a user. A book can be **taken** when it is **available**, which makes it not **available**. Similarly, a book can be **returned** when not **available**, which causes it to be **available** again.

Suppose now that we want to consider reserved books, having the added property that they may not be taken out of the library. In this case, when the book is **suspended** it stops being **available** until it is **resumed** again.

The signature for this specification is $\Sigma = (\Sigma_{Act}, \Sigma_{Att})$, where $\Sigma_{Act} = \{\mathbf{taken}, \mathbf{suspended}, \mathbf{returned}, \mathbf{resumed}\}$ and $\Sigma_{Att} = \{\mathbf{available}\}$. Hence, the set of defaults D of this specification will include:

$$\begin{aligned} & \mathbf{G}(\mathbf{taken} \rightarrow \mathbf{available}) \\ & \mathbf{G}(\mathbf{taken} \rightarrow \mathbf{X}\neg\mathbf{available}) \\ & \mathbf{G}(\mathbf{available} \wedge \neg\mathbf{taken} \rightarrow \mathbf{X}\mathbf{available}) \end{aligned}$$

and the following will be included in the exceptions that describe the reserved book:

$$\begin{aligned} & \mathbf{G}(\mathbf{suspended} \rightarrow \mathbf{available}) \\ & \mathbf{G}(\mathbf{suspended} \rightarrow \mathbf{X}\neg\mathbf{available}). \end{aligned}$$

Hence, the occurrence of the action `suspended` should not be restricted. However, `taken` should only occur when `available` is true; `available` should be seen as an enabling condition for `taken`. If, as suggested earlier, two morphisms are only compared when their interpretation domain have the same occurrences of actions, we do not have a way of preferring interpretations in which `taken` occurs only when `available` is true.

As we showed in the previous example, actions that are always enabled, i.e. actions that do not have enabling conditions, should not be blocked, even if they are or cause inconsistencies with the defaults. However, if the defaults restrict their occurrence, these conditions should be taken into consideration. In order to do that, we impose the following: in the cases where there are no enabling conditions for the occurrence of actions, we will only compare morphisms if their domains have the same occurrences at the same timepoints. If the domain of two morphisms h and h' do not satisfy the same occurrences of actions at the same timepoints, the morphisms are comparable only if there was a reason for the differences between these occurrences, either by a discrepancy in a previous timepoint, or by the map being undefined at the point in which the action occurs. Hence, we can add to the previous conditions the following two:

3. $\forall a \in \Sigma_{Act}. \forall t_1 \in \mathbb{N}. (m \Vdash_{t_1} a \wedge m' \nVdash_{t_1} a) \Rightarrow$
 $((\exists t_2 < t_1. (m' \xrightarrow{h'}_{t_2} n' \wedge h'(t_2) \downarrow \wedge (m \xrightarrow{h}_{t_2} n \vee h(t_2) \uparrow))) \vee (h(t_1) \uparrow \wedge h'(t_1) \downarrow))$
4. $\forall a \in \Sigma_{Act}. \forall t_1 \in \mathbb{N}. (m' \Vdash_{t_1} a \wedge m \nVdash_{t_1} a) \Rightarrow$
 $((\exists t_2 < t_1. (m \xrightarrow{h}_{t_2} n \wedge h(t_2) \downarrow \wedge (m' \xrightarrow{h'}_{t_2} n' \vee h'(t_2) \uparrow))) \vee (h(t_1) \downarrow \wedge h'(t_1) \uparrow))$

Surjectivity and Weak Abstractness To define the pre-order there are two further conditions that we need to impose. We want minimal morphisms to be as defined as possible: a completely undefined function does not have any discrepancy. Hence, in the same way as we did with the occurrence of actions in conditions 3 and 4, if $h \leq_{\Sigma} h'$ and there is a timepoint t_1 such that $h(t_1) \uparrow$ and $h'(t_1) \downarrow$, then there was a reason for it, namely there was a timepoint t_2 before t_1 at which there was a discrepancy that did not occur in h , or at that point $h(t_2)$ was defined and $h'(t_2)$ was undefined. This corresponds to condition 5 of definition 5 below.

The last condition of the pre-order ensures that the morphisms are as surjective as possible; this has as a result that identities are minimal, and that any minimal morphism is surjective. In this way the condition of weak abstractness is verified. If we allow non-surjective morphisms to be minimal, then we could have two interpretations linked by an agreement that would not satisfy the same formulae. In the points of the codomain interpretation that were not mapped by any point of the domain, the interpretation could satisfy different propositional symbols that would result in the two interpretations not satisfying the same formulae. The formal definition of the pre-order is the following:

Definition 5. Let $\Sigma = (\Sigma_{Act}, \Sigma_{Att})$ be a signature, and $h : m \rightarrow n$ and $h' : m' \rightarrow n'$ be interpretation morphisms. We say that $h : m \rightarrow n \leq_{\Sigma} h' : m' \rightarrow n'$ iff:

1. $\forall p \in \Sigma. m \Vdash_0 p$ iff $m' \Vdash_0 p$;
2. $\forall t_1 \in \mathbb{N}. ((m \xleftrightarrow{h}_{t_1} n \wedge m' \not\xleftrightarrow{h'}_{t_1} n' \wedge h'(t_1) \downarrow) \Rightarrow (\exists t_2 < t_1. (m \not\xleftrightarrow{h}_{t_2} n \wedge h(t_2) \downarrow \wedge (m' \xleftrightarrow{h'}_{t_2} n' \vee h'(t_2) \uparrow))))$
3. $\forall a \in \Sigma_{Act}. \forall t_1 \in \mathbb{N}. (m \Vdash_{t_1} a \wedge m' \not\Vdash_{t_1} a) \Rightarrow ((\exists t_2 < t_1. (m' \not\xleftrightarrow{h'}_{t_2} n' \wedge h'(t_2) \downarrow \wedge (m \xleftrightarrow{h}_{t_2} n \vee h(t_2) \uparrow))) \vee (h(t_1) \uparrow \wedge h'(t_1) \downarrow))$
4. $\forall a \in \Sigma_{Act}. \forall t_1 \in \mathbb{N}. (m' \Vdash_{t_1} a \wedge m \not\Vdash_{t_1} a) \Rightarrow ((\exists t_2 < t_1. (m \not\xleftrightarrow{h}_{t_2} n \wedge h(t_2) \downarrow \wedge (m' \xleftrightarrow{h'}_{t_2} n' \vee h'(t_2) \uparrow))) \vee (h(t_1) \downarrow \wedge h'(t_1) \uparrow))$
5. $\forall t_1 \in \mathbb{N}. ((h(t_1) \uparrow \wedge h'(t_1) \downarrow) \Rightarrow (\exists t_2 < t_1. (m \not\xleftrightarrow{h}_{t_2} n \wedge h(t_2) \downarrow \wedge (m' \xleftrightarrow{h'}_{t_2} n' \vee h'(t_2) \uparrow))))$
6. $(\forall t \in \mathbb{N}. (m \xleftrightarrow{h}_t n$ iff $m' \xleftrightarrow{h'}_t n') \wedge (h(t) \downarrow$ iff $h'(t) \downarrow)) \Rightarrow (\forall t \in \mathbb{N}. ((\exists t_1 \in \mathbb{N}. \overline{h}(t-1) < t_1 < h(t) \wedge \forall p \in \Sigma. (m \Vdash_t p$ iff $n \Vdash_{t_1} p)) \Rightarrow (\exists t_2 \in \mathbb{N}. \overline{h'}(t-1) < t_2 < h'(t) \wedge \forall p \in \Sigma. (m' \Vdash_t p$ iff $n' \Vdash_{t_2} p))))$.

To sum up, we only compare morphisms if the domains have the same initial state (condition 1). The minimisation of the discrepancies is expressed by condition 2. This resembles chronological minimisation [11], where defaults in earlier instants have higher priority than the ones occurring later. However, some of the problems with chronological minimisation do not occur here, as for example it would postpone the occurrence of the actions for ever. Conditions 3 and 4 make possible comparisons of interpretations only if the same actions occur at the same timepoints (unless there was a reason). The symmetry of conditions 3 and 4 is a way of avoiding blockage of the occurrence of actions. Moreover, we prefer morphisms ‘more defined’ and ‘more surjective’ (conditions 5 and 6 respectively). This relation is in fact a pre-order and it verifies the conditions presented in the previous section. In particular, the conditions 0-symmetry and 0-equivalence result from the fact that a morphism h is minimal in $\leq_{\Sigma}$ iff h is an identity [5].

Going back to our example, let $\Sigma = \{\Sigma_{Act}, \Sigma_{Att}\}$ be the signature with $\Sigma_{Act} = \{\text{push}\}$ and $\Sigma_{Att} = \{3\text{down}, 2\text{down}, 1\text{down}, 1\text{up}, 2\text{up}, 3\text{up}, \text{lever}\}$. We have that, if m is a model of the state constraint, and $m \Vdash D$ but E , then m is one of the desired models: if the chair is at 3down, 2down, or at 3up or 2up but the emergency lever has not been pushed, then it behaves as before adding the lever; if the chair is at 1down or 1up and the lever has been pushed it stops; if it is at 2up or 3up and the lever has been pushed, then it changes direction, as specified in the formulae we added. This framework can be extended to the general case with several defaults with an arbitrary precedence between them [10,5].

4 Concluding Remarks

In this paper we present a framework for specifications with exceptions for software systems that evolve over time. This framework is based on the notion of default institution [10], which allows partial re-use of specification modules. The semantics is parameterised by a notion of distance between interpretations.

We propose a temporal instantiation of this framework, which consists of the usual propositional linear temporal logic, but where the notion of morphism between interpretations is changed, and it is extended with a pre-order between these morphisms. This notion of ordering between morphisms of interpretations is used in order to deal with non-monotonicity in specifications. Hence, by describing specifications of reactive systems through the use of temporal logic, descriptions can include defaults and the temporal framework provides semantics for those specifications. We illustrate the developed formalism with an example where a specification is reused but where the new behaviour contradicts the initial specification. In this example the initial specification is seen as a default to which exceptions are added, avoiding the necessity of writing the whole specification from the beginning. This framework handles temporal default specifications in a way that it solves some of the problems of previous work in the area [1,6].

Further work is needed in order to study the applicability of this framework. Firstly, this could be generalised towards a multi-sorted first-order temporal logic or to other more powerful languages. Moreover, it is necessary to study the scalability of this formalism, and the development of algorithms to compute the models of but in a way similar to the work done for first-order logic [10].

Acknowledgements

Thanks to Pierre-Yves Schobbens and Mark Ryan for useful discussions, and to Anthony Finkelstein and Anthony Hunter for reading previous drafts of this paper. The author acknowledges financial support from the European Union (RENOIR), and from PRAXIS XXI and Fundação Calouste Gulbenkian in Portugal.

References

1. Stefan Brass, Udo Lipeck, and Pedro Resende. Specification of object behaviour with defaults. In Udo Lipeck and Gerhard Koschorreck, editors, *Proceedings of the International Workshop on Information Systems: Correctness and Reusability, ISCORE-93*, pages 155–177, 1993.
2. Dov Gabbay, C. J. Hogger, and J. A. Robinson, editors. *Nonmonotonic Reasoning and Uncertain Reasoning*, volume 3 of *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford: Clarendon Press, 1994.
3. Anthony Finkelstein and Ian Sommerville. The viewpoints FAQ. *Software Engineering Journal*, 11(1):2–4, 1996.
4. Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.

5. Sofia Guerra. *Defaults in the Specification of Reactive Systems*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, 1999.
6. Udo W. Lipeck and Stefan Brass. Object-oriented system specification using defaults. In K. V. Luck and H. Marburger, editors, *Management and Processing Complex Data Structures, Proceedings 3rd Workshop on Information Systems and Artificial Intelligence*, volume LNCS 777, pages 22–43, Berlin, 1994. Springer Verlag.
7. David Makinson. General patterns in non-monotonic reasoning. In Gabbay et al. [2], chapter 2, pages 35–110.
8. Leora Morgenstern. Inheritance comes of age: Applying nonmonotonic techniques to problems in industry. *Artificial Intelligence*, 103:237–271, 1998.
9. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.
10. Pierre-Yves Schobbens. *Exceptions in Algebraic Specifications*. PhD thesis, Université Catholique de Louvain, Faculté des Sciences Appliquées, 1992.
11. Yoav Shoham. Chronological ignorance: Experiments in nonmonotonic temporal reasoning. *Artificial Intelligence*, 36:279–331, 1988.
12. A. Sernadas, Cristina Sernadas, and J. F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

A Example 1 (Proof)

Firstly suppose that $h' : m' \rightarrow n'$ is a morphism in $Mor(\mathbf{X}\neg p, \mathbf{G}p)$ and that $h' \leq_{\Sigma} h$. We show that $h \leq_{\Sigma} h'$ by checking the conditions of the pre-order:

1. Condition 1, 2 and 3 are true since by hypothesis $h' \leq_{\Sigma} h$.
2. Condition 4 is true because there is no timepoint t such that $m \xleftrightarrow{h}_t n$ and $m' \not\xleftrightarrow{h'}_t n'$: $m \xleftrightarrow{h}_t n$ iff $t = 0$, which implies $m' \xleftrightarrow{h'}_t n'$.
3. $\{t : h(t) \uparrow\} = \{1\} \subseteq \{t : h'(t) \uparrow\}$, and then condition 5 is also verified.
4. Condition 6 is true because for all $t \in \mathbb{N}$, there is no t^* such that $\bar{h}(t-1) < t^* < h(t)$ and $\forall p \in \Sigma. m \Vdash_t p$ iff $n \Vdash_{t^*} p$.

In the other direction, suppose that m is not in the conditions described and let $h : m \rightarrow n$ be a morphism in $Min(\mathbf{X}\neg p, \mathbf{G}p)$ whose domain is m .

- $m \not\Vdash_0 p$ is impossible since h is a morphism and since $n \Vdash \mathbf{G}p$, we have that $n \Vdash_0 p$ and $m \Vdash_0 p$.
- $m \Vdash_1 p$ is impossible since m is a model of $\mathbf{X}\neg p$.
- $\exists t^* > 1$ with $m \Vdash_{t^*} p$ (which implies $h(t^*) \uparrow$). In this condition we prove that h is not minimal, contradicting the hypothesis. Let $h' : m' \rightarrow n'$ be the morphism such that

- $n' \Vdash_t p$ for all $t \in \mathbb{N}$;
- $m' \Vdash_t p$ iff $p \in \mathbb{N} - \{1\}$; and
- $h'(0) = 0$, $h'(1) \uparrow$, and $h'(t) = t - 1$ for $t > 1$.

As we have seen, $h' \leq_{\Sigma} h$. To see that this relation is strict, i.e. that $h \not\leq_{\Sigma} h'$ we note that

- if $h'(t^* - 1) \downarrow$, then $m \xleftrightarrow{h}_{t^*-1} n$ and $m' \not\xleftrightarrow{h'}_{t^*-1} n'$. Moreover, $\forall t < t^* - 1$ if $m \xleftrightarrow{h}_t n$ and $h(t) \downarrow$ then $m' \not\xleftrightarrow{h'}_t n'$ and $h'(t) \downarrow$. Hence, it fails condition 4.
- In the case that $h'(t^* - 1) \uparrow$, it fails condition 5.

Generalizing the Modal and Temporal Logic of Linear Time

Bernhard Heinemann

Fachbereich Informatik, FernUniversität Hagen
D-58084 Hagen, Germany
phone: ++49-2331-987-2714
fax: ++49-2331-987-319
`bernhard.heinemann@fernuni-hagen.de`

Abstract. In the present paper we generalize two fundamental systems modelling the flow of time: the modal logic S4.3 and propositional linear time temporal logic. We allow to consider a whole set of states instead of only a single one at every time. Moreover, we assume that these sets increase in the course of time. Thus we get a basic formalism expressing a distinguished dynamic aspect of sets, growing. Our main results include *completeness* of the proposed axiomatizations and *decidability* of the set of all formally provable formulas.

1 Introduction

Subsequently we are concerned with sets increasing in the course of time. This is a very general approach for the moment: frameworks of this kind actually occur in many fields of computer science. Maybe the reader will think of growing geometric objects first. It is in fact a widely attacked current problem of research to get an adequate specification language in order to reason about such objects formally.

Another example stems from the context of multi-agent systems. The set of states representing the *knowledge* of an agent involved in such a system changes during the system is running. In particular cases, if the agents share a *common clock* and *do not learn* while running, it actually increases; see [8]. In fact, the language we introduce below originates from this knowledge-theoretic context, and a corresponding *knowledge operator* is retained in it (which, however, serves another purpose now: quantifying inside sets).

How is it possible to treat increasing sets formally? — The approach we take is a *logical* one: spaces which consist of the sets occurring in the course of time represent the semantical domains of an appropriate language. Our aim is to provide a characterization of the *validities* then, i.e., the formulas holding under all such circumstances; moreover, we are interested in the basic algorithmic properties of the arising logical systems.

How may such systems look like? — This is the point where one has to make one's choice. Our starting point is a certain *modal* setting that has been introduced a few years ago, see [3]. It is aimed at *topological reasoning*¹ and captures *shrinking* of sets at least. Thus it offers access to our topic.² So, for convenience of the reader, we mention the very basic features of the system developed in [3]. In the simplest case two modalities are involved therein: one, designated K , which quantifies 'horizontally' over the elements of a set, and another one, $\Box$, which quantifies 'vertically' over certain of its subsets, expressing decrease in this way. The two modalities also interact (and their duals respectively). This interaction depends on the semantical structures to be examined. It is a challenging task in general to describe this interaction axiomatically.

Several classes of *set spaces*, i.e., pairs $(X, \mathcal{O})$ such that X is a non-empty set and $\mathcal{O}$ is a set of subsets of X , have been treated on the basis of the just indicated *language of set spaces*, *LSS*, up to now; see [6] for some examples, in addition to [3]. In view of what follows in the second part of this draft let us mention a particular variation of *LSS* where shrinking of sets proceeds in *discrete steps*; see [10] and [11].

In the present paper we dually deal with the *growth* of sets. We modify the interpretation of the $\Box$ -operator of the logic of set spaces appropriately. It turns out that the new situation is not quite similar to that of topological modal logic: the results are even more satisfactory here, because we get semantical completeness exactly with respect to the intended linear structures; contrasting with this see [11].

Why do we consider systems of such a general type? — We want to provide a test case presently, if our approach is adequate for dealing with phenomena indicated by the above examples, and different ones. Due to the issues of this paper we feel that our setting is in fact promising. Adding further operators and turning to finer structured semantical domains, as it has been done with the 'classical' systems mentioned at the beginning, special purpose systems applicable in 'real life' should result.

The outline of the paper is as follows. We introduce a logical language, *LGS*, which is able to speak about the *growth* of sets. This is done in two ways. First, only the modal operators K and $\Box$ are present, as in the common logic of set spaces. But the $\Box$ -operator quantifies 'upwards' over sets now, instead of the usual quantification 'downwards' considered in the logic of set spaces. We define the syntax and the semantics of the language precisely and give a list of axioms which are valid in all of the intended semantical domains. We prove completeness of the given axiomatization afterwards, which turns out to be 'nearly canonical'. The logic is shown to be also decidable. In fact, the so-

¹ Many years ago certain connections between modal logic and topology have already been discovered; see [12]. Recently, they have been utilized also for spatial reasoning; see [13].

² There are different formalisms of computational logic dealing with dynamic aspects of sets as well, but having other main points of emphasis; see [4], [9], for instance.

called *small model property* holds; i.e., every formula α^3 that is not a theorem can be falsified in a finite model of the axioms; furthermore, the size of this model depends computably on the length of α . — In the second part we add the well-known operators of *nexttime* and *until* to *LGS*, yielding the desired generalization of linear time temporal logic. The latter system is widely used in computer science, especially for verifying concurrent programs. We deal with the same questions as in the first case and obtain analogous results. But technical concerns become more complicated now due to the presence of the additional modalities; e.g., we have to work with suitable filtrations, already in the course of the completeness proof.

The paper requires acquaintance of the reader with the basics of propositional modal and temporal logic; e.g., canonical model and filtration techniques are applied frequently; see the textbooks [1] and [2]. In particular, Part One and Part Two of [7] supply enough background. — As space is limited, we have to omit many details, especially in proofs.

2 The Language

The definition of the *syntax* of *LGS* starts at a suitable (finite) alphabet, which in particular enables one to define a recursive set of *propositional variables*, *PV*. The set $\mathcal{F}$ of *LGS-formulas* is defined as the smallest set of strings satisfying the following clauses:

$$PV \subseteq \mathcal{F}, \text{ and } \alpha, \beta \in \mathcal{F} \implies \neg\alpha, K\alpha, \Box\alpha, (\alpha \wedge \beta) \in \mathcal{F}.$$

We omit brackets and use abbreviations as it is usual; furthermore, we let

$$L\alpha := \neg K\neg\alpha \text{ and } \Diamond\alpha := \neg\Box\neg\alpha.$$

The idea to define the *semantics* of *LGS* is as follows. We would like to describe the growth of a given set, Y . Thus certain supersets of Y have to be considered in the formal model. Consequently, we take a universe, X , in which all these sets are contained, and the system of these sets, $\mathcal{O}$, as the basic ingredients of interpreting formulas, which is done w.r.t. a truth-value assignment to propositions, σ . Hence we will consider certain triples $(X, \mathcal{O}, \sigma)$ as relevant structures, which are specified by the subsequent definition.

Definition 1. 1. Let X be a non-empty set and $\mathcal{O}$ a set of non-empty subsets of X . Then the pair $\mathcal{S} = (X, \mathcal{O})$ is called a *subset frame*.
 2. A subset frame $\mathcal{S}$ is called *linear*, iff $\mathcal{O}$ is linearly ordered by inclusion.⁴
 3. Let $\mathcal{S} = (X, \mathcal{O})$ be a (linear) subset frame and $\sigma : PV \times X \longrightarrow \{0, 1\}$ a mapping. Then σ is called a *valuation*, and the triple $\mathcal{M} = (X, \mathcal{O}, \sigma)$ is called a (linear) *model* (based on $\mathcal{S}$).

³ Formulas are designated by lower case Greek letters subsequently.

⁴ We look upon $\mathcal{O}$ as a (not necessarily discrete) ascending chain mostly.

We are going to interpret formulas in models at *situations* of subset frames, which are simply pairs x, U (designated without brackets mostly) such that $x \in U \in \mathcal{O}$. The set U measures ‘distance’ to the point x .

Definition 2. Let a model $\mathcal{M} = (X, \mathcal{O}, \sigma)$ and a situation x, U of $(X, \mathcal{O})$ be given. Then we define for all $A \in PV$ and $\alpha, \beta \in \mathcal{F}$:

$$\begin{aligned} x, U \models_{\mathcal{M}} A & : \iff \sigma(A, x) = 1 \\ x, U \models_{\mathcal{M}} \neg \alpha & : \iff x, U \not\models_{\mathcal{M}} \alpha \\ x, U \models_{\mathcal{M}} \alpha \wedge \beta & : \iff x, U \models_{\mathcal{M}} \alpha \text{ and } x, U \models_{\mathcal{M}} \beta \\ x, U \models_{\mathcal{M}} K\alpha & : \iff y, U \models_{\mathcal{M}} \alpha \text{ for all } y \in U \\ x, U \models_{\mathcal{M}} \Box \alpha & : \iff x, V \models_{\mathcal{M}} \alpha \text{ for all } V \supseteq U \text{ contained in } \mathcal{O}. \end{aligned}$$

In case $x, U \models_{\mathcal{M}} \alpha$ is valid we say that α *holds in $\mathcal{M}$ at the situation x, U* ; moreover, the formula $\alpha \in \mathcal{F}$ *holds in $\mathcal{M}$* (denoted by $\models_{\mathcal{M}} \alpha$), iff it holds in $\mathcal{M}$ at every situation. If there is no ambiguity, we omit the index $\mathcal{M}$ subsequently.

Example 1. Let a subset frame $(X, \mathcal{O})$ be given by

$$X := \mathbb{N} \text{ and } \mathcal{O} := \{\{0, \dots, n\} \mid n \in \mathbb{N}\}.$$

Moreover, let $Y \subseteq \mathbb{N}$ be an arbitrary subset. Define a valuation σ by $\sigma(A, x) = 1 : \iff x \in Y$, for all $A \in PV$ and $x \in \mathbb{N}$. Then the scheme $LA \rightarrow K\Box LA$ holds in the resulting linear model. This is due to the fact that if the actual initial segment of $\mathbb{N}$ intersects Y , then this is valid for all subsequent ones (which are in fact larger).

In accordance with the aims of this paper outlined above we concentrate on *linear* models subsequently. We present a list of axioms which hold in all linear models, as one can easily see.

Axioms

- (1) All $\mathcal{F}$ -instances of propositional tautologies.
- (2) $K(\alpha \rightarrow \beta) \rightarrow (K\alpha \rightarrow K\beta)$
- (3) $K\alpha \rightarrow \alpha$
- (4) $K\alpha \rightarrow KK\alpha$
- (5) $L\alpha \rightarrow KL\alpha$
- (6) $(A \rightarrow \Box A) \wedge (\neg A \rightarrow \Box \neg A)$
- (7) $\Box(\alpha \rightarrow \beta) \rightarrow (\Box\alpha \rightarrow \Box\beta)$
- (8) $\Box\alpha \rightarrow \alpha$
- (9) $\Box\alpha \rightarrow \Box\Box\alpha$
- (10) $\Box(\Box\alpha \rightarrow \beta) \vee \Box(\Box\beta \rightarrow \alpha)$
- (11) $\Box K\alpha \rightarrow K\Box\alpha$,

for all $A \in PV$ and $\alpha, \beta \in \mathcal{F}$. — For convenience, let us give some comments on these axioms. The first scheme embeds propositional logic in the actual system. The next group of axioms, consisting of the schemes (2) – (5), is well-known from the common logic of knowledge of a single agent. In terms of modal logic,

(3) – (5) express the properties of reflexivity, transitivity and weak symmetry⁵, respectively, of the modal accessibility relation; compare also schemes (8) and (9). In this sense, axiom (10) corresponds to *weak connectedness*; i.e., given arbitrary points s, t, u of a usual Kripke frame (X, R) such that $s R t$ and $s R u$, then $t R u$ or $u R t$ or $u = t$ holds iff the scheme is valid in (X, R) ; presently it is responsible for linearity, in connection with (11). The group of axioms involving only the $\Box$ -operator comprises yet another peculiarity: the scheme (6). It allows us to define the semantics in the way we did above, namely by means of situations without explicit reference to time,⁶ but it implies that the system to be defined immediately is not closed under substitution; regarding content (6) says that we assume the atomic propositions to be ‘stable’ or ‘persistent’. Finally, the scheme (11) combining both modalities is associated with the growth of sets.

Adding the following rules we get a deductive system designated **GS**.

Rules

(1)	$\frac{\alpha \rightarrow \beta, \alpha}{\beta}$	(modus ponens)
(2)	$\frac{\alpha}{K\alpha}$	(K -necessitation)
(3)	$\frac{\alpha}{\Box\alpha}$	($\Box$ -necessitation)

In the next section we show how completeness of this system w.r.t. the class of linear models can be obtained.

3 Completeness

To prove completeness of the system **GS** w.r.t. linear models we use its *canonical model* $\widetilde{\mathcal{M}}$ extensively. This model is formed in the usual way ([7], §5); i.e., the domain C of $\widetilde{\mathcal{M}}$ consists of the set of all maximal **GS**-consistent sets of formulas, and the accessibility relations induced by the modal operators K and $\Box$ are defined as follows:

$$s \xrightarrow{L} t : \iff \{\alpha \in \mathcal{F} \mid K\alpha \in s\} \subseteq t, \text{ and } s \xrightarrow{\Diamond} t : \iff \{\alpha \in \mathcal{F} \mid \Box\alpha \in s\} \subseteq t,$$

for all $s, t \in C$. Finally, the distinguished valuation of the canonical model is defined by

$$\sigma(A, s) = 1 : \iff A \in s \quad (A \in PV, s \in C).$$

The subsequent *truth lemma* is well-known.

⁵ Also called the *eucledian property*; e.g., in [7] (p. 12). A binary relation R on a set X fulfills this property by definition, iff $s R t$ and $s R u$ implies $t R u$, for all $s, t, u \in X$.

⁶ This is in accordance with the logic of set spaces and appropriate for our present selection of modal operators.

Lemma 1. *Let us denote the usual satisfaction relation of multimodal logic by $\models$, and let $\vdash$ designate **GS**-derivability. Then it holds that*

$$\widetilde{\mathcal{M}} \models \alpha[s] \iff \alpha \in s, \text{ and } \widetilde{\mathcal{M}} \models \alpha \iff \vdash \alpha,$$

for all $\alpha \in \mathcal{F}$ and $s \in C$.

Part (a) of the following proposition is basically forced by axioms (3), (4) and (5), while axioms (8), (9) and (10) imply the assertion of (b); (c) is a consequence of the scheme (11).

Proposition 1. (a) *The relation $\xrightarrow{L}$ is an equivalence relation on the set C .*
 (b) *The relation $\xrightarrow{\diamond}$ on C is reflexive, transitive and weakly connected.*
 (c) *Let $s, t, u \in C$ be given such that $s \xrightarrow{L} t \xrightarrow{\diamond} u$. Then there exists a point $v \in C$ satisfying $s \xrightarrow{\diamond} v \xrightarrow{L} u$.*

Following a common manner of speaking in *LSS* let us call the property asserted in (c) the *modified cross property*. — The next proposition reads as [6], Proposition 3.4(d).

Proposition 2. *Let $s, t \in C$ be given such that $s \xrightarrow{L} t$ and $s \xrightarrow{\diamond} t$ holds. Then s and t coincide.*

Proof. One proves by induction on $\alpha \in \mathcal{F}$ that for all pairs $(s, s') \in C \times C$ the formula α belongs to *some* $t \in C$ satisfying $s \xrightarrow{\diamond} t \xrightarrow{\diamond} s'$ iff it belongs to *all* such t . This idea is due to Georgatos [6]. The properties of the relations $\xrightarrow{L}$ and $\xrightarrow{\diamond}$ assured in Proposition 1 have to be applied in the induction steps. In particular, weak connectedness of $\xrightarrow{\diamond}$ has to be used in case $\alpha = \Box\beta$, and the modified cross property in case $\alpha = K\beta$.

With the aid of this proposition one can prove the following result by an induction argument again.

Proposition 3. *The relation $\xrightarrow{\diamond}$ on C is antisymmetric.*

For every $s \in C$ let $[s]$ denote the $\xrightarrow{L}$ -equivalence class of s . A relation $\leq$ on the set of all such classes is defined as follows:

$$[s] \leq [t] : \iff \text{there are } s' \in [s], t' \in [t] \text{ such that } s' \xrightarrow{\diamond} t',$$

for all $s, t \in C$. — As a consequence of the above assertions and the modified cross property we get:

Corollary 1. *The relation $\leq$ is a partial order such that for every $s \in C$ the set $C_s := \{[t] \mid [s] \leq [t]\}$ is linearly ordered.*

Now let $\alpha \in \mathcal{F}$ be a non-**GS**-derivable formula. Then there exists $s \in C$ containing $\neg\alpha$. Let

$$C^s := \bigcup_{[s] \leq [t]} [t].$$

For all $t \in C$ such that $[s] \leq [t]$ define a function $f_t : C_t \longrightarrow C^s$ by

$$f_t([t']) := t_1 \in [t'] \text{ satisfying } t \xrightarrow{\diamond} t_1 \quad ([t'] \in C_t).$$

According to our previous results f_t is well-defined. Let $X := \{f_t \mid [t] \in C_s\}$. Furthermore, for every $[t] \in C_s$ let $U'_{[t]} := \{f_{t'} \mid t' \in [t]\}$, and let

$$U_{[t]} := \bigcup_{[s] \leq [t'] \leq [t]} U_{[t']}.$$

Denote the set $\{U_{[t]} \mid [t] \in C_s\}$ by $\mathcal{O}$. Finally, define a valuation σ on X by

$$\sigma(A, f_t) = 1 : \Longleftrightarrow A \in t,$$

for all propositional variables A and functions $f_t \in X$. Note that σ is correctly defined as well.

The structure $\mathcal{M} := (X, \mathcal{O}, \sigma)$ is a linear model by construction, and the following *truth lemma* is valid.

Lemma 2. *For all formulas $\beta \in \mathcal{F}$ and $t, t' \in C^s$ such that $[t] \leq [t']$ it holds that $f_t, U_{[t']} \models_{\mathcal{M}} \beta$ iff $\beta \in f_t([t'])$.*

Proof. By induction on the structure of formulas. In case $\beta \in PV$ the axiom scheme (6) has to be used.

As an immediate consequence we get the desired completeness result.

Theorem 1. *Every $\alpha \in \mathcal{F}$ that is not **GS**-derivable can be falsified in a linear model.*

4 Decidability

In this section we prove that the set S of all *LGS*-formulas satisfiable in a linear model is decidable. This result can be obtained by means of the *small model property* of our logic; i.e., every satisfiable formula α can even be satisfied in a *finite* linear model having bounded size that depends computably on the length of α . Subsequently the prerequisites for proving this result are given.

Definition 3. *Let $\mathcal{I} := (I, \leq)$ be a non-empty linearly ordered set.*

1. *A subset $\emptyset \neq J \subseteq I$ is called a segment of $\mathcal{I}$, iff there is no $i \in I \setminus J$ strictly between any two elements of J .*
2. *A partition of I into segments is called a segmentation of $\mathcal{I}$.*

We will have to consider segmentations of the set $\mathcal{O}$ of linear models such that the truth value of a given formula remains unaltered on every segment.

Definition 4. Let $\alpha \in \mathcal{F}$ be a formula, $\mathcal{M} = (X, \mathcal{O}, \sigma)$ a linear model, I an indexing set and $\mathcal{P} := \{\mathcal{O}_\iota \mid \iota \in I\}$ a segmentation of $\mathcal{O}$. Then α is called stable on $\mathcal{P}$, iff for all $\iota \in I$ and $x \in X$

$$\begin{aligned} x, U &\models \alpha \quad \text{for all } U \in \mathcal{O}_\iota \text{ containing } x, \text{ or} \\ x, U &\models \neg \alpha \quad \text{for all } U \in \mathcal{O}_\iota \text{ containing } x. \end{aligned}$$

We always can achieve a finite segmentation of $\mathcal{O}$ on which a given formula is stable.

Proposition 4. Let $\alpha \in \mathcal{F}$ be a formula and $\mathcal{M} = (X, \mathcal{O}, \sigma)$ a linear model. Then there exists a finite segmentation $\mathcal{P}_\alpha := \{\mathcal{O}_1, \dots, \mathcal{O}_n\}$ of $\mathcal{O}$ such that α is stable on $\mathcal{P}_\alpha$. Moreover, $\mathcal{P}_\alpha$ can be chosen such that it refines $\mathcal{P}_\beta$ for every subformula β of α .

Proof. $\mathcal{P}$ is constructed by induction on the structure of α . One starts with the trivial segmentation $\{\mathcal{O}\}$ in case α a propositional variable. Only the cases $\alpha = (\beta \wedge \gamma)$ and $\alpha = K\beta$ contribute to a refinement of the actually obtained segmentation:

Letting

$$\mathcal{P}_\beta = \{\mathcal{O}_1, \dots, \mathcal{O}_n\} \text{ and } \mathcal{P}_\gamma = \{\mathcal{O}'_1, \dots, \mathcal{O}'_m\}$$

be segmentations which exist for β and γ respectively according to the induction hypothesis, the set

$$\{\mathcal{O}_i \cap \mathcal{O}'_j \mid 1 \leq i \leq n, 1 \leq j \leq m\}$$

induces in an obvious way a finite segmentation $\mathcal{P}_{\beta \wedge \gamma}$ on which $\beta \wedge \gamma$ is stable, thereby refining both $\mathcal{P}_\beta$ and $\mathcal{P}_\gamma$.

On the other hand, to achieve a segmentation $\mathcal{P}_{K\beta}$ on which $K\beta$ is stable each segment $\mathcal{O}_i$ of $\mathcal{P}_\beta$ is split into at most two disjoint segments; the exact number depends on how the truth value of β is coming along through $\mathcal{O}_i$ ($1 \leq i \leq n$).

Given a formula $\alpha \in \mathcal{F}$ and a linear model $\mathcal{M} = (X, \mathcal{O}, \sigma)$, we consider a finite segmentation $\mathcal{P}_\alpha := \{\mathcal{O}_1, \dots, \mathcal{O}_n\}$ of $\mathcal{O}$ according to the above proposition. We define

$$V_i := \bigcup \mathcal{O}_i, \text{ for all } i \in \{1, \dots, n\},$$

and $\mathcal{M}' := (X, \{V_1, \dots, V_n\}, \sigma)$. Then $\mathcal{M}'$ is a linear model, too. Moreover, we get the following proposition which can be proved by structural induction.

Proposition 5. Let α , $\mathcal{M}$ and $\mathcal{M}'$ be as above. Then, for all subformulas β of α , $y \in X$, $i \in \{1, \dots, n\}$ and $V \in \mathcal{O}_i$,

$$y, V \models_{\mathcal{M}} \beta \iff y, V_i \models_{\mathcal{M}'} \beta.$$

As a consequence we obtain that passing from $\mathcal{M}$ to $\mathcal{M}'$ preserves the satisfiability of α .

Corollary 2. *Let $\mathcal{M} = (X, \mathcal{O}, \sigma)$ be a linear model such that $x, U \models \alpha$, for some $x \in X$ and $U \in \mathcal{O}$, and let $\mathcal{M}' = (X, \{V_1, \dots, V_n\}, \sigma)$ be derived from $\mathcal{M}$ as previously described. Then $x, V_i \models \alpha$ for some $i \in \{1, \dots, n\}$.*

Thus a satisfiable α holds at some situation of a linear model $\mathcal{M}' = (X, \mathcal{O}', \sigma)$ such that $\mathcal{O}'$ is finite, $\mathcal{O}' = \{V_1, \dots, V_n\}$. With the aid of the following equivalence relation on X , which depends on α , we arrive at a *finite* model realizing α as well:

$$x \sim y : \iff \left\{ \begin{array}{l} \text{for all } U \in \mathcal{O}' \text{ and } A \in PV \text{ occurring in } \alpha \text{ it holds that} \\ (x \in U \iff y \in U) \text{ and } (\sigma(A, x) = 1 \iff \sigma(A, y) = 1). \end{array} \right.$$

Let $[x]_\sim$ denote the equivalence class of x w.r.t. the relation $\sim$. Define a model $[\mathcal{M}] = ([X], [\mathcal{O}], [\sigma])$ in the following way:

- $[X] := \{[x]_\sim \mid x \in X\}$
- $[\mathcal{O}] := \{[V_1], \dots, [V_n]\}$, where $[V_i] := \{[x]_\sim \mid x \in V_i\}$ ($1 \leq i \leq n$)
- $[\sigma](A, [x]_\sim) := 1 \iff (\exists y \in [x]_\sim) \sigma(A, y) = 1$ ($A \in PV$ and $[x]_\sim \in [X]$).

Then a simple induction proves the second statement of the next lemma, whereas the first one holds obviously.

Lemma 3. *$[\mathcal{M}]$ is a linear model. Moreover, for all subformulas β of α , $x \in X$ and $U \in \mathcal{O}'$,*

$$x, U \models_{\mathcal{M}'} \beta \iff [x], [U] \models_{[\mathcal{M}]} \beta.$$

Therefore, it suffices to consider finite linear models in order to falsify a given non-**GS**-derivable formula. Since the size of the carrier is in addition bounded effectively by the length of α , we have in fact established the small model property for **GS**. In particular, an algorithm deciding satisfiability can easily be obtained from this. Thus we can state:

Theorem 2. *The set S of all satisfiable LGS-formulas is decidable.*

5 The System LGS

In this section we briefly describe a *linear time version* of the logic of the growth of sets.

First, we mention the changes concerning the logical language. Two further operators, the *nexttime* operator $\bigcirc$ and the *until* operator $\mathcal{U}$, appear in formulas currently.⁷ To facilitate their intended interpretation, the set $\mathcal{O}$ occurring in linear models $\mathcal{M} = (X, \mathcal{O}, \sigma)$ is in fact a *discrete ascending chain* of subsets of X from now on; thus

$$\mathcal{O} = \{U_i \mid i \in \mathbb{N}\}, \text{ and } U_i \subseteq U_j \iff i \leq j.$$

⁷ Let us carry on designating the set of formulas $\mathcal{F}$.

The interpretation of a formula $\bigcirc\alpha$ at a situation x, U_i then reads

$$x, U_i \models_{\mathcal{M}} \bigcirc\alpha : \Longleftrightarrow x, U_{i+1} \models_{\mathcal{M}} \alpha,$$

whereas

$$x, U_i \models_{\mathcal{M}} \alpha \mathcal{U} \beta : \Longleftrightarrow \exists j \geq i : x, U_j \models_{\mathcal{M}} \beta, \text{ and } \forall i \leq k < j : x, U_k \models_{\mathcal{M}} \alpha$$

gives the interpretation of *until*. The interpretation of the K - and of the $\Box$ -operator remains unaltered.

Next we turn to an axiomatization of the validities. Only the schemes (1) – (5) and (7) of the list constituting the system **GS** are retained.⁸ The following have to be added:

- (6') $(A \rightarrow \bigcirc A) \wedge (\neg A \rightarrow \bigcirc \neg A)$
- (8') $\Box\alpha \rightarrow \alpha \wedge \bigcirc\Box\alpha$
- (9') $\bigcirc(\alpha \rightarrow \beta) \rightarrow (\bigcirc\alpha \rightarrow \bigcirc\beta)$
- (10') $\bigcirc\neg\alpha \leftrightarrow \neg\bigcirc\alpha$
- (11') $\bigcirc K\alpha \rightarrow K\bigcirc\alpha$
- (12') $\Box(\alpha \rightarrow \bigcirc\alpha) \rightarrow (\alpha \rightarrow \Box\alpha)$
- (13') $\alpha \mathcal{U} \beta \rightarrow \Diamond\beta$
- (14') $\alpha \mathcal{U} \beta \leftrightarrow \beta \vee (\alpha \wedge \bigcirc(\alpha \mathcal{U} \beta))$,

for all $A \in PV$ and $\alpha, \beta \in \mathcal{F}$. — The scheme (6') corresponds with (6) and the scheme (11') with (11) in an obvious manner. Clearly, (8') implies (8); furthermore, with the aid of (8') also the scheme (9) of the former list can easily be derived. Thus, besides (13') and (14'), the scheme (12') is the only one which is essentially new, capturing the principle of induction. Note that (8'), (11'), (12') are designated *Mix*, *Fun*, *Ind*, respectively, in [7], §9.

Adding the necessitation rule belonging to the operator $\bigcirc$,

$$(4) \quad \frac{\alpha}{\bigcirc\alpha} \quad (\bigcirc\text{-necessitation}),$$

we obtain our second logical system, **LGS**.⁹

The first theorem in this section states that **LGS** is sound and complete w.r.t. the class of linear models.

Theorem 3. *The set of formulas derivable in the system **LGS** coincides with the set of formulas holding in all linear models.*

While the soundness part of this theorem is straightforward to prove, it takes some trouble to establish completeness. Nevertheless, the proof partly proceeds along the lines of that for linear time temporal logic; so, we may focus on the differences only.

⁸ All the others are in fact derivable in the new system.

⁹ For the sake of brevity we suppress consideration of the *until* operator from now on. In fact, this operator does not demand new techniques.

Let $\alpha \in \mathcal{F}$ be given. Define the following sets of formulas based on the set $sf(\alpha)$ of subformulas of α :

$$\begin{aligned}\Gamma &:= sf(\alpha) \cup \{\bigcirc \Box \beta \mid \Box \beta \in sf(\alpha)\} \\ \Gamma^\neg &:= \Gamma \cup \{\neg \beta \mid \beta \in \Gamma\} \\ \Gamma^\wedge &:= \Gamma^\neg \cup \{\beta \mid \beta \text{ a finite conjunction of distinct elements of } \Gamma^\neg\} \\ \Gamma^L &:= \{L\beta \mid \beta \in \Gamma^\wedge\} \\ \tilde{\Gamma} &:= \Gamma^\wedge \cup \Gamma^L.\end{aligned}$$

Obviously $\tilde{\Gamma}$ is finite and closed under subformulas.

As in the completeness proof for **GS** in Section 3 we take advantage of the set of maximal **LGS**-consistent sets of formulas. Let us denote this set C . Note that we have an additional accessibility relation on C now, corresponding to the modal operator $\bigcirc$:

$$s \xrightarrow{\bigcirc} t : \iff \{\alpha \in \mathcal{F} \mid \bigcirc \alpha \in s\} \subseteq t \quad (s, t \in C).$$

Due to the scheme (10') the relation $\xrightarrow{\bigcirc}$ is actually a *function*.

Subsequently we build a certain filtration w.r.t. $\tilde{\Gamma}$ of a generated substructure; we again write C for the resulting carrier – by abuse of notation. To this end we first define

$$s \sim t : \iff s \cap \tilde{\Gamma} = t \cap \tilde{\Gamma},$$

for all $s, t \in C$. Moreover, we let $\bar{s}$ designate the $\sim$ -equivalence class of s , and $\bar{C} := \{\bar{s} \mid s \in C\}$. Since $\tilde{\Gamma}$ is finite, $\bar{C}$ is a finite set as well.

So far we have formed a filtration of the set C . Next we introduce filtrations of the accessibility relations. For convenience of the reader we repeat the corresponding definition. Let Δ be a modal operator under consideration, i.e., $\Delta = K, \bigcirc$, or $\Box$. Let ∇ denote its dual.¹⁰ Then, designating $\xrightarrow{\nabla}$ the accessibility relation on C which belongs to the operator Δ , as above, a binary relation $\vdash^\nabla$ on $\bar{C}$ is called a $\tilde{\Gamma}$ -filtration of $\xrightarrow{\nabla}$, iff the following requirements are satisfied for all $s, t \in C$:

- $s \xrightarrow{\nabla} t$ implies $\bar{s} \vdash^\nabla \bar{t}$, and
- $\bar{s} \vdash^\nabla \bar{t}$ implies $\{\beta \mid \Delta \beta \in s \cap \Gamma\} \subseteq t$.

Now let the binary relation $\xrightarrow{\bigcirc}$ on $\bar{C}$ be a filtration of $\xrightarrow{\bigcirc}$. Let $\xrightarrow{\bigcirc^*}$ denote the reflexive and transitive closure of $\xrightarrow{\bigcirc}$. The following lemma expresses the essential advantage of passing from the canonical model to a filtration; see [7], 9.8.

Lemma 4. *The relation $\xrightarrow{\bigcirc^*}$ on $\bar{C}$ is a filtration of the relation $\xrightarrow{\diamond}$ on C .*

¹⁰ Because of axiom (10') the operator $\bigcirc$ is self-dual.

The *minimal* filtration of the relation $\xrightarrow{\nabla}$ plays a crucial part in the present completeness proof. It is defined by

$$\bar{s} \xrightarrow{\nabla} \bar{t} : \iff \text{there are } s' \in \bar{s}, t' \in \bar{t} \text{ such that } s' \xrightarrow{\nabla} t',$$

for all $s, t \in C$; it is in fact a filtration of $\xrightarrow{\nabla}$ ($\nabla = \bigcirc, L$).

A first property substantiating the importance of the minimal filtration is stated in the so-called *Fun-lemma* ([7], 9.9), which can be obtained as a consequence of the scheme (10'):

Lemma 5. *Let $\xrightarrow{\bigcirc}$ be the minimal filtration of $\xrightarrow{\bigcirc}$. Let $s \in C$ be given and assume that $\bigcirc\beta \in \Gamma$. Then*

$$\bigcirc\beta \in s \text{ iff there is a point } t \in C \text{ such that } \bar{s} \xrightarrow{\bigcirc} \bar{t} \text{ and } \beta \in t.$$

Passing to a filtration, functionality of $\xrightarrow{\bigcirc}$ is lost in general. The above lemma represents its substitute.

There is a further rather technical part of the completeness proof for linear time temporal logic that can be transmitted to the present case. It deals with a meticulous analysis of the *clusters* of the relation $\xrightarrow{\bigcirc^*}$ and a related one, respectively, in which it is essentially used that the scheme

$$\Box(\Box(\alpha \rightarrow \Box\alpha) \rightarrow \alpha) \rightarrow (\Diamond\Box\alpha \rightarrow \Box\alpha)$$

is **LGS**-derivable; see [7], 9.11, for the details, which are skipped here.

We now proceed to the peculiarities of the system **LGS**. Note that we did not apply axiom scheme (11') so far. This scheme is fundamental for the following result which is also caused by the fact that the set $\tilde{\Gamma}$ has been defined appropriately.

Proposition 6. *The relation $\xrightarrow{L}$ is an equivalence relation on $\bar{C}$. Moreover, the modified cross property holds; i.e., for all $s, t, u \in C$ such that $\bar{s} \xrightarrow{L} \bar{t} \xrightarrow{\bigcirc} \bar{u}$ there exists $v \in C$ satisfying $\bar{s} \xrightarrow{\bigcirc} \bar{v} \xrightarrow{L} \bar{u}$.*

Proof. In order to establish the lemma some **LGS** proof theory has to be applied. Nevertheless, reflexivity and symmetry of the relation $\xrightarrow{L}$ are immediately clear, due to the fact that we deal with the *minimal* filtration of $\xrightarrow{L}$. Towards transitivity and the modified cross property we let for every $s \in C$

$$\tilde{\Gamma}_s := s \cap \tilde{\Gamma}, \text{ and } \psi_{\tilde{\Gamma}_s} := \bigwedge_{\beta \in \tilde{\Gamma}_s} \beta \wedge \bigwedge_{\gamma \in \tilde{\Gamma} \setminus \tilde{\Gamma}_s} \neg \gamma.$$

Now let $s, t \in C$ be given. Then the following conditions are equivalent:

- (a) $\bar{s} \xrightarrow{L} \bar{t}$;
- (b) $\psi_{\bar{t}_s} \wedge L\psi_{\bar{t}_t}$ is consistent;
- (c) $\vdash \psi_{\bar{t}_s} \rightarrow L\psi_{\bar{t}_t}$.¹¹

This is Proposition 2.8 of [3]. So, we can argue as in [3], proof of Lemma 2.10, to obtain transitivity of $\xrightarrow{L}$. Utilizing that the relation $\xrightarrow{L}$ is *symmetrical* we can also conclude from this that the modified cross property is valid.

We are going to show that every formula which is not derivable in **LGS** can be falsified in a linear model. So, let $\gamma \in \mathcal{F}$ be given and assume that $\not\vdash \gamma$. Then there exists a maximal consistent set s of formulas such that $\alpha := \neg \gamma \in s$. Let $\tilde{\Gamma}$ depending on α be as above and consider the filtration

$$\bar{\mathcal{M}} := (\bar{C}, \{\xrightarrow{L}, \xrightarrow{\bigcirc}, \xrightarrow{\bigcirc^*}\}, \bar{\sigma})$$

of $(C, \{\xrightarrow{L}, \xrightarrow{\bigcirc}, \xrightarrow{\diamond}\}, \sigma)$, where the latter structure is the submodel of the ‘canonical model’ generated by s , the valuation $\bar{\sigma}$ is induced by σ , and $\xrightarrow{L}, \xrightarrow{\bigcirc}$ are the minimal filtrations of $\xrightarrow{L}$ and $\xrightarrow{\bigcirc}$, respectively.

Starting with the class $\bar{s}$ of s and performing the unwinding procedure on $\bar{\mathcal{M}}$ as in the temporal completeness proof (see [7], p. 82) we obtain a sequence $f_{\bar{s}}$:

$$\bar{s} = \bar{s}_0, \bar{s}_1, \dots, \bar{s}_m, \dots, \bar{s}_n = \bar{s}_m, \bar{s}_{n+1} = \bar{s}_{m+1}, \dots, \bar{s}_{n+r \cdot (n-m)+i} = \bar{s}_{m+i}, \dots,$$

for all $r \in \mathbb{N}$ and $0 \leq i < n - m$, such that

$$\bar{s}_j \in \bar{C} \text{ and } \bar{s}_j \xrightarrow{\bigcirc} \bar{s}_{j+1}$$

holds for all $j \in \mathbb{N}$. For every $\nu \in \mathbb{N}$ and $\bar{t} \in \bar{C}$ such that $\bar{s}_\nu \xrightarrow{L} \bar{t}$ we inductively define a sequence $f_{\bar{t}}^\nu$ by

$$\begin{aligned} f_{\bar{t}}^\nu(0) &:= \bar{t} \\ f_{\bar{t}}^\nu(j+1) &:= \text{some } \bar{u} \in \bar{C} \text{ satisfying } \bar{s}_{\nu+j+1} \xrightarrow{L} \bar{u} \text{ and } f_{\bar{t}}^\nu(j) \xrightarrow{\bigcirc} \bar{u}. \end{aligned}$$

Given a pair $(\nu, \bar{t}) \in \mathbb{N} \times \bar{C}$ there always exist functions $f_{\bar{t}}^\nu$ satisfying these defining equations, according to Proposition 6. We then choose exactly one of them depending on $(\nu, \bar{t})$.

Call such an $f_{\bar{t}}^\nu$ *maximal*, iff $\nu = 0$ or there does not exist a $\bar{u} \in \bar{C}$ such that $\bar{u} \xrightarrow{\bigcirc} \bar{t}$ and $\bar{s}_{\nu-1} \xrightarrow{L} \bar{u}$. Let X be the set of all sequences $f_{\bar{t}}^\nu$ which are maximal. Furthermore, let

$$U_i := \{f_{\bar{t}}^\nu \in X \mid \nu \leq i\} \quad (i \in \mathbb{N}), \text{ and } \mathcal{O} := \{U_i \mid i \in \mathbb{N}\}.$$

¹¹ Currently, “ $\vdash$ ” denotes **LGS**-derivability.

Finally, let a valuation τ be given by

$$\tau(A, f_t^\nu) := \bar{\sigma}(A, \bar{t}),$$

for all $A \in PV$ and $f_t^\nu \in X$. Then $\widehat{\mathcal{M}} := (X, \mathcal{O}, \tau)$ is a linear model, and the results obtained so far can be used to prove the following assertion.

Proposition 7. *For all subformulas β of α , points $\bar{t}, \bar{u} \in \bar{C}$ and natural numbers $\nu, i \in \mathbb{N}$ such that $\nu \leq i$ and $f_u^\nu(j) = \bar{t}$ for some $j \in \mathbb{N}$, we have*

$$\bar{\mathcal{M}} \models \beta[\bar{t}] \iff f_u^\nu, U_i \models_{\widehat{\mathcal{M}}} \beta.$$

Since passing to a filtration preserves validity of the formula w.r.t. which the filtration has been carried out the proof of Theorem 3 can easily be completed now with the aid of Proposition 7.

The final topic in this section is *decidability* of the set of **LGS**-theorems.

The above completeness proof obviously does not yield the finite model property of our logic. So, decidability cannot be concluded directly. Nevertheless, we can make use of that proof to a large extent looking at certain Kripke models suitably derived from linear models.

Definition 5. *Let $\mathcal{M} := (W, \{R, S, T\}, \sigma)$ be a trimodal Kripke model; i.e., W is a non-empty set, R, S and T are binary relations on W , and σ is a valuation. Then $\mathcal{M}$ is called an **LGS**-model, iff*

- R is an equivalence relation on W ,
- S a function on W ,
- T is the reflexive and transitive closure of S ,
- for all $s, t, u \in W$: if $s R t$ and $t S u$, then there exists an element $v \in W$ such that $s S v$ and $v R u$,
- for all $s, t \in W$ such that $s S t$ and every $A \in PV$ it is valid that $\mathcal{M} \models A[s]$, iff $\mathcal{M} \models A[t]$.

In the above definition the relation R corresponds with the modality K ; accordingly, S and $\bigcirc$ as well as T and $\Box$ are related.

It turns out that the system **LGS** is sound and complete w.r.t. the class of **LGS**-models as well.

Theorem 4. *A formula $\alpha \in \mathcal{F}$ is **LGS**-derivable, iff it holds in every **LGS**-model.*

Proof. Soundness can easily be established. Towards completeness we let for a given linear model $\mathcal{M} = (X, \mathcal{O}, \sigma)$, where $\mathcal{O} = \{U_i \mid i \in \mathbb{N}\}$,

- $W := \{(x, U_i) \mid x \in U_i \in \mathcal{O}\}$
- $(x, U_i) R (y, U_j) : \iff i = j$
- $(x, U_i) S (y, U_j) : \iff x = y$ and $j = i + 1$
- $(x, U_i) T (y, U_j) : \iff x = y$ and $j \geq i$
- $\tilde{\sigma}(A, (x, U_i)) = 1 : \iff \sigma(A, x) = 1$,

for all $A \in PV$, $x, y \in X$, $i, j \in \mathbb{N}$ and $U_i, U_j \in \mathcal{O}$.

In this way $\mathcal{M}$ gives rise to an *LGS*-model $\mathcal{M}^\# := (W, \{R, S, T\}, \tilde{\sigma})$, and an easy induction shows that for all $\alpha \in \mathcal{F}$ and $(x, U) \in W$ it holds that

$$x, U \models_{\mathcal{M}} \alpha \text{ iff } \mathcal{M}^\# \models \alpha[x, U].$$

Applying now Theorem 3 yields completeness of **LGS** w.r.t. *LGS*-models.

Let us revisit the finite structure $\bar{\mathcal{M}}$ obtained by filtration in the course of the previous completeness proof for **LGS**. Apart from the functionality of the relation $\stackrel{\circ}{\mapsto}$, $\bar{\mathcal{M}}$ is already an *LGS*-model. But the remaining shortcoming can be circumvented in a standard manner which is completely analogous to the proceeding in temporal logic. In fact, *LGS*-models play the part here which *induction models* play there; see [7], p. 84 f., for the details.

Consequently, we get the finite (small) model property of the system **LGS** w.r.t. *LGS*-models. This implies the claimed decidability result.

Theorem 5. *The set of formulas $\alpha \in \mathcal{F}$, which hold in every linear model, is decidable.*

6 Concluding Remarks

We have introduced two languages modelling the growth of sets, a modal and a (linear time) temporal one, and we have developed corresponding logics. In both cases a *completeness* and a *decidability* result has been obtained. Moreover, the second formalism provides a nice and useful generalization of propositional linear time temporal logic.

The *complexity* of the given systems is high because of their relationship with the logic of synchronous systems with no learning mentioned in the introduction; see [8]. Note, however, that the logic **GS** is also close to the standard modal system S4.3, of which the satisfiability problem is NP-complete; see [14].

Acknowledgement We would like to thank one of the anonymous referees for a suggestion concerning the completeness proof of the previous section.

References

1. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Forthcoming; see URL <http://turing.wins.uva.nl/~mdr/Publications/modal-logic.html>
2. Chellas, B. F.: Modal Logic: An Introduction. Cambridge University Press, Cambridge (1980)
3. Dabrowski, A., Moss, L.S., Parikh, R.: Topological Reasoning and The Logic of Knowledge. *Annals of Pure and Applied Logic* **78** (1996) 73–110
4. Davoren, J.: Modal Logics for Continuous Dynamics. PhD dissertation, Department of Mathematics, Cornell University (1998)
5. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge(Mass.) (1995)

6. Georgatos, K.: Knowledge on Treelike Spaces. *Studia Logica* **59** (1997) 271–301
7. Goldblatt, R.: *Logics of Time and Computation*. CSLI Lecture Notes 7, Stanford, CA (1987)
8. Halpern, J.Y., Vardi, M.Y.: The Complexity of Reasoning about Knowledge and Time. I. Lower Bounds. *Journal of Computer and System Sciences* **38** (1989) 195–237
9. Hansen, M.R., Chaochen, Z.: Duration Calculus: Logical Foundations. *Formal Aspects of Computing* **9** (1997) 283–330
10. Heinemann, B.: Topological Nexttime Logic. In: Kracht, M., de Rijke, M., Wansing, H., Zakharyashev, M. (eds.): *Advances in Modal Logic*, Vol. 1. CSLI Publications 87, Stanford, CA (1998) 99–113
11. Heinemann, B.: Temporal Aspects of the Modal Logic of Subset Spaces. *Theoretical Computer Science* **224**(1-2) (1999) 135–155
12. McKinsey, J.C.C.: A Solution to the Decision Problem for the Lewis Systems S2 and S4, with an Application to Topology. *J. Symbolic Logic* **6**(3) (1941) 117–141
13. Nutt, W.: On the Translation of Qualitative Spatial Reasoning Problems into Modal Logics. *Lecture Notes in Computer Science* Vol. 1701 (1999) 113–125
14. Ono, H., Nakamura, A.: On the Size of Refutation Kripke Models for Some Linear Modal and Tense Logics. *Studia Logica* **39** (1980) 325–333

Process Algebra versus Axiomatic Specification of a Real-Time Protocol

Antonio Cerone

Software Verification Research Centre,
The University of Queensland, Brisbane QLD 4072, Australia
antonio@it.uq.edu.au,
http://svrc.it.uq.edu.au/pages/Antonio_Cerone.html

Abstract. In this paper we present two different approaches used in specifying a well-known audio control protocol with real-time characteristics. The first approach is based on Circal, a process algebra that permits a natural representation of timing properties and the analysis of interesting aspects of timing systems. The second approach is based on the Timed Interval Calculus, a set-theoretical notation for concisely expressing properties of timed intervals. The comparison between the two approaches shows that they are almost complementary: the former allows an easy modelling of the most procedural aspects of the protocol and provides a fully automatic proof but cannot catch all timing aspects; the latter allows easy modelling of all timing properties but the proof is quite hard and cannot be fully automated. This suggests a decomposition of the proof into subproofs to be performed in different proof environments.

1 Introduction

The case study presented in this paper is a time-sensitive audio control protocol developed by Philips. This protocol is intended to be used in future high fidelity systems where control signals pass between the various components of the system such as an amplifier, CD players, speakers, radio, etc., via a bus. The presence of such a control network adds additional features to the system, but market forces require that this extra functionality is achieved at minimal costs with a minimum of additional electronic components. Most of the modules in such an audio system contain a microprocessor and it is these microprocessors which will be used to realise a hopefully robust communication mechanism between them.

The protocol is an important benchmark case study which has challenged a number of formalisms. A version of the protocol with only one sender and one receiver was previously specified and analysed by Bosscher et al. [2] using linear hybrid systems. In subsequent papers by other authors, properties stated and proved by hand [2] have been automatically verified by using model checking based tools, such as Kronos [8], HyTech [11], Uppaal [1,13] and the Circal System [4,5], and theorem provers, such as the Larch Prover [10]. Recently the protocol has been modelled in a real-time process algebra by Chen, who has used a weak bisimulation to manually verify its implementation against its specification [7].

In the current paper two very different approaches used in specifying the audio control protocol are directly compared. In the first approach [5] the protocol is modelled using the Circal process algebra. Although Circal [14] is a discrete formalism, it permits a natural representation of timing properties and the analysis of interesting aspects of timing systems [4,5,6]. The automated verification of the protocol is entirely performed within the process algebra framework using a model-checking technique without recourse to temporal logic.

In the second approach the protocol is modelled using the Timed Interval Calculus (TIC) [9]. The formal verification has been carried out by hand, but it is currently being automated through an implementation of the Timed Interval Calculus within the Ergo theorem prover [3].

We show that the two approaches are complementary in terms of complexity of the specification. The sender can be easily specified in an algorithmic fashion using the process algebra, but requires the definition of a complex infrastructure to build up the axiomatic specification in TIC. On the other hand, the time based decoding by the receiver can be immediately specified in TIC but is complex in Circal. Moreover, the process algebra cannot characterise a deadline for the decoding, due to the state explosion caused by the tick actions, whereas deadlines can be trivially defined in TIC. Finally, the correctness proof is simple and can be fully automated in the process algebra, whereas is quite complex in TIC.

2 Description of the Protocol

The protocol forms part of the physical layer of an interface bus that interconnects the components of stereo systems. Messages, which consist of finite sequences of 0 and 1 bits, are encoded by a sender into timed transitions of the voltage between two levels on the single wire bus connecting the components. Receivers in components attached to the bus interpret the voltage transitions and reconstruct the bitstream messages. The senders and receivers run on different microprocessors. Since the microprocessors run code in addition to the protocol software, sender and receiver clocks may not be synchronised in frequency.

We restrict our example to only two processors communicating over a bus, one sender and one receiver, each with its own independent clock [2]. In this way problems of bus collisions due to different senders sending at the same time are avoided. We also suppose that the delay on the bus is negligible and that the sender transmits just one message.

Figure 1(a) represents this view of the protocol. The sender processor of the audio control protocol accepts a message from a user which it then tries to send to the receiver processor over the bus. A message is encoded by the sender according to a Manchester encoding scheme which is an algorithm for constructing a bus signal equivalent of the message. The time flow is represented as a sequence of slots of the same length, $4 * Q_S$ (where $Q_S = 2.22 * 10^{-4}$ sec in the Philips protocol), and each bit of the sequence is sent by the sender as a voltage level transition in the middle of a slot: a bit 0 is a falling transition and a bit 1 is a rising transition (see Figure 1(b)).

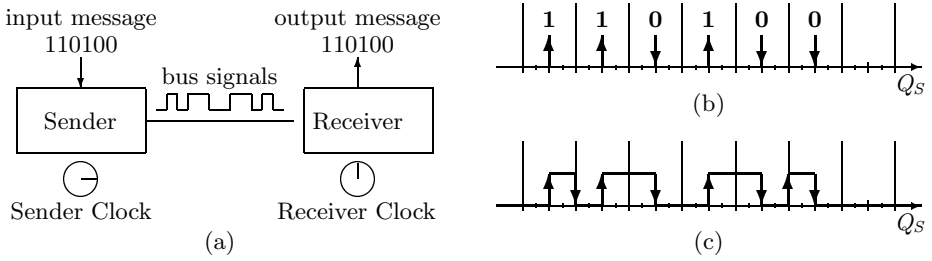


Fig. 1. (a) The audio control protocol; (b) Manchester encoding: rising and falling voltage level transitions; (c) Manchester encoding: additional transitions.

We assume that the voltage level on the wire is low before the message is transmitted and is reset to the low value after transmitting. When an identical bit is sent twice in a row an additional transition is added at the boundary between the corresponding slots (see Figure 1(c)).

Since falling transitions take a significant time to change from high to low level, they do not appear to the receiver as edges (see Figure 2(a)). Consequently the receiver can observe only rising edges and has to reconstruct the bitstream message from the lengths of the time intervals between successive rising edges (see Figure 2(b)). The receiver does not know when a slot starts, so there is a

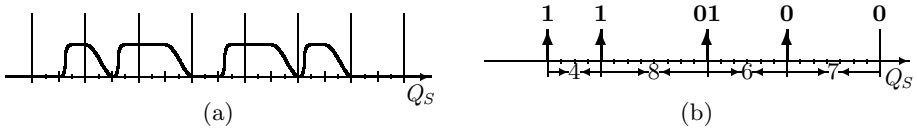


Fig. 2. Manchester encoding: (a) physical shape of the edges; (b) the receiver interprets the rising edges.

potential ambiguity in interpreting the start of a message. The receiver cannot distinguish between an initial 0, which would be encoded as a slot with a rising edge at the start and a falling edge in the middle, and an initial 1, which would be encoded with a rising edge in the middle of the slot. This problem is resolved by imposing the following constraint on the input:

C0 Every message starts with the bit 1

A further ambiguity arises from the loss of falling edges during transmission. The receiver cannot distinguish sequence 10 from a 1 at the end of a message. In this case the only difference is in the timing of the final falling edge, which cannot be observed by the receiver. This problem is overcome by imposing the following constraint on the input

C1 Every message either has an odd length or ends with 00

Let σ be the sequence of the bits that have already been decoded at a certain point in time. If only the last rising edge on the wire has still to be decoded at that time, and it is interpreted as occurring in the middle of a slot, then the decoded message is $\sigma 10$ when σ has an odd length and is $\sigma 1$ when σ has an even length. So the message can be decoded without ambiguity.

Because of the drift between the sender and receiver clocks, the distance between successive rising edges measured by the receiver is in general different from that measured by the sender. The protocol has been designed to achieve reliable communication even in the presence of this significant *timing uncertainty*. The interpretation of the time distance between successive rising edges will be correct provided that the drift is held within limits that depend on the protocol specification.

Let us define an upper bound on the time needed for the receiver to decode a given message. Since the first bit (which is always 1) is immediately encoded, if the given message consists of m bits, then the last bit is encoded by the sender after a time $4(m-1)Q_S$. However, the receiver has to wait an additional time in order to be sure that the end of the message has been reached. Actually, $8Q_S$ is the longest time that may occur between rising edges. It occurs when encoding the sequence 101. Moreover, the drift between sender and receiver must be taken into account. If ρ is the ratio between the sender clock rate and the receiver clock rate, then $Q_R = \rho Q_S$, where Q_S is the time unit of the sender (a quarter of a slot) and Q_R is the time unit of the receiver. The specification of the protocol [2] requires that the receiver detects the end of the message when a time $9Q_R$ has elapsed after observing the last rising edge. Therefore, the upper bound on the time needed to decode a given message is $4(m-1)Q_S + 9Q_R = (4m - 4 + 9\rho)Q_S$. Notice that the decoding is correct only if $8Q_S < 9Q_R$, that is $\rho > \frac{8}{9}$.

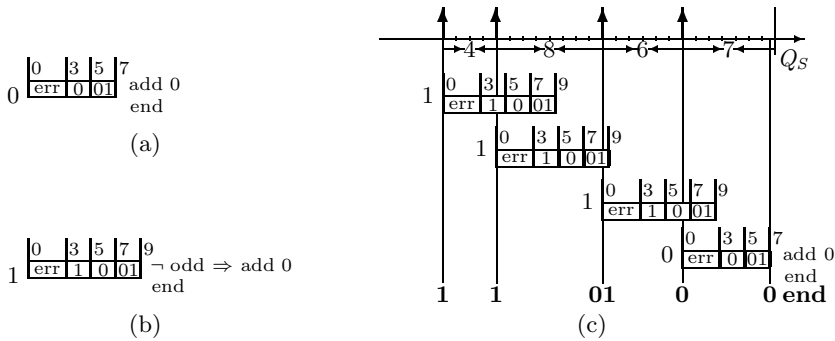


Fig. 3. (a) Decoding intervals when the previously decoded bit is 0. (b) Decoding intervals when the previously decoded bit is 1. (c) Example of message decoding.

In Figure 3 we illustrate the decoding of the message encoded in Figure 1. The time intervals between successive rising edges are interpreted in different ways according to the value of the bit previously decoded. Figure 3(a) and Figure 3(b) define the interpretation of the time intervals when the value of the bit previously

decoded is respectively 0 and 1. In Figure 3(c) the first rising edge is immediately decoded as a 1. The subsequent three edges are decoded using the interpretation given in Figure 3(b). The end of the message is detected using the interpretation in Figure 3(a). In this case an additional 0 is added to the decoded message. With the interpretation in Figure 3(b) the additional 0 is added at the end of the message only if the decoded prefix of the message does not have an odd length. In the example in Figure 3 the ratio between the sender clock rate and the receiver clock rate is $\rho = 1.067$.

3 The Process Algebra Approach

In this section we describe the modelling of the audio control protocol in the process algebra approach. For clarity, we do not use the formal notation of the process algebra, but instead present the protocol model in a graphical fashion. Circal is a process algebra with multiway communication similar to CSP [12], but with the additional feature that several actions can be performed simultaneously. We describe the sequential behaviour of a process using a finite state machine (FSM) diagram with transitions labelled by sets of simultaneous actions rather than single actions. Parallel composition of processes is described by representing components as boxes with bullets denoting communication ports. Interaction among processes is represented by connecting ports by means of arrows (for data passing) or lines (for pure synchronisation) labelled by actions. Hiding of internal actions is represented by enclosing components within a box and allowing only arrows and lines labelled by visible actions to go outside the box. Notice that, due to multiway communication, when more than two ports are connected we use additional bullets to join arrows or lines (see Figure 5).

3.1 Bitstream Messages

A message is readily modelled in a process algebra by a nested series of guarded processes where the guards are events consisting of single actions that can describe a bit 0, or a bit 1, or the “end of the message”. Different actions are used for input and output message. We represent 0, 1 and “end of the message” by in_0 , in_1 and in_E , respectively, in the input, and by out_0 , out_1 and out_E , respectively, in the output. For example, message 110 is represented in the input as $in_1 in_1 in_0 in_E \Delta$ and in the output as $out_1 out_1 out_0 out_E \Delta$, where Δ is the termination process. The messages that satisfy Constraints **C0** and **C1** defined in Section 2 are characterized by the C process whose behaviour is defined by the FSM in Figure 4. Process C forces messages to start with in_1 (Constraint **C0**) and to be able to perform the in_E action only when Constraint **C1** is satisfied (that is in states O_0 , O_1 and E_{00}).

3.2 Protocol Physical Components

The main protocol physical components are the S sender process and the R receiver process, which are structured in further components as shown in Fig-

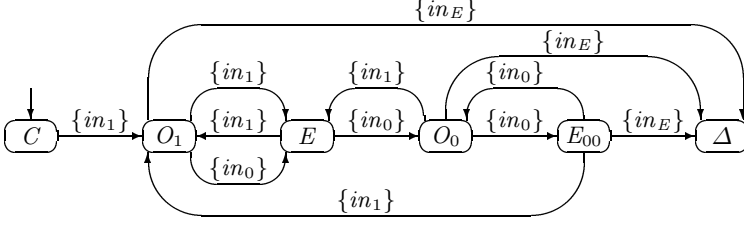


Fig. 4. Process C models constraints **C0** and **C1**.

ure 5(a) and in Figure 5(b), respectively. In S the I_{in} component transforms every input action in_y , $y = 0, 1, E$, into an action i_{zy} , where z is the value 0 or 1 of the previous input bit. If in_y , $y = 0, 1$, is the first bit of the message, then it is transformed into i_{0y} .

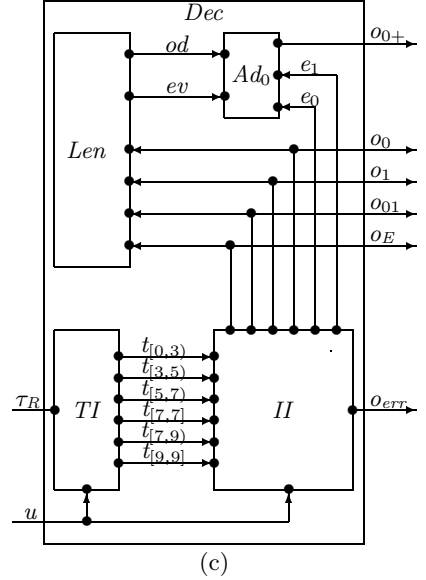
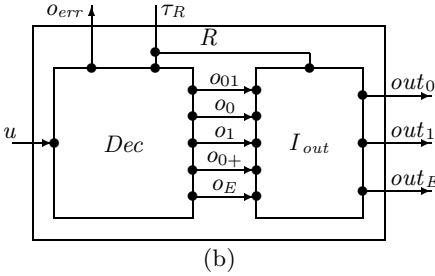
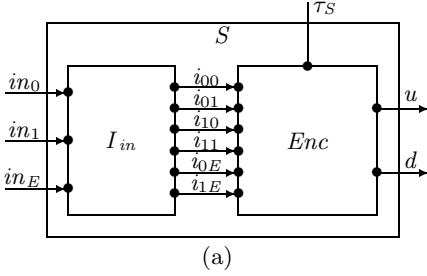


Fig. 5. (a) Components of sender process S ; (b) Components of receiver process R ; (c) Sub-components of decoder process Dec .

The Enc process [4,5] both defines the sender clock by means of the sender ticks τ_S belonging to each action set and generates the u and d actions, which model respectively rising and falling edges. Notice that in a process algebra action occurrences are related by a temporal order relation. The temporal order is in general a partial order, due to the presence of choice operators, which induces a branching behaviour. However, a quantitative representation of time is not

explicit, but is given by introducing tick actions, with the implicit assumption that the distance between two subsequent ticks is a fixed constant. Therefore, we assume that the distance between two subsequent τ_S ticks is Q_s . The encoding algorithm implemented by the *Enc* process is quite simple [4,5].

The *Dec* decoder process, which is a component of the receiver, is further structured as shown in Figure 5(c). The *TI* process measures the time occurring between two successive u events by counting the number of ticks τ_R and generates one of the following actions:

$$t_{[0,3)}, t_{[3,5)}, t_{[5,7)}, t_{[7,7]}, t_{[7,9)}, t_{[9,9]},$$

where action $t_{[a,b)}$ characterizes the time interval $[a, b)$ and action $t_{[a,a]}$ characterizes the time point $[a, a]$.

The *II* process generates the appropriate o_y output action for every simultaneous occurrence of one u and either one $t_{[a,b)}$ or one $t_{[a,a]}$, with $y \in \{err, 0, 1, 01, E\}$. Actions o_{err} , o_0 , o_1 , o_{01} and o_E define respectively an error, a bit 0, a bit 1, a sequence 01 and the end of the message. The current output action also depends on the previous output bit as shown in Figure 3. Therefore the value of the previous output bit is encoded in the state of the *II* process. Moreover

- the *II* process generates simultaneously with every o_E either an action e_0 , if the last decoded bit is 0, or an action e_1 , if the last decoded bit is 1;
- the *Len* process records by means of events od and ev the odd and even length of the part of the message that has already been decoded;
- the *Ad₀* process adds an additional 0 at the end (action o_{0+} occurring simultaneously with o_E) when the detected message has an even length (occurrence of action ev), or ends with 0 (occurrence of action e_0).

The *I_{out}* process transforms o_0 into out_0 , o_1 into out_1 , o_{01} into out_0out_1 , action set $\{o_{0+}, o_E\}$ into out_0out_E and o_E into out_E .

3.3 The Constraining Processes

In addition to the physical *components* of the protocol, we need to model *assumptions* on the behaviour of the system. The assumptions given by the *C* process in Figure 4 characterise all the messages that satisfy Constraints **C0** and **C1**. A second constraint must characterise the drift between sender and receiver. In our analysis we suppose that the rate of the sender and receiver clocks are steady and therefore we consider a fixed rate of drift. An example of such a constraint is modelled by the $D_{8,9}$ process in Figure 6, which constrains the temporal ordering of the τ_S and τ_R actions such that there are 8 ticks of the sender clock every 9 ticks of the receiver clock. Notice that for any $D_{s,r}$ process, $s, r \in \mathbb{N}$, the number of ticks of the receiver clock before resetting to D_0 is at most 9, since this is the maximum length of a decoding interval (see Figure 3). Any $D_{s,r}$ process can be algorithmically generated for given values s and r [5].

The Circal model P of the audio control protocol is defined by composing in parallel S , R and $D_{s,r}$, $s, r \in \mathbb{N}$, and hiding u , d , τ_S and τ_R , as shown in Figure 7(a).

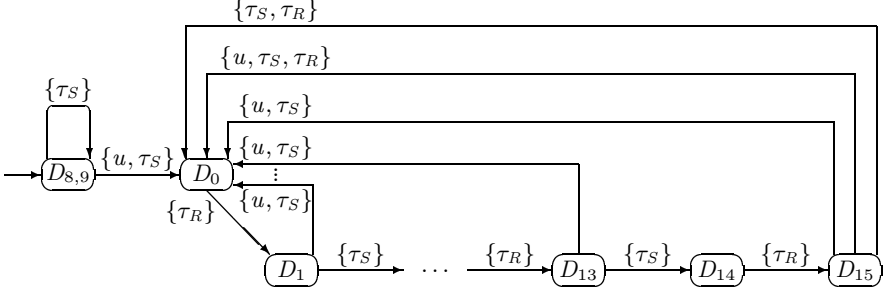


Fig. 6. Process $D_{8,9}$ models the drift between sender and receiver ($\rho = \frac{8}{9}$).

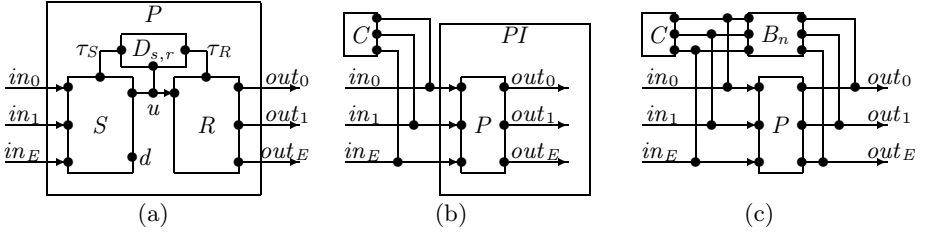


Fig. 7. (a) Model of the protocol; (b) Constrained input of the protocol; (c) Constrained protocol composed with the buffer.

3.4 Untimed Verification of the Protocol

Circal allows a quantitative representation of time in terms of tick actions [6], but this often leads to state explosion. Therefore, when verifying the protocol in the process algebra approach we do not base the notion of correctness on the upper bound of the time needed to decode the message. We define indeed the notion of correctness in terms of the following properties:

P0: The protocol is receptive to every message that satisfies **C0** and **C1**;

P1: If the protocol is receptive to a message that satisfies **C0** and **C1**, then the message is output unchanged.

Let PI be the process modelling the input to which the protocol is receptive, that is the protocol model with the output hidden. Property **P0** is satisfied iff the behaviour of the PI process is included in the behaviour of the C process. This is automatically verified using the Circal System by testing the equivalence between the C process and the parallel composition of C and PI (see Figure 7(b)). In fact, the behaviour of C includes the behaviour of PI iff PI does not constrain C , that is iff the parallel composition of PI and C is equivalent to C .

The fact that the message is output unchanged can be described by the B_n process, which is a buffer of capacity n that accepts three kinds of inputs, in_0 , in_1 and in_E , and can accept an input and generate an output simultaneously.

Therefore, property **P1** is satisfied iff the behaviour of the constrained protocol, that is the composition of C and P , is included in the behaviour of B_n . This is automatically verified using the Circal System by testing the equivalence between the constrained protocol, given by the parallel composition of P and C , and the parallel composition of the constrained protocol and the buffer, that is the parallel composition of P , C and B_n shown in Figure 7(c).

In practice Circal process P in our description is parameterised by the relative clock rates of the sender and receiver (expressed through values s and r and modelled by $D_{s,r}$). The B_n process is parameterised on the capacity n of the buffer. Verification of the protocol in the Circal System is carried out by testing whether the two equivalences given above as characterisations of properties **P0** and **P1** hold for particular values of the three parameters. It was found that the equivalences are true only if the buffer size was at least 2 and the protocol parameters were chosen such that $0.889 \approx \frac{8}{9} < \rho = \frac{s}{r} \leq \frac{8}{7} \approx 1.143$ [4,5], which is consistent with the theoretical value given in Section 2.

4 The Axiomatic Approach

The Timed Interval Calculus (TIC) is a Z-based notation for expressing properties of time intervals. It is based on a continuous time domain $\mathbb{T}$ modelled by the real numbers.

Time intervals are represented as the set of all times between some infimum a and supremum b . For instance $(a, b]$ denotes the left-open and right-closed interval between times a (exclusive) and b (inclusive). Similarly for (a, b) , $[a, b)$ and $[a, b]$. The set of all time intervals is denoted by $\mathbb{I}$.

The principal specification tool of TIC consists of special brackets for defining the set of all time intervals during which a given predicate is everywhere true [9]. For instance $\{P\}$ is the set of all left-open and right-closed time intervals $I \in \mathbb{I}$ such that for each time $t \in I$ predicate P is true at t . Similarly for $\langle P \rangle$, $[P]$ and $\lceil P \rceil$. Within predicate P there may be occurrences of functions on the time domain that are not applied to arguments; they must be interpreted as applied to every point in the whole interval I [9]. P may also contain the following special variables: α denotes the left endpoint of the interval, ω denotes the right endpoint of the interval and δ denotes the length of the interval. We define $\llbracket P \rrbracket = \langle P \rangle \cup [P]$. Similarly for $\llbracket P \rrbracket$, $\langle P \rrbracket$, $[P \rrbracket$ and $\lceil P \rrbracket$.

An important capability of TIC is an operator for connecting intervals end-to-end, to support reasoning about sequences of behaviours. The concatenation of two sets of intervals X and Y is the set $X; Y$ of all the intervals z such that there exist two disjoint intervals $x \in X$ and $y \in Y$ with supremum of x equal to infimum of y and whose union is z .

4.1 Sender

In the axiomatic approach a message is modelled as a sequence of bits. The set of all possible messages is the set Msg of non empty finite sequences on

$\text{Bit} = \{0, 1\}$. We also define the set of the boolean values $\mathbb{B} = \{\text{true}, \text{false}\}$. Let $\text{head} : \text{Msg} \rightarrow \text{Bit}$ and $\text{head2} : \text{Msg} \rightarrow \text{Msg}$ be the functions that return respectively the first bit and the sequence of the first two bits of the message, $\text{odd} : \text{Msg} \rightarrow \mathbb{B}$ be the function that returns true if the message has an odd length and false otherwise, $\text{tail} : \text{Msg} \rightarrow \text{Msg}$ be the function that returns the message without its first bit, $\text{front} : \text{Msg} \rightarrow \text{Msg}$ be the function that returns the message without its last bit, $\text{last} : \text{Msg} \rightarrow \text{Bit}$ and $\text{last2} : \text{Msg} \rightarrow \text{Msg}$ be the functions that return respectively the last bit and the sequence consisting of the last two bits of the message. The set of the messages that meet **C0** and **C1** is defined as follows.

$$\text{GoodMsg} = \{m \in \text{Msg} \mid \text{head } m = 1 \wedge (\text{odd } m = \text{true} \vee \text{last2 } m = \langle 0, 0 \rangle)\}$$

Let $Q = 2.22 * 10^{-4}$ be the constant that denotes one quarter of a slot measured by the sender. We set the infrastructure necessary for the encoding by introducing set

$$\text{MidSlot} = \{t \in \mathbb{T} \mid \exists n \in \mathbb{N}. t = 4nQ\},$$

variable $i \in \text{GoodMsg}$, which gives the input message, function $p : \mathbb{T} \rightarrow \text{Msg}$, which records the sequence of bits still to be decoded, function $u : \mathbb{T} \rightarrow \mathbb{B}$, which defines the rising edges, and the following axioms:

$$\mathbf{Neg.} \quad \llbracket \omega \leq 0 \rrbracket \subseteq \llbracket \text{head}(p) = 0 \wedge \text{tail}(p) = i \wedge u = \text{false} \rrbracket$$

$$\mathbf{Intvl.} \quad \langle \omega \in \text{MidSlot} \wedge \delta = 4Q \rangle \subseteq$$

$$\langle p = p(\alpha) \wedge (\#(p) > 0 \Rightarrow p(\omega) = \text{tail}(p)) \rangle$$

$$\mathbf{Up.} \quad \langle u = \text{true} \rangle = \emptyset$$

Axiom **Neg** defines function p at negative times by prefixing input message i with 0. The value of p at the left endpoint of an open interval whose right endpoint is the middle of a time slot and whose length is $4Q$ is extended by axiom **Intvl** to the whole open interval ($p = p(\alpha)$). Moreover, p is defined at the right endpoint as the value that will be extended to the next interval ($p(\omega) = \text{tail}(p)$). Notice that the formalism allows the definition of functions at endpoints that do not belong to the intervals characterised by the axioms. Therefore, axiom **Neg** is the base and axiom **Intvl** is the inductive step of the definition of p . Function u returns value true at any time when a rising edge occurs. Since rising edges are assumed to be instantaneous, axiom **Up** ensures that there is no open interval where $u = \text{true}$, that is $u = \text{true}$ at isolated points only. Moreover axiom **Neg** ensures that no rising edge occurs at a negative time.

The encoding of the message is then defined by the following axioms:

$$\mathbf{Enc00.} \quad \langle \omega \in \text{MidSlot} \wedge \delta = 2Q \wedge \text{head2}(p) = \langle 0, 0 \rangle \rangle ; \langle \delta = 0 \rangle \subseteq \langle u = \text{true} \rangle ; \langle u = \text{false} \rangle$$

$$\mathbf{Enc01.} \quad \langle \omega \in \text{MidSlot} \wedge \text{head2}(p) = \langle 0, 1 \rangle \rangle ; \langle \delta = 0 \rangle \subseteq \langle u = \text{false} \rangle ; \langle u = \text{true} \rangle$$

- Enc10.** $\{\omega \in \text{MidSlot} \wedge \text{head2}(p) = \langle 1, 0 \rangle\} ; \{\delta = 0\} \subseteq \{u = \text{false}\}$
- Enc11.** $\{\omega \in \text{MidSlot} \wedge \delta = 2Q \wedge \text{head2}(p) = \langle 1, 1 \rangle\} ; \{\delta = 0\} \subseteq \{u = \text{false}\} ; \{u = \text{true}\}$
- EncNext.** $\{\alpha \in \text{MidSlot} \wedge \delta = 2Q\} \subseteq \{u = \text{false}\}$
- EncEnd.** $\{\alpha \in \text{MidSlot} \wedge \#(p(\alpha)) \leq 1 \wedge \delta > 0\} \subseteq \{u = \text{false}\}$

The first four axioms define the encoding of the current bit, taking into account the last bit that has already been encoded. This is done by *sampling* the value of p on the first half of the current time slot, which is a $2Q$ long interval. If the bit to be encoded is 1, then, independently of the last encoded bit, a rising edge is generated on the right endpoint of the interval, which is the middle of the current time slot (Axioms **Enc01** and **Enc11**). When the last encoded bit is also 1 (Axiom **Enc11**), a falling edge is generated at the left endpoint of the interval, but does not need to be represented. If the bit to be encoded is 0 (Axioms **Enc00** and **Enc10**), then the falling edge generated on the right endpoint of the interval does not need to be represented, while a raising edge is generated at the left endpoint of the interval when the last encoded bit is also 0 (Axiom **Enc00**).

Axiom **EncNext** ensures that after the middle of a slot there is at least an open gap of $2Q$ without rising edges. Axiom **EncEnd** ensures that when the length of p is not greater than 1, no more rising edges may be generated.

4.2 Receiver

The output of the protocol is expressed by function $o : \mathbb{T} \rightarrow \text{Msg}$. Let λ denote the empty message and the $\wedge$ infix operator denote concatenation of sequences. The decoding by the receiver is a simple translation of the decoding intervals given in Figure 3(a) and in Figure 3(b) into TIC:

- DecFirst.** $\{\alpha = 0 \wedge u = \text{true}\} \subseteq \{o(\omega) = \langle 1 \rangle\}$
- Dec03.** $\{u = \text{true}\} ; \{u = \text{false} \wedge 0 \leq \delta < 3\rho Q\} ; \{u = \text{true}\} \subseteq \{o(\omega) = \lambda\}$
- Dec35.** $\{u = \text{true}\} ; \{u = \text{false} \wedge 3\rho Q \leq \delta < 5\rho Q\} ; \{u = \text{true}\} \subseteq \{o(\omega) = o(\alpha) \wedge \langle \text{last } o(\alpha) \rangle\}$
- Dec570.** $\{u = \text{true} \wedge \text{last } o = 0\} ; \{u = \text{false} \wedge 5\rho Q \leq \delta < 7\rho Q\} ; \{u = \text{true}\} \subseteq \{o(\omega) = o(\alpha) \wedge \langle 0 \rangle \wedge \langle 1 \rangle\}$
- Dec571.** $\{u = \text{true} \wedge \text{last } o = 1\} ; \{u = \text{false} \wedge 5\rho Q \leq \delta < 7\rho Q\} ; \{u = \text{true}\} \subseteq \{o(\omega) = o(\alpha) \wedge \langle 0 \rangle\}$
- Dec70.** $\{u = \text{true} \wedge \text{last } o = 0\} ; \{u = \text{false} \wedge \delta = 7\rho Q\} ; \{\text{true}\} \subseteq \{o(\omega) = o(\alpha) \wedge \langle 0 \rangle \wedge \delta = 7\rho Q\} ; \{o = o(\alpha)\}$

Dec791. $\{u = \text{true} \wedge \text{last } o = 1\}; \{u = \text{false} \wedge 7\rho Q \leq \delta < 9\rho Q\};$

$$\{u = \text{true}\} \subseteq \{o(\omega) = o(\alpha) \wedge \langle 0 \rangle \wedge \langle 1 \rangle\}$$

Dec9e. $\{u = \text{true} \wedge \text{last } o = 1 \wedge \text{odd}(o) = \text{false}\}; \{u = \text{false} \wedge \delta = 9\rho Q\};$

$$\{\text{true}\} \subseteq \{o(\omega) = o(\alpha) \wedge \langle 0 \rangle \wedge \delta = 9\rho Q\}; \{o = o(\alpha)\}$$

Dec9o. $\{u = \text{true} \wedge \text{last } o = 1 \wedge \text{odd}(o) = \text{true}\}; \{u = \text{false} \wedge \delta = 9\rho Q\};$

$$\{\text{true}\} \subseteq \{o = o(\alpha)\}$$

Notice that the value of o is unspecified at most points in time during decoding. It is specified only when a rising edge occurs and gives the sequence of bits that has been decoded up to that point in time. However, when the end of the message is detected, the value of o is fixed to the decoded message for any subsequent time (Axioms **Dec70**, **Dec9e** and **Dec9o**).

4.3 Timed Verification of the Protocol

In the process algebra approach we have presented an untimed verification of the protocol. In TIC we can reason explicitly about time and prove not only that the message is decoded correctly, but that it is decoded within the deadline defined in Section 2.

Before verifying the protocol it is necessary to define the constraints under which the verification is carried out. Constraints **C0** and **C1** are built into the specification through the GoodMsg set. As for the drift we do not need to use a fixed value but just to give a lower and an upper bound. Therefore, we add to our specification the following axiom.

Drift. $(\rho \geq 0.9) \wedge (\rho \leq 1.1)$

We have seen that when the end of the message is detected the value of o is fixed to the decoded message for any subsequent time. The protocol is correct iff on every interval that starts at least at a time $(4(\#i) - 4 + 9\rho)Q$ after the middle of the first slot functions o and i are equal. Therefore, the correctness requirement for the protocol is the following:

Corr. $\{\alpha \geq (4(\#i) - 4 + 9\rho)Q\} \subseteq \{o = i\}$

The protocol was verified by proving that the conjunction of all given axioms implies **Corr**. The proof is somewhat involved but can be divided into the following steps:

1. $\{u = \text{true}\} \subseteq \{\text{last } o = \text{head } p \wedge (\#p \geq 2 \Rightarrow \text{last2 } o = \text{head2 } p)\}$
2. $\{u = \text{true}\} \subseteq \{(\text{front } o) \wedge p = i \wedge \delta = 0\}$
3. $\{\alpha = (4(\#i) - 4)Q \wedge \delta \geq 4Q\} \subseteq \{\#p = 1 \wedge \delta = 4Q\}; \{\#p = 0\}$
4. $\{u = \text{true}\}; \{u = \text{false}\}; \{\alpha = (4(\#i) - 4)Q \wedge \delta \geq 4Q\} \subseteq$
 $\{(\text{front } o) \wedge p = i \wedge u = \text{true}\}; \{u = \text{false}\};$
 $\{\alpha = (4(\#i) - 4)Q \wedge \delta = 4Q \wedge \#p = 1 \wedge u = \text{false}\}; \{u = \text{false}\}$

5. $\{u = \text{true}\} ; \{\delta \geq 9\rho Q \wedge u = \text{false}\} ; \{\alpha = (4(\#i) - 4 + 9\rho)Q\} \subseteq$
 $\{u = \text{true}\} ; \{\delta \geq 9\rho Q \wedge u = \text{false}\} ; \{o = i\}$
6. $\{\alpha = (4(\#i) - 4 + 9\rho)Q\} \subseteq \{o = i\}$

Step 1 is complex and requires an exhaustive analysis of all compatible combinations of pairs of encoding axioms and their subsequent combination with the appropriate decoding axioms consistently with Axiom **Drift**. This required several pages of proof.

Step 2 was obtained by induction from Step 1 with the additional use of Axiom **Up**.

Step 3 was obtained by induction from the following

Base. $\{\omega = 0 \wedge \delta = 4Q\} \subseteq \{\#p = \#i + 1\}$

Induct. $\{\omega \in \text{MidSlot} \wedge \delta = 4Q\} \subseteq \{\#p > 0 \Rightarrow (\#(p(\omega)) = \#p - 1)\}$

with induction base **Base** derived from **Neg** and induction step **Induct** derived from **Intvl**.

Since $\alpha = (4(\#i) - 4)Q \in \text{MidSlot}$ from Step 2, Step 3 and **EncEnd** we can derive Step 4.

Step 5 was obtained by the application of property

$$\{u = \text{true}\} ; \{u = \text{false} \wedge \delta \geq 9\rho Q\} \subseteq$$

$$\{u = \text{true} \wedge (p = \langle 0 \rangle \vee p = \langle 1, 0 \rangle \vee p = \langle 1 \rangle)\} ; \{u = \text{false}\},$$

which is derived from the encoding axioms, to Step 4 together with **Dec9e**, **Dec9o** and **Dec70**.

Step 6 was derived from 5. Requirement **Corr** is a simple consequence of 6.

5 Comparison of the Two Approaches

We have presented two entirely different approaches to the specification and verification of an audio control protocol with real-time characteristics.

The first approach is based on the Circal process algebra and allowed an un-timed verification of the protocol correctness. In this approach time was modelled in terms of discrete ticks and is used to implement the encoding and decoding of the message. The use of ticks often leads to a state explosion when generating the global timed behaviour of the system. A solution to this problem is to abstract time away before carrying out the verification. The relation between the local times of the sender and receiver is established by the $D_{s,r}$ process, which plays the rôle of a timing constraint [6].

In the second approach time plays a central rôle both in specifying and verifying the system. The great expressive power of TIC in modelling time allows the verification of an upper bound to the time needed to decode a given message.

It is interesting to notice that the modelling challenges are almost complementary in the two approaches. Modelling the sender is quite an easy task using the process algebra, due to the procedural nature of the sender. In TIC instead it

is necessary to define an infrastructure in which to represent the message encoding. There are many possible choices at this stage, none of them intuitive. The result is a complex construction where a simple behaviour is obscured within a declarative specification. On the other hand, the receiver is hard to model using Circal, whereas it is almost immediate in TIC.

The correctness proof is quite simple in Circal and is automatically performed using the Circal System. A graphical interface to the Circal model of the protocol has been constructed using Tcl/Tk [15]. The user interface allows the input of a sequence of messages and specification of relative clock timings both in the simulation and the formal verification of the protocol.

In TIC the correctness proof is very complex and was carried out by hand. However the most complex part of the proof is given by Step 1. This step of the proof establishes the correctness of the correspondence between encoding and decoding of single bits. All possible combinations of encodings and decodings must be exhaustively checked. Such a proof step cannot be fully automated. When performed with the help of a theorem prover it still requires a great amount of interaction with the user in constructing all the single cases.

The proof of Steps 3–6 is much shorter than the proof of just Step 1. It shows difficulties only when encoding and decoding axioms are used together, which happens at Step 5. The use of encoding and decoding axioms together is necessary to verify that eventually the input and the output sequences coincide. This is actually the untimed correctness verification that is fully automated in the process algebra approach. If we replace requirement **Corr** by

$$\textbf{Timing. } \{\alpha \geq (4(\#i) - 4 + 9\rho)Q\} \subseteq \{\alpha = o(\alpha)\}$$

we may verify in TIC only the timing, with the assumption that the actual correctness has been automatically verified in the process algebra framework. In TIC such a pure timing verification is much easier to be automated.

6 Conclusion

We have expressed the same requirement in two different formalisms, allowing their expressiveness and capabilities to be directly compared. The process algebra solution offered a simple, operational model which could be analysed automatically, but was unable to capture all of the necessary timing properties. The axiomatic solution offered an abstract, non-constructive model that captured all of the requirements, but required challenging proof techniques to analyse.

Perhaps the best approach, therefore, is to use different formalisms to analyse different properties. It is in general possible to decompose a property or requirement into subproperties, which may then be verified within different proof environments. We have shown this using a well-known case study. However we need to construct a different model of the system under analysis for each proof environment to be used. We must therefore ensure the consistency of all these models. A possible way is a formal translation between specification formalisms.

Acknowledgements I would like to thank Colin Fidge for helpful discussions and useful comments on this paper. This research has been supported by the Information Technology Division of the Defence Science Technology Organisation.

References

1. A. Bengtsson, W. Griffioen, K. Kristoffersen, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio control protocol with bus collision. In *8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*. Springer, 1996.
2. D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio control protocol. In *3rd School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFS'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 170–192. Springer, 1994.
3. A. Cerone. Axiomatisation of an interval calculus for theorem proving. Technical Report 05-00, Software Verification Research Centre, The University of Queensland, Brisbane, Australia, Jan 2000.
4. A. Cerone, A. J. Cowie, G. J. Milne, and P. A. Moseley. Description and verification of a time-sensitive protocol. Technical Report CIS-96-009, University of South Australia, Adelaide, Australia, 1996.
5. A. Cerone, A. J. Cowie, G. J. Milne, and P. A. Moseley. Modelling a time-dependent protocol using the Circal process algebra. In *Lecture Notes in Computer Science*, volume 1201 of *International Workshop on Hybrid and Real-Time Systems (HART'97)*, pages 124–138. Springer, 1997.
6. A. Cerone and G. J. Milne. Specification of timing constraints within the Circal process algebra. In *6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 1997.
7. L. Chen. Verification of an audio control protocol within real time process algebra. In *2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 70–77, Clearwater Beach, Florida, USA, March 1998.
8. C. Daws and S. Yovine. Two examples of verification of multirated timed automata with Kronos. In *7th 1995 IEEE Real-Time Systems Symposium*, Pisa, Italy, 1995. IEEE Comp. Soc.
9. C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhurst. A set-theoretic model for real-time specification and reasoning. In *Mathematics of Program Construction (MPC'98)*, volume 1422 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 1998.
10. W. Griffioen. Proof-checking an audio control protocol with LP. Technical Report CS-R9570, CWI, Department of Software Technology, Amsterdam, The Netherlands, Oct 1995.
11. P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In *7th International Conference on Computer-Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 381–394. Springer, 1995.
12. C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.
13. K. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, USA, 1995.

14. G. J. Milne. *Formal Specification and Verification of Digital Systems*. McGraw Hill, 1994.
15. Verification of a time-dependent protocol (web page).
http://www.acrc.unisa.edu.au/doc/circal/circal_protocol.html.

Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software

Mark Lawford^{1*}, Jeff McDougall², Peter Froebel³, and Greg Moum³

¹ Computing and Software, McMaster Univ., Hamilton, ON, Canada, L8S 4L7
lawford@mcmaster.ca

² JKM Software Technologies Inc., 17160 Keele Street, R.R.1, Kettleby, ON, L0G 1J0
jkm@pathcom.com

³ Ontario Power Generation, 700 University Ave., Toronto, ON, Canada M5G 1X6
peter.froebel, g.moum@ontariopowergeneration.com

Abstract. In this paper we describe how a functional version of the 4-variable model can be decomposed to improve its practical application to industrial software verification problems. An example is then used to illustrate the limitations of the functional model and motivate a modest extension of the 4-variable model to an 8-variable relational model. The 8-variable model is designed to allow the system requirements to be specified as functions with input and output tolerance relations, as is typically done in practice. The goal is to create a relational method of specification and verification that models engineering intuition and hence is easy to use and understand.

1 Introduction

The CANDU Computer Systems Engineering Centre of Excellence Standard for Software Engineering of Safety Critical Software [4] states the following as its first fundamental principle: “*The required behavior of the software shall be documented using mathematical functions in a notation which has well defined syntax and semantics.*” In order to achieve this, Ontario Power Generation Inc.¹ (OPGI) and Atomic Energy of Canada Limited (AECL) have jointly defined a detailed software development process to govern the specification, design and verification of safety critical software systems. The software development process results in the production of a coherent set of documents that allow for static analysis of the properties of the design described in the Software Design Description (SDD), comparing them against the requirements described in the Software Requirements Specification (SRS).

In this work we review how this functional verification has been done in practice [5]. We first describe how a functional version of the 4-variable model of

* Partially supported by NSERC grant 217249-99.

¹ OPGI is the electricity generation company created from Ontario Hydro.

[11] can be decomposed to facilitate tool support and to reduce the manual effort required to perform and document the specification and verification tasks. An example from [5] is then used to motivate the extension to a relational model. The relational model is necessary to rigorously account for tolerances that must normally be considered when trying to implement a system to meet an ideal system behavior.

We propose the 8-variable model, a modest extension of the 4-variable model that takes into consideration input and output tolerances while still permitting the use of functional requirements specification and design descriptions. The model formalizes the engineering practice of appealing to tolerances when necessary. By attempting to formalize the process, the authors hope to provide a sound basis for tool support of the entire verification process and provide opportunities for further applications of fundamental relational algebraic concepts.

Section 2 provides an overview of the basic concepts and notation required by the paper. Section 3 explains the (functional) Systematic Design Verification (SDV) procedure and its limitations regarding tolerances. These limitations motivate the 8-variable model of Section 4.

2 Preliminaries

Functions and relations are shown in italics (e.g., f , REQ). All sets of time series vectors from the 4- and 8-variable models are shown bold (e.g., $\mathbf{BM}$). All other mathematical terms are shown in italics (e.g., $bm \in \mathbf{BM}$).

For a set V_i , we will denote the *identity map on the set V_i* by id_{V_i} (i.e., $id_{V_i} : V_i \rightarrow V_i$ such that $v_i \mapsto v_i$). Given functions $f : V_1 \rightarrow V_2$ and $g : V_2 \rightarrow V_3$, we will use $g \circ f$ to denote *functional composition* (i.e., $g \circ f(v_1) = g(f(v_1))$). The *cross product* of functions $f : V_1 \rightarrow V_2$ and $f' : V'_1 \rightarrow V'_2$, defines a function $f \times f' : V_1 \times V'_1 \rightarrow V_2 \times V'_2$ such that $(v, v') \xrightarrow{f \times f'} (f(v), f'(v'))$.

It will also be convenient to consider the operation of *relational composition*. For $F \subseteq V_1 \times V_2$ and $G \subseteq V_2 \times V_3$, $F \bullet G = \{(v_1, v_3) \in V_1 \times V_3 \mid (\exists v_2 \in V_2) : (v_1, v_2) \in F \wedge (v_2, v_3) \in G\}$. Thus for the functions f and g as defined above, $g \circ f = f \bullet g$. A relation F is said to be *total* if $(\forall v_1 \in V_1)(\exists v_2 \in V_2) : (v_1, v_2) \in F$.

Denote the set of all equivalence relations on V by $Eq(V)$. Any function $f : V \rightarrow R$ induces an equivalence relation $\ker(f) \in Eq(V)$, the equivalence kernel of f , given by $(v_1, v_2) \in \ker(f)$ if and only if $f(v_1) = f(v_2)$. We define the standard partial order on equivalence relations as follows. Given equivalence relations $\theta_1, \theta_2 \in Eq(V)$, we say that θ_1 is a *refinement* of θ_2 , written $\theta_1 \leq \theta_2$, iff each cell (equivalence class) of θ_1 is a subset of a cell of θ_2 (i.e., $(v, v') \in \theta_1$ implies $(v, v') \in \theta_2$ for all $(v, v') \in V \times V$). We can now formally state a basic existence claim for functions that will be used later in the paper.

Claim. Given two functions with the same domain, $f : V_1 \rightarrow V_3$ and $g : V_1 \rightarrow V_2$, there exists $h : V_2 \rightarrow V_3$ such that $f = h \circ g$ iff $\ker(g) \leq \ker(f)$.

3 Functional Systematic Design Verification

This section provides an overview of the (functional) SDV procedure used in [5]. We highlight elements of the process, such as the decomposition of proof obligations, that facilitate tool support and reduce the effort required to perform and document the SDV tasks. Although the SDV procedure covers other types of verification problems, we will concentrate on the verification of simple functional properties that often compose the majority of system requirements. The reader is referred to [8] for the complete procedure. The section concludes with an example that illustrates the limitations of the functional approach.

3.1 SDV Procedure Overview

The software engineering process described here is based upon the *Standard for Software Engineering of Safety Critical Software* [4] that was jointly developed by OPGI and AECL. This standard requires that the software development and verification be broken down into series of tasks that result in the production of detailed documents at each stage. The software development stages relevant to this paper are governed by the Software Requirements Specification Procedure [3] and the Software Design Description Procedure [6]. These procedures respectively produce the Software Requirements Specification (SRS) and Software Design Description (SDD) documents. In addition to other methods, these documents make use of a form of Parnas' tabular representations of mathematical functions [2,10] to specify the software's behavior. Tables provide a mathematically precise notation (see [1] for the formal semantics) for the SRS and SDD in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike [5].

The underlying models of both the SRS and SDD are based upon Finite State Machines (FSM). The SDD adds to the SRS functionality the scheduling, maintainability, resource allocation, error handling, and implementation dependencies. The specification technique for defining the implementation is based on a virtual machine which will execute the source code which is to be implemented. The primary difference between this virtual machine and the FSM describing the SRS is that execution is not instantaneous, but takes a finite amount of time, and thus the order of execution must be specified to avoid race conditions. The SRS is produced by software experts with the help of domain experts. It is used by lead software developers to produce the SDD which is then used by all the developers to produce the actual source code.

The software engineering standard [4] requires that the SDD be formally verified against the SRS and then the code formally verified against the SDD to ensure that the implementation meets the requirements. These formal verifications are governed by the SDV Procedure and Systematic Code Verification (SCV) Procedure. For the purposes of this paper we will concentrate on the SDV process.

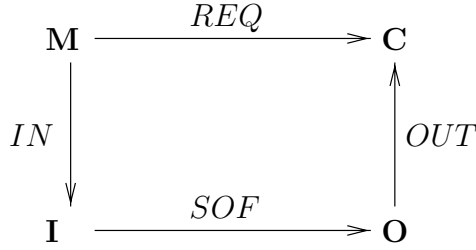


Fig. 1. Commutative diagram for 4 variable model

The objective of the SDV process is to verify, using mathematical techniques or rigorous arguments, that the behavior of every output defined in the SDD, is in compliance with the requirements for the behavior of that output as specified in the SRS. It is based upon a variation of the four variable model of [11] that verifies the functional equivalence of the SRS and SDD by comparing their respective one step transition functions. The resulting proof obligation in this special case:

$$REQ = OUT \circ SOF \circ IN \quad . \quad (1)$$

is illustrated in the commutative diagram of Figure 1. Here *REQ* represents the SRS state transition function mapping the monitored variables **M** (including the previous pass values of state variables) to the controlled variables and updated (current) state represented by **C**. The function *SOF* represents the SDD state transition function mapping the behavior of the implementation input variables represented by statespace **I** to the behavior of the software output variables represented by the statespace **O**. The mapping *IN* relates the specification's monitored variables to the implementation's input variables while the mapping *OUT* relates the implementation's output variables to the specification's controlled variables. The following section briefly outlines the refinement of the relational methods in [11] to the simple functional case in (1).

3.2 Specialization of the 4-Variable Model

In the 4-variable model of [11], each of the 4 “variable” state spaces **M**, **I**, **O**, and **C** is a set of functions of a single real valued argument that return a vector of values - one value for each of the quantities or “variables” associated with a particular dimension of the statespace. For instance, assuming that there are n_M monitored quantities, which we represent by the variables $m_1, m_2, \dots, m_{n_M}$, then, the possible timed behavior of the variable m_i can be represented as a function $m_i^t : \mathbb{R} \rightarrow Type(m_i)$, where $m_i^t(x)$ is the value of the quantity m_i at time x . We can then take **M** to be the set of all functions of the form $m^t(x) = (m_1^t(x), m_2^t(x), \dots, m_{n_M}^t(x))$. Thus the relations corresponding to the arrows of the commutative diagram then relate vectors of functions of a single real valued argument.

In order to simplify the 4-variable model to a FSM setting, we restrict ourselves to the case where each of the 4 “variables” $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set of “time series vectors”. For example, $\mathbf{M}$ actually refers to all possible sets of observations ordered (and equally spaced) in time, each observation being a vector of n_M values. We will use the term *monitored variable* to refer to quantity m_i which is the i th element in the vector ($i \in \{1, \dots, n_M\}$). Let $m \in \mathbf{M}$ be a time series vector of observations of the monitored variables. With a slight abuse of notation, we will use $m_i(z)$ to denote the z th observation of the i th element ($z \in \{0, 1, 2, \dots\}$) of the monitored variables for the time series vector m . Similarly $m(z)$ represents the z th observation of the n_M values in the monitored variable vector for time series m .

For this model, the time increment between each of the observations is defined to be the positive real value $\delta > 0$. Thus observation z corresponds to time $(z * \delta)$. The increment δ is taken to be at least an order of magnitude less than any time measurements of interest. The value of m_i at any point between two observations (i.e., in the range of time $[z * \delta, (z + 1) * \delta)$) is defined to be equal to $m_i(z)$.

Each of the “variables” $\mathbf{M}$, $\mathbf{C}$, $\mathbf{I}$, $\mathbf{O}$ in the specialized 4-variable model has the same frequency of observation, but may have a different number of values in its vector. The value n_M is defined to be the number of elements in $\mathbf{M}$, which are observed over time, while n_I is defined to be the number of elements in $\mathbf{I}$, which are observed over time. Normally $n_I = n_M$. Similarly n_O is defined to be the number of elements in $\mathbf{O}$, which are observed over time and n_C is defined to be the number of elements in $\mathbf{C}$, which are observed over time. Normally, $n_C = n_O$.

Requirements (REQ): The required behavior of the subsystem is described with *REQ*. At OPGI *REQ* is modeled as a FSM, defining a relation over $\mathbf{M} \times \mathbf{C}$. While, in general, the FSM could be nondeterministic, much of the system behavior can be modeled by a deterministic FSM with the result that for the verification of these properties we can assume that *REQ* is a function (i.e., *REQ* : $\mathbf{M} \rightarrow \mathbf{C}$).

In this case a new set of time series vectors, $\mathbf{S}$, is introduced to describe the state of the FSM. Let $c \in \mathbf{C}$, $m \in \mathbf{M}$, $s \in \mathbf{S}$, and $z \in \{0, 1, 2, \dots\}$. The z th value of a controlled variable time series vector $c(z)$ depends on both the values of $m(z)$ and $s(z)$, related by the vector function *OUTPUT* (i.e., $c(z) = \text{OUTPUT}(m(z), s(z))$). Also, the value $s(z + 1)$ depends on both the values of $m(z)$ and $s(z)$, related by the vector function *NEXTSTATE*. (i.e., $s(z + 1) = \text{NEXTSTATE}(m(z), s(z))$).

The SRS procedure [3] shows how a set of functions $f_1, f_2, \dots, f_j$ can be defined such that when a subset of them are composed, they define the *OUTPUT* function. When a different, though not necessarily disjoint, set of them are composed, they define the *NEXTSTATE* function. We have called the process of defining these functions the “decomposition of *REQ*”.

Design (SOF): The implemented behavior of the subsystem is described with *SOF*. *SOF* can be modeled as a directed graph with $p + 2$ nodes. Within this graph, each node is either one of p FSMs, or $\mathbf{I}$, or $\mathbf{O}$, and each edge represents

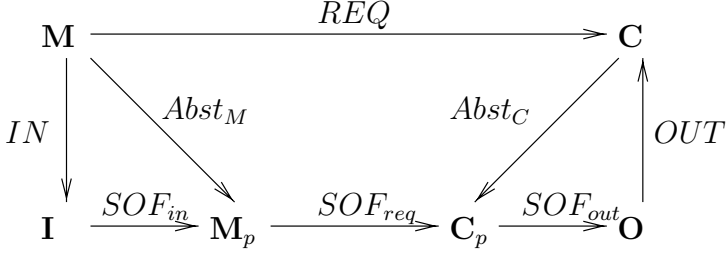


Fig. 2. Vertical decomposition: Isolation of hardware hiding proof obligations

data flow between two of these. The node containing I must not be the destination of any edge. The node containing O must not be the source of any edge. In this way, SOF defines a relation over $I \times O$. If the design is produced following the SDD procedure, then each of the FSMs represents a program called from the mainline. We assume a constant mainline loop structure, with each program called 1 or more times within the loop. If called more than once, calls are assumed to be evenly spaced within the loop. The loop is assumed to take a constant amount of time to execute.

For a large number of the implementation properties, the FSMs composing SOF can be modeled as deterministic FSM allowing us to consider the special case when SOF defines a function. In this case, when both REQ and SOF are functions, if we are also able to restrict ourselves to functional maps for IN and OUT , we can verify the commutative diagram in Figure 1 by comparing the one step transition functions of the FSMs defining REQ and SOF . More detailed descriptions of the underlying SRS and SDD models can be found in [3] and [6], respectively, as well as [8].

3.3 Decomposing the Proof Obligations

In Figure 2 we decompose the proof obligation (1) to isolate the verification of hardware interfaces. The M_p and C_p state spaces are the software's internal representation of the monitored and controlled variables, referred to as the *pseudo-monitored* and *pseudo-controlled variables*, respectively. The proof obligations associated with SDV then become

$$Abst_C \circ REQ = SOF_{req} \circ Abst_M \quad (2)$$

$$Abst_M = SOF_{in} \circ IN \quad (3)$$

$$id_C = OUT \circ SOF_{out} \circ Abst_C \quad (4)$$

The first of these equations represents a comparison of the functionality of the system and should contain most of the complexity of the system. The last two

represent comparisons of the hardware hiding software of the system. These obligations are often fairly straightforward and are discharged manually.

As an example to help the reader interpret the above decomposition, suppose an actual physical monitored plant parameter belonging to $\mathbf{M}$ is the temperature of the primary heat transport system which might have a current value of 500.3 Kelvin. The hardware corresponding to the temperature sensors and A/D converters might map this via IN to a value of 3.4 volts in a parameter in $\mathbf{I}$. A hardware hiding module might then process this input corresponding to the mapping SOF_{in} , producing a value of 500 Kelvin in the appropriate temperature variable belonging to the software state space $\mathbf{M}_p$. Further “vertical” decomposition is performed by isolating outputs and in effect restricting $\mathbf{M}$ and projecting $\mathbf{C}$ to the variables relevant to a particular subsystem such as the pressure sensor trip described in the Section 3.4.

The observant reader may have noted that the controlled variable abstraction function is defined as $Abst_C : \mathbf{C} \rightarrow \mathbf{C}_p$ which is seemingly the “wrong” direction. The proof obligation (4) forces $Abst_C$ to be invertible, preventing the possibility of trivial designs for SOF_{req} being used to satisfy the main obligation (2). As we will see below, this allows the verifier to define only one abstraction mapping for each pair of corresponding SRS and SDD state variables that occur as both inputs and outputs in the decomposition. The SDV procedure provides recourse for the case when there is not a 1-1 correspondence between $\mathbf{C}$ and $\mathbf{C}_p$ through the use of a pseudo-SRS that can be defined to more closely match the SDD. The interested reader is referred to [7] for further details.

Typically the verification of a subsystem as represented by (2), the inner part of the commutative diagram, can be decomposed “horizontally” at both the SRS and SDD level into a sequence of intermediate verification steps, thereby reducing the larger, more complex proof obligation into a number of smaller, more manageable verification tasks. This is represented in Figure 3 where each equality of the form:

$$SOF_i \circ Abst_{V_{i-1}} = Abst_{V_i} \circ REQ_i . \quad (5)$$

becomes a verification block. The price paid for this vertical and horizontal decomposition is that for each block the verifier must provide a cross reference between the internal variables making up the $\mathbf{V}_{i-1}, \mathbf{V}_i$ state spaces at the SRS level and the internal variables making up the $\mathbf{V}_{(i-1)p}, \mathbf{V}_{ip}$ state spaces at SDD level, as well as defining the abstraction functions, $Abst_{V_{i-1}}$ and $Abst_{V_i}$. Now the benefits of defining all the abstraction functions, including $Abst_C$, from top to bottom (SRS to SDD) in Figures 2 and 3 becomes more apparent. The values of many of the controlled variables from the previous execution pass of the SRS and SDD often become inputs to the calculation of current internal state and output variables. Similarly, state variables that are the output of one sequential block become the input of the following block. Defining all abstraction functions from top to bottom and then only performing the check for invertibility at the outputs embodied by (4) allows the verifier to use the same abstraction functions

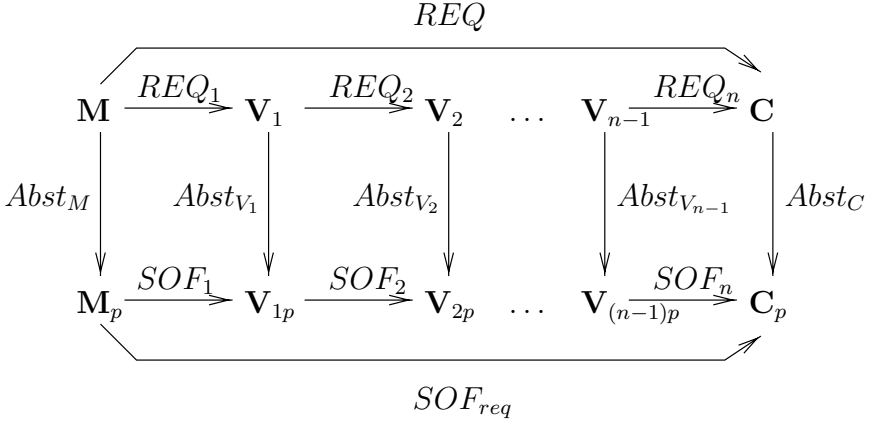


Fig. 3. Horizontal (sequential) decomposition of proof obligations

whether a state variable occurs at the input or output of a block. This technique reduces the number of abstraction functions required by up to one half.

3.4 Limitations of a Functional Model

The following example uses a simplified sensor trip to demonstrate how the verification task can be partitioned, and highlights the limitations of the functional SDV procedure regarding support for tolerances. Currently the verification tool suite described in [5] only supports functional verification. Work remains to be done on the incorporation of tolerances on inputs and outputs through the use of relational methods. Often SRS and SDD behaviors are not functionally equivalent, but they are within specified tolerances. Currently in these cases, separate rigorous arguments must be made, appealing to tolerances to explain any differences in functionality. Ideally, one would like to be able to use formal mathematical proofs incorporating the tolerances when necessary without an excessive increase in proof complexity and workload associated with the documentation. In many cases it should be possible to add existential quantifiers for variables with tolerances and then make minor modifications to the original theorem statements.

We now describe the verification of a simplified pressure sensor trip that monitors a pressure sensor and initiates a reactor shutdown when the sensor value exceeds a normal operating setpoint. We will use tabular specifications for their readability. In all of the tables in Figure 4, the functions return the value in the right column when the condition in the left column is satisfied.

Tables $f\_PressTrip$ and $PTRIP$ in Figure 4 give the proposed SRS and SDD implementations respectively for the sensor trip. The SRS specification of the pressure sensor trip makes use of deadbands to eliminate sensor chatter. In the

sentrip : THEORY

BEGIN

k_PressSP : int = 2450

k_DeadBand : int = 50

KDB : int = k_DeadBand

KPSP : int = k_PressSP

Trip : TYPE = {Tripped, NotTripped}

AI : TYPE = subrange(0, 5000)

f_PressTrip((Pressure : posreal), (f_PressTripS1 : Trip)) : Trip = TABLE

$\text{Pressure} \leq \text{k\_PressSP} - \text{k\_DeadBand}$	NotTripped
$\text{k\_PressSP} - \text{k\_DeadBand} < \text{Pressure} < \text{k\_PressSP}$	f_PressTripS1
$\text{Pressure} \geq \text{k\_PressSP}$	Tripped

ENDTABLE

PTRIP((PRES : AI), (PREV : bool)) : bool = TABLE

$\text{PRES} \leq \text{KPSP} - \text{KDB}$	FALSE
$\text{KPSP} - \text{KDB} < \text{PRES} \wedge \text{PRES} < \text{KPSP}$	PREV
$\text{PRES} \geq \text{KPSP}$	TRUE

ENDTABLE

Trip2bool((TripVal : Trip)) : bool = TABLE

TripVal = Tripped	TRUE
TripVal = NotTripped	FALSE

ENDTABLE

posreal2AI((x : posreal)) : AI = TABLE

$x \leq 0$	0
$0 < x \wedge x < 5000$	floor(x)
$x \geq 5000$	5000

ENDTABLE

Sentrip1 : THEOREM

$(\forall (\text{Pressure} : \text{posreal}, \text{f\_PressTripS1} : \text{Trip}) :$
 $\text{Trip2bool}(\text{f\_PressTrip}(\text{Pressure}, \text{f\_PressTripS1})) =$
 $\text{PTRIP}(\text{posreal2AI}(\text{Pressure}), \text{Trip2bool}(\text{f\_PressTripS1})))$

END sentrip

Fig. 4. Formatted PVS specification for pressure sensor trip example

function definitions, $f\_PressTripS1$ and $PREV$ play corresponding roles as the arguments for the previous value of the state variable computed by the function. The verification is performed using SRI's Prototype Verification System (PVS) automated proof assistant [9,12] to handle typechecking and proof details.

Figure 4 also contains the supporting type, constant and abstraction function definitions for the verification block. The abstraction function $posreal2AIType$ models the A/D conversion of the sensor values by taking the integer part of its input using the built in function $floor(x)$ from the PVS prelude file. It is used to map the real valued SRS input $Pressure$ to the discrete SDD input $PRES$ which has type $AIType$. $AIType$ consists of the subrange of integers between 0 and 5000, denoted by $subrange(0, 5000)$ in Figure 4.

At the bottom of the specification, the theorem $Sentrip1$ is an example of a *block comparison theorem* that is used to prove a specific instance of the general block verification equation (5) that relates the SRS and SDD inputs and outputs. If $Pressure$ and $PRES$ were both real numbers, related by the identity map, then the block comparison theorem $Sentrip1$ would be easily proved, but in this case, where $PRES$ is a discrete input, when attempting the block comparison, PVS reduces the problem to attempting to prove that for all input values the following equation holds:

$$\neg(f\_PressTripS1 = Tripped \wedge floor(Pressure) \leq 2400 \quad 2400 < Pressure < 2450) . \quad (6)$$

For any value of $Pressure$ in the open interval $(2400, 2401)$ when $f\_PressTrip$ was tripped in the previous pass, the above formula is *FALSE*. The problem occurs because whenever $2400 < Pressure < 2401$, the abstraction function $posreal2AIType$ maps $Pressure$ to the same value as 2400, but when $f\_PressTripS1 = Tripped$, the SRS function $f\_PressTrip$ maps $Pressure$ values greater than 2400 to *Tripped* while 2400 gets mapped to *NotTripped*. In other words,

$$\ker(posreal2AIType \times Trip2bool) \not\subseteq \ker(f\_PressTrip) . \quad (7)$$

so we know by the claim from Section 2 that there is no SDD design that can satisfy the block comparison theorem $Sentrip1$. The interested reader is referred to [5] for further details on the use of PVS in this example and the OPGI systematic design verification process in general.

This is an example of when functional equality is more strict than practically necessary. Due to the accuracy of the sensors and A/D hardware in the actual implementation, all input values have a tolerance of ± 5 units. In this case, the SDD function $PTRIP$ actually provides acceptable behavior.

The generalized relational version of the 4-variable model originally put forward in [11] easily handles this case by allowing REQ , IN and OUT from Figure 1 to be relations between state spaces consisting of sets of vectors of functions of time. We have found that the majority of system properties making

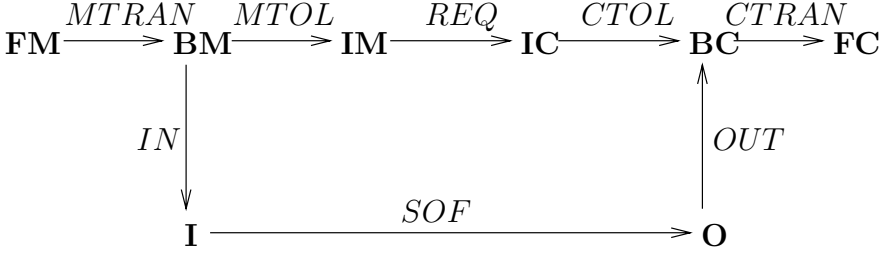


Fig. 5. 8-Variable Model

up *REQ* are, in practice, easily specified and verified if they are modeled as deterministic state machines where the *NEXTSTATE* and *OUTPUT* functions have tolerances associated with their inputs and outputs as outlined below.

4 The 8-Variable Model

When trying to compare the ideal required behavior to a design, allowable tolerances must be specified precisely so that it can be determined whether a proposed design exhibits acceptable behavior. The 8-variable model shown in Fig. 5 has been developed to take tolerances into account in such a way that the ideal required behavior from the functional 4-variable model of Section 3.2 needs no modification. Thus domain experts still specify the behavior functionally as *REQ*, with tolerances embodied by the *MTOL* and *CTOL* (i.e., *REQ* is a function) while *MTOL* and *CTOL* are tolerance relations. To facilitate the specification of tolerance relations, the monitored and controlled state spaces **M** and **C** of the original 4-variable model are replaced by triples of state spaces **FM**, **BM**, **IM** and **IC**, **BC**, **FC** respectively.

In this model, **FM** refers to the *Field Monitored Variables*. These are mathematical variables² which model properties of the environment that are being measured. For example, the heat transport temperature in degrees C might be modeled by a field monitored variable.

BM refers to the *Boundary Monitored Variables*. These are mathematical variables which model properties being measured at the limits of the subsystem being described. For example, if the subsystem was a trip computer, a voltage at the terminal blocks of the computer could be modeled by a boundary monitored variable. Although the required behavior of the subsystem only needs to describe behavior relative to boundary monitored variable values, it is convenient for the specifier writing the requirements to give each boundary monitored variable a name which reflects the property being modeled by the associated field monitored variable. For example, the SRS may name a boundary monitored variable

² As opposed to software variables.

$m\_HT\_Temperature$, even though it is a voltage value which is being modeled at the boundary. This allows the requirements specifier to concentrate on the problem domain when describing the required behavior of the subsystem. Typically there is a functional relationship *MTRAN* between **FM** and **BM**.

IM refers to the *Inner Monitored Variables*. Inner monitored variables are mathematical variables which represent the boundary monitored variables, with Monitored Variable Accuracy taken into account.

Monitored variable accuracy describes a range of values, such that the subsystem is required to respond as described to at least one of the values within that range. For example, if in the example above the boundary monitored variable accuracy is ± 0.5 V, and the value of the boundary monitored variable at some point in time is 2.5 V, then the requirements are saying that the subsystem may respond as if the value at that time is any one of the values in the range [2.0, 3.0] V. In this way, monitored variable accuracy results in the SRS describing a set of allowable behaviors. Any design which exhibits one of the allowable behaviors, meets the requirements.

The ideal required behavior in the SRS (i.e., *REQ*), when applied to the inner monitored variables, provides a description of required behavior which accounts for decisions regarding accuracy. For example, the requirement specifier can use a condition $m\_HT\_Temperature > m\_HT\_Setpoint$, rather than explaining within the condition how to account for the accuracy of the variable $m\_HT\_Temperature$. Note that in some cases, **IM** may be the same as **BM**. For example, in a trip computer a configuration EPROM value does not change on-line and is represented as a digital value. Thus the accuracy would be ± 0 . The *MTOL* relation between **BM** and **IM** is typically not a functional relationship, since one value in a boundary monitored variable is related to many values in the corresponding inner monitored variable when there is a non-zero monitored variable accuracy associated with the variable.

I refers to the *Input Variables*. Input variables are mathematical variables which model the information available to the software. For example, a voltage which is converted to a digital value via an A/D converter may be made available to the software as a base-2 integer in a register. The value read from that register could be modeled as an input variable. The *IN* relation between **BM** and **I** is usually not a functional relationship. This relation takes into account quantization of values (i.e., loss of accuracy due to constructing a discrete representation of a continuous quantity) and hardware inaccuracies (e.g., an A/D converter tolerance).

O refers to the *Output Variables*. Output variables are mathematical variables which model the values set by the software. For example, if the software sets a bit in a register to indicate that a trip should occur, the bit could be modeled as an output variable.

IC refers to the *Inner Controlled Variables*. Inner controlled variables are mathematical variables which represent the boundary controlled variables, before Controlled Variable Accuracy is taken into account.

Controlled variable accuracy describes a range of values, such that the response of the subsystem is required to equal a value which is within that range around the value described by the ideal required behavior. For example, if a boundary controlled variable accuracy is ± 0.1 V, and the ideal required value of the controlled variable at a point in time is 2.1 V, then the requirements are saying that the subsystem may respond at that time with a result which is any one of the values in the range $[2.0, 2.2]$ V. As with monitored variable accuracy, controlled variable accuracy results in the SRS describing a set of allowable behaviors. Any design which exhibits one of the behaviors allowed is thereby meeting the requirements.

BC refers to the *Boundary Controlled Variables*. These are mathematical variables which model properties being controlled at the limits of the subsystem being described. For example, a voltage produced by the subsystem at the terminal blocks of the computer could be modeled by a boundary controlled variable. Note that, as with boundary monitored variables, it is convenient for the specifier writing the requirements to give each boundary controlled variable a name which reflects the property being modeled by the associated field controlled variable. The *CTOL* relation between **IC** and **BC** is not functional since one value in an inner controlled variable can be related to many values in the corresponding boundary controlled variable. The *OUT* relation between **O** and **BC** is not typically functional since it takes into account hardware inaccuracies (e.g., D/A converter tolerance).

FC refers to the *Field Controlled Variables*. These are mathematical variables which model properties of the environment that are being controlled. Typically there is a functional relationship *CTRAN* between **BC** and **FC**. If **BC** and **FC** are not the same variables, then the system level documentation should describe the transformation between them.

Collectively, the variables **FM**, **BM** and **IM** will be referred to as Monitored Variables. Similarly, **IC**, **BC** and **FC** will be referred to as Controlled Variables.

Monitored Variable Accuracy: If each boundary monitored variable bm_i , has an accuracy requirement $+a_i/-b_i$, where $a_i \geq 0$ and $b_i \geq 0$, then *MTOL* defines a relation over **BM** $\times$ **IM** such that $(bm, im) \in MTOL \iff$

$$(\forall z \in \{0, 1, \dots\})(\forall i \in \{1, \dots, n_M\})bm_i(z) - b_i \leq im_i(z) \leq bm_i(z) + a_i \quad . \quad (8)$$

Controlled Variable Accuracy: If each boundary controlled variable bc_i , has an accuracy requirement $+c_i/-d_i$, where $c_i \geq 0$ and $d_i \geq 0$, then *CTOL* defines a relation over **IC** $\times$ **BC** such that $(ic, bc) \in CTOL \iff$

$$(\forall z \in \{0, 1, \dots\})(\forall i \in \{1, \dots, n_C\})(ic_i(z) - d_i \leq bc_i(z) \leq ic_i(z) + c_i) \quad . \quad (9)$$

4.1 Design Verification

From Fig. 5, there are two “paths” from **BM** to **BC**. The first path is via **IM** and **IC**, with *MTOL*, *REQ*, and *CTOL*. The second path is via **I** and **O**, with *IN*, *SOF*, and *OUT*. More precisely, when *MTOL*, *REQ* and *CTOL* are composed, they describe the *REQUIREMENTS* relation which is the subset of **BM** $\times$ **BC** given by: $REQUIREMENTS = MTOL \bullet REQ \bullet CTOL$.

Similarly, when *IN*, *SOF* and *OUT* are composed, they describe the relation $DESIGN = IN \bullet SOF \bullet OUT \subseteq \mathbf{BM} \times \mathbf{BC}$.

Design verification of functional requirements with tolerances is then the process of showing two things:

$$DESIGN \subseteq REQUIREMENTS, \text{ and} \quad (10)$$

$$DESIGN \text{ is total.} \quad (11)$$

Thus, by (10), all behaviors that the design may exhibit represent acceptable behavior according to the requirements, and by (11) the design is defined for all possible values of boundary monitored variables. Note that together conditions (10) and (11) guarantee that *REQUIREMENTS* is total so that the acceptable behavior of the system has been completely specified for all possible monitored variable values.

It may be that the designer does not want or need *REQUIREMENTS* to be complete (e.g., in case when there are input combinations that are physically impossible). This happens when the system being controlled, the “plant”, places restrictions on how the monitored variables can be related to the controlled variables. In the standard 4-variable model of [11] this is modeled by the relation $NAT \subseteq \mathbf{M} \times \mathbf{C}$. In the case of the proposed 8-variable model we could have $NAT \subseteq \mathbf{BM} \times \mathbf{BC}$. In this case (11) could be replaced by the requirement:

$$NAT \cap REQUIREMENTS \subseteq DESIGN . \quad (12)$$

to guarantee that (10) being met by a non-trivial design.

4.2 The Simplified Sensor Trip Revisited

For the simple sensor trip example in Section 3.4, *MTRAN* can be used to define the mapping from the physical pressure in kPa to a real valued sensor output voltage while *CTRAN* relates the *Tripped/NotTripped* value of *f_PressTrip* to the *Open/Closed* state of a physical relay. The ± 5 tolerance on the input due to the uncertainty in the sensors and A/D conversion is modeled by *MTOL* and *CTOL* is just the identity map due to the discrete nature of the trip output. The remaining maps *REQ*, *SOF*, *IN* and *OUT* are still modeled by the functions *f_PressTrip*, *PTRIP*, *posreal2AI* and the inverse of *Trip2bool*, respectively.

Using PVS’s dependent typing capabilities, the block comparison theorem *Sentrip1* can be restated as the easily proved theorem:

Sentrip1 : THEOREM

$$\begin{aligned}
 &(\forall (\text{Pressure} : \text{posreal}, \text{f\_PressTripS1} : \text{Trip}) \\
 &\quad (\exists (\text{Pressure2} : \{(x : \text{posreal}) | \text{Pressure} - 5 \leq x \leq \text{Pressure} + 5\}) : \\
 &\quad \text{Trip2bool}(\text{f\_PressTrip}(\text{Pressure2}, \text{f\_PressTripS1})) = \\
 &\quad \text{PTRIP}(\text{posreal2AI}(\text{Pressure}), \text{Trip2bool}(\text{f\_PressTripS1}))))
 \end{aligned}$$

5 Conclusion

The main goals of this paper have been to provide insight into how relational methods can be adapted to increase their utility in practical applications. We have outlined a functional specification and verification technique for safety critical software based upon the 4-variable model of [11]. A simple example was used to illustrate the limitations of a functional model and motivate a modest extension of the theory to a relational setting that we call the 8-variable model. The main benefit of this model refinement of the relational 4-variable model is that the method is intuitive and easy to use for both requirements specification and design description since engineers typically prefer to think in terms of functions with tolerances when dealing with safety critical systems.

Acknowledgements: The authors would like to thank Mike Viola of OPGI for his comments and constructive criticism regarding early drafts of this paper.

References

1. R. Janicki and R. Khédri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 2000. To appear.
2. R. Janicki, D. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink et al., editors, *Relational Methods in Computer Science*, Advances in Computing Science, ch. 12, p. 184–196. Springer Wien NewYork, 1997.
3. E. Jankowski and J. McDougall. Procedure for the Specification of Software Requirements for Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1001-PROC Rev. 1, July 1995.
4. P. Joannou *et al.* Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1, January 1995.
5. M. Lawford and P. Froebel. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Submitted to FMSD.
6. J. McDougall and J. Lee. Procedure for the Software Design Description for Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1002-PROC Rev. 1, October 1995.
7. J. McDougall, M. Viola, and G. Moum. Tabular representation of mathematical functions for the specification and verification of safety critical software. In *SAFE-COMP'94*, p. 21–30, Anaheim, October 1994. Instrument Society of America.
8. G. Moum. Procedure for the Systematic Design Verification of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Procedure CE-1003-PROC Rev. 1, December 1997.

9. Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *TACAS '97*, LNCS 1217, p. 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
10. D. Parnas. Tabular representation of relations. Technical Report 260, Communications Research Laboratory, McMaster University, October 1992.
11. D. Parnas and J. Madey. Functional documentation for computer systems engineering. Technical Report CRL No. 273, Telecommunications Research Institute of Ontario, McMaster University, September 1991.
12. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

Algebraic State Machines

Manfred Broy¹, Martin Wirsing²

¹ Institut für Informatik, Technische Universität München
80290 München, Germany
email:broy@informatik.tu-muenchen.de

² Institut für Informatik, Ludwig-Maximilians-Universität München
80538 München, Germany
email:wirsing@informatik.uni-muenchen.de

Abstract. We introduce the concept of an *algebraic state machine*. This is a state transition machine all parts of that are described by algebraic and logical means. This way we base the description of state transition systems exclusively on the concept of algebraic specifications. Also the state of an algebraic state machine is represented by an algebra. In particular, we describe the state spaces of the state machine by algebraic techniques, and the state transitions by special axioms called *transition rules*. Then we show how known concepts from algebraic specifications can be used to provide a notion of parallel composition with asynchronous interaction for algebraic state machines. As example we introduce a notion of object-oriented component and show how algebraic state machines can formalize such components.

1 Introduction

State machines are a useful concept for describing the dynamic behavior of systems. Many specification formalisms have been developed based on the concept state machines such as classical automata, I/O-machines [24], state charts [20] but also UNITY [12] or TLA [23]. One of the problems with using state machines for describing complex systems is the large number of states that these systems have and thus must be specified. One way to avoid this problem is abstract away from all the states to a smaller number of higher level states. Many properties of the concrete state machine can then be formulated at and inferred from the higher-level model.

In recent years, *abstract state machines* (formerly called evolving algebras) have been developed explicitly for the purpose of abstracting from the concrete states (see e.g. [6], [18, 19]). Abstract state machines have been successfully applied for modeling e.g. concurrent algorithms and the operational semantics of programming languages [19].

One of the central ideas of abstract state machines is the use of algebras as the elements of the state space¹. On these state spaces the state transitions are defined by a general form of assignment assigning a new algebra as interpretation to the signature such that in fact high level programs are written. Keeping this in mind abstract state machines are rather an abstract programming method than a specification method. In contrast, we are interested in an axiomatic specification technique in the following. Therefore we find it suggestive to work with algebraic specifications to specify the algebras of the state space.

Algebraic specifications are a classical logical concept for axiomatising algebras. In this paper we show how the idea of algebras as the states of a state machine can be very straightforwardly supported by algebraic specifications. By this, the enormous work in the area of algebraic specifications (see, for instance, [31], [3]) is immediately available to support the specification of the state machines with algebras as states. We speak in the following of *algebraic state machines*.

When dealing with large complex systems, it is not always appropriate to describe them by one large state transition system. In many cases it is more adequate to describe the system as composed of a number of smaller components that work in parallel and cooperate and communicate to achieve the required behavior. This suggests looking for a notion of composition of components that themselves may be described by state transition machines or again are decomposed into a system of cooperating subcomponents. For this purpose we need clear concepts of composition, cooperation, and communication. We work with a concept of (asynchronous) communication over channels. State transitions consume input from the input channels and produce output on the output channels. They can be abstracted into stream processing functions for which well-understood composition concepts are available. As an example for the power and flexibility of algebraic state machines we introduce a simple notion of object-oriented component defined by algebraic state machines over a class diagram.

Along these lines we unify the idea of algebraic specifications and stream processing into networks of algebraic state machines. The main objective of this paper is to integrate the idea of algebras as state smoothly into well-understood concepts like algebraic specifications and stream processing components.

The paper is organized as follows. In section 2 we introduce some basic notions of algebraic specifications such as signatures and algebras. In section 3 we introduce the basic idea of algebraic state machines. In section 4 we describe the syntactic and semantic structure of specifications for algebraic state machines. In section 5 we study three examples including a simple object model and class diagrams and discuss issues of structuring the state space. In section 6 we show how to carry over known ideas of abstraction and composition to algebraic state machines. In particular we define asynchronous parallel composition with feedback as a general basis for composing algebraic state machines. In section 7 we compare our approach with related work. In section 8 we conclude with a brief discussion on the significance of algebraic state machines.

¹ Actually this idea is not new; See, for instance, [15] or [16].

2 Signatures and Algebras

Each model of a reactive system is based on a data model. Data models are used to represent the information and data structures needed to model the relevant aspects of an application domain and to provide the computing structures needed for representing the internal data of information processing systems. In general, most of the common data models capture mainly the structure of the data and their relationship, but not their characteristic operations. These, of course, should be an integral part of every data model. Along these lines we understand a data model always as family of data sets named by sorts together with their relationships and the basic characteristic operations and functions.

From a mathematical point of view, a data model is a mathematical structure called a *heterogeneous algebra*. Such an algebra consists of a family of carrier sets and a family of functions each of which is named by an identifier. More technically, we assume a set S of *sorts*² (often also called *data types* or *modes*) and a set F of symbols for constants including function symbols with a fixed sort called the functionality of the constant determined by the mapping

$$\mathbf{fct} : F \rightarrow S$$

The function **fct** associates with every symbol in the set F its sort. In the case of function symbols these are functional sorts that determine the domain sorts and the range sort of the function. We assume that the set of sorts S contains, besides basic sorts (such as *Bool*, *Nat*, etc.), also tuples of sorts (records) as well as *functional* sorts and even *polymorphic* sorts (see [27] for details).

Both the sets S and F provide only names. The pair $\Sigma = (S, F)$ together with the mapping **fct** that assigns sorts (also called functionalities) to the identifiers in F is often called the *signature* of the algebra. The signature is the static part of a data model and provides a syntactic view onto the data model.

An *algebra* of the signature Σ , called a Σ -*algebra*, contains carrier sets for the sorts and elements for the function symbols. In every algebra A of the signature $\Sigma = (S, F)$ we associate

- with every sort $s \in S$ a carrier set s^A (a set of data elements) and
- with every function symbol $f \in F$ a constant or function f^A of the requested sort or functionality.

An algebra gives meaning to a signature. It is the interpretation of a signature and can be seen as the semantic context of a data model.

An *algebraic specification* describes a class of algebras in a property-oriented axiomatic way. A specification $\text{SPEC} = (\Sigma, \text{AX})$ consists of a signature Σ and a set of laws AX ; the elements of AX are predicate logic formulas (with equality) over the signature Σ . An algebra is a *model* of SP if it has signature Σ and satisfies all laws in

² We believe, like many other computing scientists and software engineers that data sorts (typing) are a very helpful concept in modelling application and software structures.

AX. Structuring of specifications is achieved by so-called specification-building operators, which allow one to extend, rename and combine specifications in a compositional way (see [31, 27]).

3 Algebraic State Machines

A *state machine* with input and output consists of a state space State (a set of states), a set of input actions A_{in} , a set of output actions A_{out} , a (possibly non-deterministic) state transition function

$$\Delta: \text{State} \times A_{\text{in}} \rightarrow \wp(\text{State} \times A_{\text{out}})$$

and, last but not least, a set of initial states $\text{State}_0 \subseteq \text{State}$. Here, $\wp(M)$ denotes the power set over a set M .

In our case we consider state machines that we connect by channels to their environment. Every channel has assigned a sort that characterizes the sort of messages that are sent along the channel. The set of input channels is denoted by I and the set of output channels is denoted by O . By $I^{[*]}$ we denote the set of valuations of the channels in I by finite communication histories. These are the mappings

$$v: I \rightarrow M^*$$

where M is the set of messages and M^* is the set of finite sequences of messages. Of course, for every channel valuation v we require that for every channel $c \in I$ the sequence $v.c$ is a sequence of elements of sort M_c where M_c denotes the sort of the channel c .

An algebraic state machine is a state machine where the state is an algebra. Given a signature³ Σ and a set Alg of Σ -algebras, an *algebraic state machine* is given by a state transition function

$$\Delta: \text{Alg} \times I^{[*]} \rightarrow \wp(\text{Alg} \times O^{[*]})$$

In addition we assume a set $\text{Alg}_0 \subseteq \text{Alg}$ of algebras that are the initial states of the state machines.

We describe algebraic state machines by techniques of algebraic specification. Loose algebraic specifications describe sets⁴ of algebras and therefore are very well-suited for the description of the state spaces of algebraic state machines, which are exactly sets of algebras.

³ In abstract state machines even the signature of the algebras in the state space may vary when performing state transitions. We do not consider this as an important feature since a change of the elements of a signature can always be encoded in a class of algebras with an universal signature. Note that the set of symbols used in a textual description (which is always of finite length) is always finite, anyhow.

⁴ Formally speaking, the models form a class (in the mathematical sense) not a set. This is due to a foundational problem of axiomatic set theory which is, however, not important in our context.

4 Specification of Algebraic State Machines

To describe algebraic state machines by logical axioms, we define a hierarchical signature structured in four layers including four sets of axioms. The signature defines the function symbols and sorts. The layers split the signature into *static* and *dynamic* parts. The static parts of the signature are the sorts and functions of the context specification that are used to form the state machine with the same interpretation in each state. In other words, the static parts never change in state transitions. They represent invariants of the system.

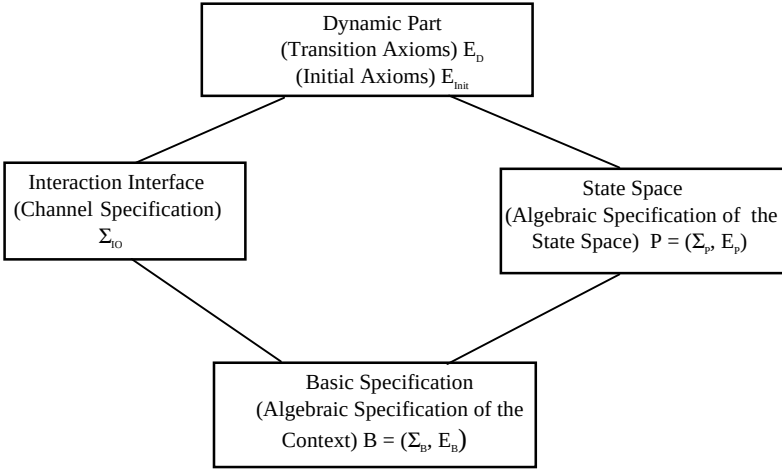


Fig. 1. Structure of the Specification Hierarchy of an Algebraic State Machine

The specification of an algebraic state machine consists, in particular, of the following constituents (cf. Figure 1):

- An algebraic specification $B = (\Sigma_B, E_B)$ called the *basic specification* or *the context*; the context defines the fundamental sorts and functions on which the channels with their sorts and the dynamic part of the description of the algebraic state machine are based. It defines all the sorts and function symbols that we assume to be given in the environment in which the specification is used, and on which the specification is hierarchically based.
- An *interface signature* Σ_{IO} defining a set of input and a set of output channels, together with their sorts; the sorts are required to be introduced in the context specification B .

- An algebraic specification called the *state space* that defines the state space by a specification $P = (\Sigma_p, E_p)$. It consists of a signature and a number of axioms. The state space of the associated state machine is the set of models associated with this algebraic specification. The state space specification is hierarchically based on the basic specification; moreover, it contains those axioms that are invariantly true for all algebras in the state space and dynamic function symbols whose interpretation may vary in the different states. P is supposed to be a *conservative extension* of the basic specification B (see [31]).
- A set of axioms E_{Init} that describe the properties of the initial states in addition to the static properties given by the state space specification P that hold for all algebras that are members of the state space.
- A set of transition axioms E_d of the form

$$\{\text{Pre}\} i_1:E_1, \dots, i_k:E_k / o_1:G_1, \dots, o_j:G_j \{\text{Post}\}$$

where

- Pre is a logical formula over the state space, called the *precondition*,
- $i_1, \dots, i_k$ are input channels, and $E_1, \dots, E_k$ are terms over the interface signature denoting sequences of the sort of the respective channel; together with the channels this is called the *input pattern*,
- $o_1, \dots, o_j$ are output channels, and $G_1, \dots, G_j$ are terms over the interface signature denoting sequences of the sort of the respective channel; together with the channels this is called the *output pattern*,
- Post is a logical formula over the signature of the state space including the (declared) identifiers of the state space also in a primed form; it is called the *postcondition*.

The meaning of such an algebraic specification of a transition machine with input and output is rather straightforward.⁵ The specification P defines a set of algebras that include the state space. These are those algebras of a signature that comprise the sorts and function symbols of the context as well as the sorts and function symbols of the state space and that satisfy the axioms of these two specifications. The interface signature defines $k+j$ constants corresponding to the input and output messages on the channels. Given any model of P the values of the constants are computed by interpreting the terms $E_1, \dots, E_k, G_1, \dots, G_j$.

The transition function of the state machine is defined by the transition axioms. The transition axioms define a logical relation between input and output messages involved in a transition as well as the given state and the successor state. Both the

⁵ In section 6 we will refine this semantics. By abstracting from the state space we will associate a black box view with algebraic state machines which allows us to describe the cooperation of algebraic state machines.

state A and the successor state A' are represented by algebras which share a model of the context as common subalgebra. The primed function symbols refer to the functions of the successor state.

The algebraic specification of a state machine should be structured into the layers given in Fig. 1 forming a proper hierarchy. Then they fulfil the following logical requirements.

- The state space axioms do not impose further logical properties onto the basic specification; the same holds for the axioms describing the initial state. Technically speaking, all layers shown in Fig. 1 except the top one are conservative extensions of the layers on which they are based.

Of course what we have described so far is not a fully formal semantics but rather a sketch. However, to give such a semantics is only an extended exercise in the semantics of algebraic specification and could be given as in [5] or [17]. Before we discuss the cooperation of algebraic state machines in networks we illustrate our approach by a few simple examples.

For the syntax our specifications we choose the specification language SPECTRUM [27]. In contrast to other algebraic specification languages such as CASL [34] it has the advantage to offer polymorphic sorts and functions.

5 Examples

In this section we will study three examples: specifications of algebraic state machines for interactive sets, for a simple object model and for class diagrams. In each case we will give two specifications: one based on a classical algebraic specification of finite sets and stores, resp., and one more in the spirit of abstract state machines with (dynamic) functions as state attributes.

5.1 Interactive Sets

A simple example of a system that can well be modeled by a state transition system is a component that implements an interactive set in an object-oriented style. This is an encapsulated set that can be manipulated by message exchange, only. We start with the basic specification. It defines in a polymorphic style the algebra of finite sets.

SPEC SET =

{ **sort** Set α ;

$\emptyset$:	Set α ;	<i>empty set</i>
$\_ \cup \{ \_ \}, \_ \setminus \{ \_ \}$:	Set $\alpha, \alpha \rightarrow \text{Set } \alpha$;	<i>add, delete an element</i>
$\_ = \emptyset$:	Set $\alpha \rightarrow \text{Bool}$;	<i>test for empty set</i>
$\_ \in \_$:	$\alpha, \text{Set } \alpha \rightarrow \text{Bool}$;	<i>element test</i>

Set α **generated_by** $\emptyset, \cup\{\}$; *generation axiom for finite sets*

$\emptyset = \emptyset$; $\neg(\text{set} \cup \{x\} = \emptyset)$;

$\neg(x \in \emptyset)$; $x \in \text{set} \cup \{x\}$;

$x \in \text{set} \Rightarrow x \in \text{set} \cup \{y\}$; $x \neq y \Rightarrow x \in \text{set} \cup \{y\} = x \in \text{set}$;

$\emptyset \setminus \{x\} = \emptyset$;

$(\text{set} \cup \{x\}) \setminus \{x\} = \text{set} \setminus \{x\}$; $x \neq y \Rightarrow (\text{set} \cup \{y\}) \setminus \{x\} = (\text{set} \setminus \{x\}) \cup \{y\}$;

$(\text{set} \cup \{x\}) \cup \{y\} = (\text{set} \cup \{y\}) \cup \{x\}$; $x \in \text{set} \Rightarrow \text{set} \cup \{x\} = \text{set}$;

}

This is a classical algebraic specification as it is well-known by now. We can form a state machine on top of this specification using the following specification of the messages arriving on the input channel (strictly speaking we have to introduce besides the constructors also selectors, but for sake of brevity we omit them here):

SPEC INPUT = { **sort** In α = $\text{iselem}(\alpha) \mid \text{add}(\alpha) \mid \text{sub}(\alpha) \mid \text{empty}$ }

The messages on the output channel are of the sort Bool. The following state machine has a very simple dynamic part with a state attribute s of sort Set α . It makes only a restricted use of the concept of algebras as states.

SPEC SetASM =

{ **based_on** SET, INPUT;

interface: in: In α **input channel**;

 out: Bool **output channel**;

state: s: Set α ;

initial: $s = \emptyset$;

dynamic: {true} in: empty / - $\{s' = \emptyset\}$;

 {true} in: iselem(x) / out: $(x \in s)$ $\{s' = s\}$;

 {true} in: add(x) / - $\{s' = s \cup \{x\}\}$;

 {true} in: sub(x) / - $\{s' = s \setminus \{x\}\}$;

}

In the following, we introduce some abbreviations for frame axioms which ensure that the dynamic elements of the state space remain (almost) unchanged (see e.g. [35] for a discussion. For any $f: T \rightarrow R$, $f_1: T_1 \rightarrow R_1, \dots, f_n: T_n \rightarrow R_n$, $a: T, a_1: T_1, \dots, a_k: T_k$ we define:

f remains unchanged: $\text{SAME}(f) \text{ for } \forall x \in T. f'(x) = f(x)$ *f remains unchanged except for a:* $\text{SAME}(f-a) \text{ for } \forall x \in T. x \neq a \Rightarrow f'(x) = f(x)$
 $f_1, \dots, f_n$ remain unchanged except for $a_1, \dots, a_k, k \leq n$:
 $\text{SAME}(f_1-a_1, \dots, f_k-a_k, f_{k+1}, \dots, f_n) \text{ for}$
 $\text{SAME}(f_1-a_1) \wedge \dots \wedge \text{SAME}(f_k-a_k) \wedge \text{SAME}(f_{k+1}) \wedge \dots \wedge \text{SAME}(f_n)$

We now give a specification that makes more extensive use of the characteristics of algebraic specifications and algebraic state machines by choosing a function as state attribute. To achieve this we include the specification of sets into the dynamic part (and therefore we do not need SET as part of the base).

SPEC SetASM1 =

```

{ based_on   INPUT;

  interface   in:      In  $\alpha$    input channel;
               out:     Bool   output channel;

  state:      isel:     $\alpha \rightarrow \text{Bool}$ ;

  initial:    isel(a) = false;

  dynamic:    {true} in: empty      / -           {  $\forall a \in \alpha: \text{isel}'(a) = \text{false}$  };
               {true} in: iselem(x) / out: isel(x) { SAME(isel) };
               {true} in: add(x)    / -           {  $\text{isel}'(x) = \text{true} \wedge \text{SAME}(\text{isel}-x)$  };
               {true} in: sub(x)    / -           {  $\text{isel}'(x) = \text{false} \wedge \text{SAME}(\text{isel}-x)$  };
}

```

Note that here the specification SET has become implicit. We have merged the specification of sets and that of the transition axioms. Formally each state of the state machine is a characteristic function *isel* which corresponds to a set. Therefore the notion of set is not explicitly present.

Which of the two specifications SetASM and SetASM1 we consider as being better structured is an interesting question. In SetASM1 the fact that the state machine implements a set is rather implicit. The laws of interaction and of data manipulation are combined while in SetASM they are properly separated. The generation principle is replaced by the principle of reachability. Even, if the sort α is infinite, only those states can be reached where *isel* is true for a finite set of elements of sort α . A proof of this requires a reachability analysis or a proof using an invariant technique.

Comparing SetASM and SetASM1 semantically, we see that any model of the state specification of SetASM can be extended in a unique way by a function *isel* and vice versa any model of the state specification of SetASM1 can be extended by an appropriate interpretation of the sort Set and the state attribute *s*. Thus the state specifications of SetASM and SetASM1 implement each other (in the form of an Forget-Identify implementation relation, see e.g. [31]). These implementation relations extend directly to the axioms for initial states and all parts of the transition

axioms. Thus SetASM and SetASM1 are equivalent w.r.t. the Forget-Identify implementation relation.

5.2 A Simple Object Model

As a more advanced example we define a simple object model by algebraic techniques. The specification OBJECT_MODEL describes for given classes represented by the sort α and given attributes for each class (for each sort α) the object model in a polymorphic style. The sort Store denotes the set of object stores and the set Obj α denotes the set of object identifiers (references) for the class α . More precisely, α is a record sort of the attributes of the class.

SPEC OBJECT_MODEL =

```
{ sort           Store, Obj  $\alpha$ ;
  emptyStore: Store;           empty store
  update:      Store, Obj  $\alpha$ ,  $\alpha \rightarrow$  Store; update of an object
  newObj:      Store,  $\alpha \rightarrow$  Obj  $\alpha$ ;       creation of an object
  newObjStore: Store,  $\alpha \rightarrow$  Store;       storage allocation for a new object
  valid:       Store, Obj  $\alpha \rightarrow$  Bool;    test, if object id is declared for a store
  deref:       Store, Obj  $\alpha \rightarrow \alpha$ ;   dereferencing
```

Axioms:

Store, Obj **generated_by** emptyStore, newObj, newObjStore;

For the empty store:

valid(emptyStore, r) = false;

For selective update:

valid(s, v) $\Rightarrow$ valid(update(s, v, a), r) = valid(s, r)
 $\wedge$ deref(update(s, v, a), r) = **if** r = v **then** a **else** deref(s, r) **fi**;

For object storage allocation:

valid(s, newObj(s, a)) = false;

valid(newObjStore(s, a), r) = (valid(s, r) $\vee$ r = newObj(s, a));

valid(newObjStore(s, a), r) $\Rightarrow$

deref(newObjStore(s, a), r) = **if** r = newObj(s, a) **then** a **else** deref(s, r) **fi**;

}

Here Obj α is again a polymorphic sort. This means that for every sort M we may form the sort Obj M of the object identifiers of sort M. The operation newObj creates a new (anonymous) object identifier and newObjStore allocates the necessary storage for the new object. The operation valid tests if an object identifier is declared for a store; update(s, v, a) updates the object identifier v in object store s by the value a. Note that update is well-defined only for valid object identifiers.

The following two specifications introduce the sort Method and the sort Response. The elements of sort Method are input to the object store while the elements of sort Response are the output of the object store for method calls. Again $\text{Obj } \alpha$ stands for the polymorphic sorts of objects (or more exactly of object identifiers). Again we omit to introduce the selector functions, that we would need to give a proper specification of these simple sorts.

SPEC METHODS =

{ **sorts** $\text{Obj } \alpha$; Method = create(α) | upd($\text{Obj } \alpha$, α) | der($\text{Obj } \alpha$); }

SPEC RESPONSES =

{ **sorts** $\text{Obj } \alpha$; Response = ob($\text{Obj } \alpha$) | res(α); }

Finally we are ready to define the algebraic state machine based on these specifications above. It is given by the following specification:

SPEC ObjASM =

{ **based_on** OBJECT_MODEL, METHODS, RESPONSES;

interface: in: Method **input channel;**
 out: Response **output channel;**

state: s: Store;

initial: s = emptyStore;

dynamic:

 {true} in: create(a)/ out: ob(newObj(s, a)) {s' = newObjStore(s, a)};

 {valid(s, b)} in: upd(b, a)/ out: - {s' = update(s, b, a)};

 {valid(s, b)} in: der(b) / out: res(deref(s, b)) {s' = s};

}

Again the specification is rather straightforward. It follows closely the scheme defined in [11].

The specification ObjASM describes the state transition function of the algebraic state machine for objects. The state transition diagram shown in Fig. 2 visualizes it.

In this specification the fact that an object is valid is a stable property. If an object gets valid it remains valid forever. There is no way to eliminate an object. This diagram in Fig. 2 is a schematic description of the *life cycle* of a single object. Of course, it is not difficult to specify a more refined object model where objects can be deleted meaning that object identifiers can be made invalid.

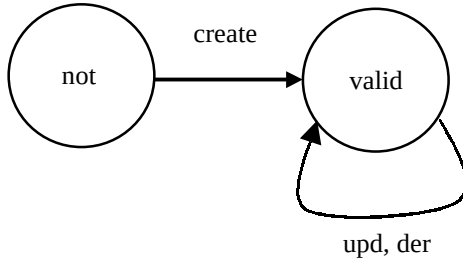


Fig. 2. Object Life Cycle

So far the example is again not using the full power of algebraic state machines since again the dynamic part consists only of a variable s of sort Store . However, making more explicit use of the concepts of algebraic state machines we may eliminate the sort Store completely and replace it by the following algebraic state machine ObjASM1 where we encode the store by using the functions valid and deref as state attributes. The state machine is based on the specification OBJECT_MODEL1 which introduces an infinite polymorphic sort $\text{Obj } \alpha$ of object identifiers.

```

SPEC OBJECT_MODEL1 =
{ based_on SET;
  sort Obj  $\alpha$ ;
   $\forall \text{set} \in \text{Set Obj } \alpha: \exists a \in \text{Obj } \alpha: \neg(a \in \text{set});$    Obj  $\alpha$  is infinite
}

```

The invariant (axiom) of this specification expresses that there is an infinite number of elements of sort $\text{Obj } \alpha$ for each sort α . Note that $\text{Set } \alpha$ is the sort of finite sets.

Then the algebraic state machine is specified as follows:

```

SPEC ObjASM1 =
{ based_on    OBJECT_MODEL1, METHODS, RESPONSES;
  interface:  in: Method    input channel;
                  out: Response output channel;
  state:      valid:  $\text{Obj } \alpha \rightarrow \text{Bool};$       test, if an object id is declared
                  deref:  $\text{Obj } \alpha \rightarrow \alpha;$       dereferencing
  initial:     $\forall r \in \text{Obj } \alpha. \text{valid}(r) = \text{false};$ 
}

```

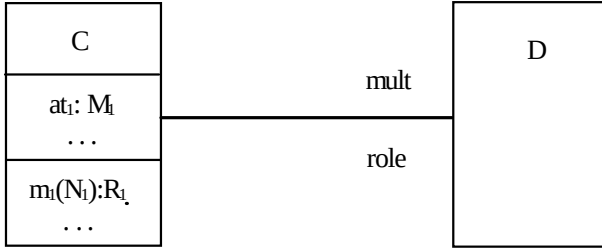



Fig. 3. Class diagram

First, we define the algebraic specification C_BASE1 (of case (ii)) by extending $OBJECT\_MODEL1$ as follows.

C_BASE1 is based on the specifications of all basic types such as INT . For each class C we introduce two sorts

$CRecord$ *representing the record sort of the attributes and the association roles of C*

$C = Obj\ CRecord$ *representing the object identifiers of C*

Elements of the sort $CRecord$ are constructed using the function symbol

$c: M_1, \dots, M_k, E \rightarrow CRecord$

where

$$E = \begin{cases} Set(D) & \text{if } mult = * \\ D & \text{if } mult = 1 \end{cases}$$

To define C_BASE (case (i)) we further extend C_BASE1 by the sort $Store$ and the selectors and updates for the attributes (similar to the specification $OBJECT\_MODEL$). The association is represented as an attribute of C :

$at_i: Store, C \rightarrow M_i, \quad updAt_i: Store, C, M_i \rightarrow Store, \quad i = 1, \dots, k$

$role: Store, C \rightarrow E, \quad updRole: Store, C, E \rightarrow Store,$

with the axioms

$deref(s, x) = c(m_1, \dots, m_i, \dots, m_k, e) \Rightarrow$

$at_i(s, x) = m_i \wedge updAt_i(s, x, n_i) = update(s, x, c(m_1, \dots, n_i, \dots, m_k, e))$

$\wedge role(s, x) = e \wedge updRole(s, x, e_i) = update(s, x, c(m_1, \dots, m_k, e_i))$

The methods specialize the specifications $METHODS$ and $RESPONSES$. Instead of polymorphic functions we specialize the names of $create$, ob and res by the appropriate class name and use " res_C " instead of " $ob\ C$ ".

SPEC C_METHODS =

{ **based_on** C_BASE1;

sort CMethod = createC($M_1, \dots, M_k, E$) | $m_1(N_1)$ | ... | $m_l(N_l)$ }

SPEC C_RESPONSES =

{ **based_on** C_BASE1;

sort CResponse = res_ T_1 (T_1) | ... | res_ T_m (T_m)

where $T_1, \dots, T_m$ are the elements of the set $\{C, R_1, \dots, R_r\} \setminus \{\text{Void}\};$ }

For example, if r is a term of sort R_1 , then $\text{res}_{R_1}(r)$ is a term of sort CResponse.

In the following we consider as example a simple class diagram for bank accounts (see Fig. 4) and specify it using the techniques above. Moreover and more importantly, we use our specification technique of algebraic state machines to add two kinds of constraints to the diagram: invariants and pre- and postconditions.

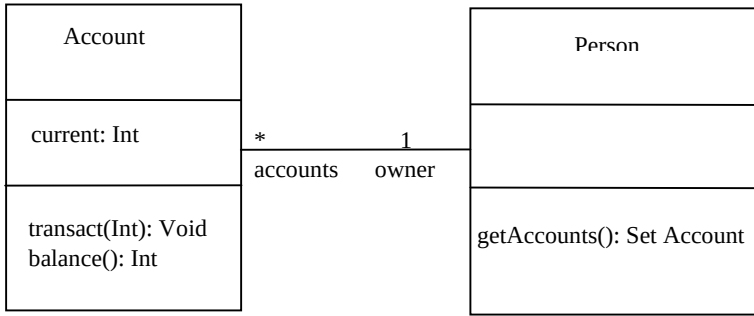


Fig. 4. Simple class diagram

The class diagram consists of two classes Account and Person which are related by an association. The class Account has an attribute current indicating the current balance of the account and two operations transact and balance for changing, and querying the balance. The class Person has the method getAccounts that computes the set of accounts of a person. The association relating both classes is of a type one-to-many: an account has exactly one owner whereas a person may posses several accounts. For simplicity, we omit further attributes and operations of both classes.

According to the translation schema (ii) above the class diagram of Fig. 4 induces automatically the following specification AP_BASE1 that introduces sorts and operations for the object identifiers and (records of) attributes and association values of class Account and Person.

SPEC AP_BASE1 =

{ **based_on** OBJECT_MODEL1, INT;

```

sort      PersonRecord, AccountRecord;
           Person   = Obj PersonRecord;
           Account  = Obj AccountRecord; }

```

The specification AP_BASE extends AP_BASE1 by constructor, update and select operations.

SPEC AP_BASE =

```
{ based_on   C_BASE1;
```

```
  sort      Store;
```

Constructors:

```
      person:      Set Account  $\rightarrow$  PersonRecord;
```

```
      account:     Int, Person  $\rightarrow$  AccountRecord;
```

Selectors and updates:

```
      accounts:    Store, Person  $\rightarrow$  Set Account;
```

```
      current:     Store, Account  $\rightarrow$  Int;
```

```
      owner:       Store, Account  $\rightarrow$  Person;
```

```
      updAccounts: Store, Person, Set Account  $\rightarrow$  Store;
```

```
      updCurrent:  Store, Account, Int  $\rightarrow$  Store;
```

```
      updOwner:    Store, Account, Person  $\rightarrow$  Store;
```

Axioms:

```
deref(s, p) = person(set)  $\Rightarrow$ 
```

```
  accounts(s, p) = set
```

```
   $\wedge$  updAccounts(s, p, set1) = update(s, p, person(set1));
```

```
deref(s, a) = account(c, p)  $\Rightarrow$ 
```

```
  current(s, a) = c  $\wedge$  owner(s, a) = p
```

```
   $\wedge$  updCurrent (s, a, c1) = update(s, a, account(c1, p))
```

```
   $\wedge$  updOwner (s, a, p1) = update(s, a, account(c, p1));
```

```
}
```

The instantiation of method and response specifications is as follows. For the create operations we assume standard initial values for most arguments. The operation `transact` introduces input messages of the form of a method call `acc.transact(m)` where `acc` denotes an object (identifier) of class `Account` and `m` an integer. Similarly the operations `balance( )` and `getAccounts( )` introduce incoming method calls of the form `acc.balance( )` and `p.getAccounts( )`. As responses we introduce messages which carry a value as an argument; e.g. `res_Int(3)` is a message carrying the value 3.

SPEC AP_METHODS =

```
{ based_on   AP_BASE1;
```

```

sort AP_Methods = createAccount(Person) | createPerson()
                  | Account.transact(Int) | Account.balance( ) | Person.getAccounts( ); }

```

SPEC AP_RESPONSES =

```

{ based_on    AP_BASE1;

```

```

  sort AP_Response = res_Account(Account) | res_Person(Person)
                    | res_Int(Int) | res_Set_Account(Set Account); }

```

Now we are ready to instantiate the algebraic state machine ObjASM. The one-to-many relationship induces an invariant relating the roles of owner and accounts⁶: if a person p is the owner of an account acc then acc is an element of the accounts attribute of p , and vice versa.

Moreover, we specify requirements which are not expressed in the class diagram: an invariant for the balance of any account, initial states, the behavior of the methods. As invariant we require that accounts can not have a negative balance. The behavior of all methods is specified using pre- and postconditions. Note that the method `transact` has a non-trivial precondition in order to ensure the preservation of the invariant.

SPEC AP_ASM = {

based_on AP_BASE, AP_METHODS, AP_RESPONSES;

interface: in: AP_Methods **input channel;**
 out: AP_Response **output channel;**

state: s: Store;

invariants: $\forall acc \in \text{Account}, p \in \text{Person}. \text{valid}(s, acc) \wedge \text{valid}(s, p) \Rightarrow$
 $\text{current}(s, acc) \geq 0 \wedge$ *balance requirement*
 $(acc \in \text{accounts}(s, p) \Leftrightarrow p = \text{owner}(s, acc));$ *one-to-many ass. invariant*

initial: s = emptyStore;

dynamic:

```

{accounts(s, p) = set  $\wedge$  account(0, p) = a}
  in: createAccount(p) / out: res_Account(newObj(s,a))
  {s' = updAccounts(newObjStore(s, a), p, set  $\cup$  {newObj(s,a)}};

{true}        in: createPerson( ) /out: res_Person (newObj(s, person( $\emptyset$ )))
               { s' = newObjStore(s, person( $\emptyset$ ))};

{current(s, acc) = x  $\wedge$  x+m  $\geq$  0} in: acc.transact(m) / out: -
               { s' = updCurrent(s, acc, c+m) };

{current(s,acc) = x }    in: acc.balance( ) / out: res_Int(x) {s' = s};

```

⁶ This invariant could be automatically generated from an extended translation schema. We have omitted this here for reasons of space.

```

{accounts(s,p) = set}    in: p.getAccounts( ) / out: res_Set_Account(set)
                                {s' = s};
}

```

The translation schema (ii) induces an algebraic state machine specification AP_ASM1 with the same interface as AP_ASM and AP_BASE1, AP_METHODS and AP_RESPONSES as base specifications. The function valid and the attributes accounts, current, owner describe the state space. The pre- and postconditions of the transition axioms make extensive use of the frame axioms. By assuming these axioms implicitly we could considerably simplify the specification.

SPEC AP_ASM1 = {

based_on AP_BASE1, AP_METHODS, AP_RESPONSES;

interface: in: AP_Methods **input channel;**
 out: AP_Response **output channel;**

state: valid : Obj $\alpha \rightarrow$ Bool; *test, if an object id is declared*
 accounts: Person $\rightarrow$ Set Account;
 current: Account $\rightarrow$ Int;
 owner: Account $\rightarrow$ Person;

invariants: $\forall acc \in \text{Account}, p \in \text{Person}. \text{valid}(acc) \wedge \text{valid}(p) \Rightarrow$
 $\text{current}(acc) \geq 0 \wedge$ *balance requirement*
 $(acc \in \text{accounts}(p) \Leftrightarrow p = \text{owner}(acc));$ *one-to-many association invariant*

initial: $\forall acc \in \text{Account}, p \in \text{Person}. \text{valid}(acc) = \text{false} \wedge \text{valid}(p) = \text{false};$

dynamic:

```

{valid(acc) = false  $\wedge$  accounts(p) = set}
  in: createAccount(p) / out: res_Account(acc)
    {valid'(acc) = true  $\wedge$  current'(acc) = 0  $\wedge$  owner'(acc) = p  $\wedge$ 
      accounts'(p) = set  $\cup$  {acc}  $\wedge$ 
      SAME(valid-acc, current-acc, owner-acc, accounts-p)};

{valid(p) = false}    in: createPerson( ) /out: res_Person(p)
  { accounts'(p) =  $\emptyset$   $\wedge$  SAME(valid, current, owner, accounts-p) };

{valid(acc)  $\wedge$  current(acc) = x  $\wedge$  x+m  $\geq$  0} in: acc.transact(m) / out: -
  {current'(acc) = x+m  $\wedge$  SAME(valid, current-acc, accounts, owner)};

{valid(acc)  $\wedge$  current(acc) = x} in: acc.balance( ) / out: res_Int(x)
  { SAME(valid, accounts, current, owner) };

{valid(p)  $\wedge$  accounts(p) = set} in: p.getAccounts( )/out: res_Set_Account(set)
  { SAME(valid, accounts, current, owner) };
}

```

As for the previous examples we can prove that AP_ASM and AP_ASM1 are equivalent w.r.t. the Forget-Identify implementation relation.

This example demonstrates that we can extend our method step by step into a full blown object oriented modeling technique in a very transparent and straightforward way. Everything we introduced is strictly covered by our basic model.

6 Composing Algebraic State Machines

Algebraic state machines are only a special case of state machines with input and output. Therefore we can associate a black box view with algebraic state machines as shown in [9] and compose them as defined in [10]. In the following we give a black box behavior semantics of state machines and show how to compose algebraic state machines.

6.1 Black Box View

We are interested in system models that allow us to represent systems in a modular way. We think of a system as being composed of a number of subsystems that we call *components*. Moreover, a system itself is a component again which can be part of a larger system. A component is a self-contained unit that encapsulates a hidden state with a clear cut interface. Via its interface it is connected with the environment. In this section we introduce a simple, very abstract mathematical notion of a system component.

For us, a (system) *component* is an information processing unit that communicates with its environment through a set of input and output channels. This communication takes place in a (discrete) time frame.

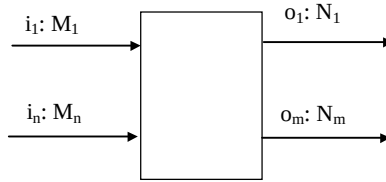


Fig. 5. Graphical Representation of a Component as a Data Flow Node with Input Channels $i_1, \dots, i_n$ and Output Channels $o_1, \dots, o_m$ and their respective sorts $M_1, \dots, M_n$ and $N_1, \dots, N_m$

In software engineering, it is helpful to distinguish between a *black box view* and a *glass box view* of a component. In a black box view we are only interested in the interface of a component with its environment. For this we have to describe the causal dependencies between the input and the output messages. In a glass box view we are interested in the internal structure of a component, which can either be given by its local state space together with a state transition relation or by its decomposition into subcomponents. We first give a model for the black box view.

Let Σ_{io} be the interaction interface signature (see Section 4) with a set of input channels and a set O of output channels. For simplicity, we use just one set M of data sorts for messages on the channels in the sequel in order to keep the presentation simple. A graphical representation of a component with its syntactic interface is shown in Fig. 3.

Let M be a sort of messages and signals. By M^* we denote the set of finite sequences over the set of messages M , by M^∞ we denote the set of infinite sequences over the set of messages M . By $\hat{}$ we denote the concatenation of sequences. The set M^∞ can be understood to be represented by the total mappings from the natural numbers N into M . Formally we define the set of *timed streams* as follows (we write S^∞ for the function space $N^+ \rightarrow S$ and N^+ for $N \setminus \{0\}$).

$$M^\chi =_{\text{def}} (M^*)^\infty$$

By $\langle \rangle$ we denote the empty stream, by $\langle m \rangle$ the one-element stream, by $\langle m_1 \dots m_k \rangle$ the stream of length k with the elements $m_1, \dots, m_k$. For every set of channels C , every mapping $v: C \rightarrow M^\chi$ provides a complete communication history. Note that $(C \rightarrow M^*)^\infty$ and $C \rightarrow (M^*)^\infty$ are isomorphic.⁷

We denote the set of the valuations of the channels in C by infinite timed streams

$$C \rightarrow M^\chi \text{ by } C^\rightarrow.$$

For every number $i \in N$ and every stream $x \in M^\chi$ we denote by $x \downarrow i$ the sequence of the first i sequences in the stream x . It represents the observation for the communication over the first i time intervals. By

$$\bar{x} \in M^* \cup M^\infty$$

we denote the finite or infinite stream that is the result of concatenating all the finite sequences in the stream x . This sequence $\bar{x}$ is finite if and only if only a finite number of sequences in x are nonempty. Going from the stream x to $\bar{x}$ provides a *time abstraction*. In the stream x we can find out in which time interval a certain message arrives, while in $\bar{x}$ we see only the messages in their order of communication without any indication of their timing.

We use both notations $x \downarrow$ and $\bar{x}$ as well as the concatenation introduced for streams x also for tuples and sets of timed streams by applying them pointwise.

As an example we consider the bank account machine AP_ASM of section 5.3. We fix a model A of the (sum of the) context specifications AP_BASE , $AP\_METHODS$ and $AP\_RESPONSES$. Then the set of input messages is given by the interpretation AP_Method^A of the sort AP_Method in A , the set of output messages is $AP\_Responses^A$. Note that A is also a model of the context specification of AP_ASM1 ; in particular, the sets of input and output messages are the same.

⁷ Moreover, the set M^χ is isomorphic to the set of streams over the set $M \cup \{\sqrt{}\}$ which contain an infinite number of time ticks (here $\sqrt{}$ denotes a time tick; we assume $\sqrt{} \notin M$).

$(AP\_Method^{A^*})^\infty$ is the set of timed infinite input message streams,

$I^\rightarrow = \{in\} \rightarrow (AP\_Method^{A^*})^\infty$ is the set of valuations of the input channel in by infinite timed method streams,

$I^{[*]} = \{in\} \rightarrow AP\_Method^{A^*}$ is the set of valuations of in by finite timed method streams.

An example for a finite input stream $v \in I^{[*]}$ is given by

$$v(in) = \langle a1.balance() \ a1.transact(-50) \ a1.balance() \rangle$$

where $a1 \in Account^A$. An infinite timed stream $x \in I^\rightarrow$ may start with $v(in)$ for the first time unit, contain no message during the second time unit, and then have another balance request and so on:

$$x(in) = \langle v(in) \rangle \langle a2.balance() \rangle \dots$$

Fig. 5 describes the syntactic interface of a component with the input channels $i_1, \dots, i_n$ of the sorts $M_1, \dots, M_n$ and the output channels $o_1, \dots, o_m$ of sorts $N_1, \dots, N_m$. In the theoretical treatment we assume for simplicity always the same sort M .

We represent the behavior of a component with the set of input channels I and the set of output channels O by a set-valued function:

$$F: I^\rightarrow \rightarrow \wp(O^\rightarrow)$$

This function yields the set of output histories $F.v$ for each input history v . Given the input history v a component with the behavior F produces one of the output histories in $F.v$. We write $F.v$ for $F(v)$ to save brackets.

Only if a set-valued function on streams fulfils certain properties we accept it as a representation of a behavior of a component. To give a precise definition of these requested properties we introduce a number of notions for set-valued functions on streams. A function

$$F: I^\rightarrow \rightarrow \wp(O^\rightarrow)$$

is called

- *timed*, if for all $i \in \mathbb{N}$ we have

$$v \downarrow i = z \downarrow i \Rightarrow F(v) \downarrow i = F(z) \downarrow i$$

Then the output in the time interval i only depends on the input received till the i 'th time interval. In the literature a function F with this property is called a *causal* function, too.

- *time guarded*, if for all $i \in \mathbb{N}$ we have

$$v \downarrow i = z \downarrow i \Rightarrow F(v) \downarrow i+1 = F(z) \downarrow i+1$$

Time guardedness assumes in addition to timedness that reaction to input is delayed by at least one time unit.

- *realisable*, if there exists a time guarded function⁸ $f: I^{\rightarrow} \rightarrow O^{\rightarrow}$ such that for all input histories v we have:

$$f.v \in F.v$$

By $[F]$ we denote the set of time guarded functions f with $f.v \in F.v$ for all x .

- *fully realisable*, if for all input histories v we have:

$$F.v = \{f.v: f \in [F]\}$$

We assume in the following that stream processing functions that represent the behaviors of components are always time guarded and fully realisable.

We define a state transition machine by a state transition function

$$\Delta: \text{State} \times I^{[*]} \rightarrow \wp(\text{State} \times O^{[*]})$$

and a set

$$\text{State}_0 \subseteq \text{State}$$

of initial states.

In the case of algebraic state machines State and State_0 are sets of algebras. Given a specification $(B, (I, O), P=(\Sigma, E), (E_{\text{init}}, E_D))$ (see section 4) we have

$$\text{State} = \text{Mod}(P), \text{State}_0 = \{A \in \text{Mod}(P) \mid A \models E_{\text{init}}\},$$

and Δ relates all those algebras of $\text{Mod}(P)$ which satisfy the axioms E_D .

For example, let $A1$ be a model of the state part of AP_ASM which has (at least) two accounts $a1, a2 \in \text{Account}^{A1}$ such that for instance $\text{current}(s, a1)^{A1} = 143$ and $\text{current}(s, a2)^{A1} = 20$. Then $\Delta_{\text{AP\_ASM}}(A1, v)$ (where v is the three element stream defined above in this section) consists of a state algebra $A2$ and a stream $w \in O^{[*]}$ such that $A2$ differs from $A1$ only in the value of $\text{current}(s, a1)$ and

$$\text{current}(s, a1)^{A2} = 93 \wedge w.o = \langle \text{res\_Int}(143) \text{ res\_Int}(93) \rangle$$

A state transition machine is nondeterministic, in general. In each transition step it takes a state and a communication pattern of its input streams and produces a successor state and a communication pattern for its output streams. For this kind of state transition machines, we represent the set of possibly updated states and outputs of a transition with the help of a predicate. Of course, sets of pairs of states and output patterns can be used here directly. A state machine models the behavior of an information processing unit with input channels from the set I and output channels from the set O in a time frame as follows. Given a family of finite sequences $v \in I^{[*]}$ representing the sequence of input messages $x(c)$ received in a time interval on the channel $c \in I$ of the component in state $\sigma \in \text{State}$, every pair (σ', y) in the set $\Delta(\sigma, x)$ represents a possible successor state and the sequence of output messages $y(c)$ produced on channel $c \in O$ in the next time interval.

⁸ Since functions can be seen as a special case of set valued functions where the result sets contain exactly one element, the notion of time guardedness extends to functions.

We associate a stream processing function with a state machine that is given by the transition function Δ using the following definition. More precisely, we associate a time guarded function F_{σ} with every state $\sigma \in \text{State}$ as defined by the following equation:

$$\begin{aligned}
 F_{\sigma}(x) = \{y \in O^{\rightarrow} : \\
 & (\exists i \in I^{[*]}, o \in O^{[*]}, \sigma' \in \text{State}, x' \in I^{\rightarrow}, y' \in O^{\rightarrow} : \\
 & \quad \bar{y} = o \hat{\ } \bar{y}' \wedge \bar{x} = i \hat{\ } \bar{x}' \wedge (\sigma', o) \in \Delta(\sigma, i) \wedge y' \in F_{\sigma'}(x')) \vee \\
 & (\forall i \in I^{[*]} : i \leq \bar{x} \Rightarrow \forall o \in O^{[*]}, \sigma' \in \text{State} : \neg(\sigma', o) \in \Delta(\sigma, i)) \}
 \end{aligned}$$

Here $\leq$ denotes the prefix relation on streams. If the state transition relation contains cycles then the definition of F_{σ} is recursive. In this case, we cannot be sure that by the equation above the behavior F_{σ} is uniquely specified. Therefore we define F_{σ} by the largest (in the sense of pointwise set inclusion) time guarded function that fulfils this equation.

The first (existentially quantified) part of the left-hand side of this defining equation handles the case where at least one of the input patterns applies. The second part treats the case where for the input stream x none of the input patterns applies. This case is mapped onto arbitrary output called *chaos*⁹. This definition is justified by the principle that, if nothing is specified for an input pattern, the system may react by arbitrary output. This definition, moreover, guarantees that the function F_{σ} is always fully realizable.

If we do not want to associate a chaotic, but a more specific behavior with input situations where no input pattern applies, we can work with default transitions (for instance time ticks) or simply drop the second clause in the definition. Working with chaos, however, has the advantage that adding input patterns for input for which no pattern applied so far is a correct refinement step in the development process.

When dealing with states that are algebras A we rather write $[\Delta]_A$ instead of F_A . As an example we consider the stream processing function $[\text{AP\_ASM}]$ associated with AP_ASM . Let $A1, x, v, w$, be as above. The function

$$[\text{AP\_ASM}]_{A1} : \{in\}^{\rightarrow} \rightarrow \wp(\{out\}^{\rightarrow})$$

abstracts from the states. For the timed stream x we get a result $y \in \{out\}^{\rightarrow}$ of the form

$$\langle \langle \text{res\_Int}(143) \text{ res\_Int}(93) \rangle \rangle \langle \text{res\_Int}(20) \rangle \dots \rangle$$

where the results are delayed by one time unit. Applying $[\text{AP\_ASM}]_{A1}$ to a stream $x1$ of the form

$$\langle \langle a2.balance() \rangle \langle a2.transact(-30) \rangle \dots \rangle$$

leads to chaos beginning with the third time unit; the result streams z have the form

⁹ If no input pattern applies we assume that a specific behaviour is not required.

$$z(\text{out}) = \langle \rangle \dots \langle \text{res\_Int}(20) \rangle \wedge y(\text{out}) \rangle$$

where $y(\text{out})$ is an arbitrary element of $(\text{AP\_Responses}^{\text{A}^*})^\infty$.

Because of the one-to-one correspondence of the states of (the models of) AP_ASM and AP_ASM1 , it is easy to see that the stream semantics of both specifications are the same.

Our model of the behavior of a component works with timed input and output streams. Since the input and output patterns of algebraic state machines do not refer to the timing of the messages, in the definition we work with time abstractions of the input and output streams.

Note that we can give along these lines a precise treatment for sophisticated concepts like priorities and spontaneous transitions due to our carefully chosen semantic model that includes time. Without an explicit notion (at least on the semantic level) of time a proper semantic treatment of priorities or of spontaneous reactions is difficult or even impossible.

6.2 Composition

When modeling systems and system components the composition of larger systems from smaller ones is a basic structuring technique. We consider only one basic composition operator for asynchronous interaction, namely parallel composition with feedback. It comprises *sequential* composition, *parallel* composition, and *feedback* as special cases. As well known these three composition operators suffice to formulate all kinds of networks of reactive information processing components, provided we have simple components available for permuting and copying input and output lines.

We work with channels and this makes it very simple to form networks. We only have to choose the names of the channels such that each channel occurs at most for one component as an input channel and at most for one component as an output channel. These all may be composed by the parallel composition with feedback as specified formally below.

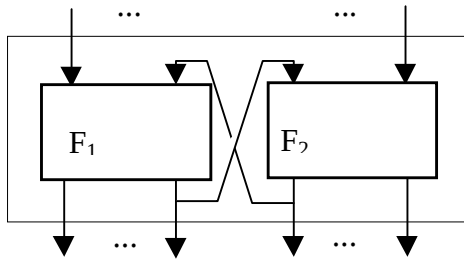


Fig. 6. Parallel Composition with Feedback

To define the parallel composition with feedback let ASM1 and ASM2 be two algebraic state machines of the form

$$(B_j, (I_j, O_j), P_j = (\Sigma_j, E_j), D_j) \quad j = 1, 2$$

with base specifications B_j , input channels I_j , output channels O_j , state specifications P_j and dynamic parts D_j .

We assume for the channels

$$O_1 \cap O_2 = \emptyset, O_1 \cap I_1 = \emptyset, O_2 \cap I_2 = \emptyset,$$

and for the signatures of P_j that the non-basic function symbols of P_1 are disjoint from all function symbols of P_2 and vice versa.

Then the input and output channels of the composed specification are given by

$$I =_{\text{def}} (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O =_{\text{def}} (O_1 \setminus I_2) \cup (O_2 \setminus I_1).$$

The state space specification P is the sum of P_1 and P_2 :

$$P =_{\text{def}} (\Sigma_1 \cup \Sigma_2, E_1 \cup E_2)$$

Then the composition $\text{ASM1} \otimes \text{ASM2}$ is semantically defined as stream processing function

$$[\text{ASM1} \otimes \text{ASM2}] : \text{Mod}(P) \rightarrow [I^\rightarrow \rightarrow \wp(O^\rightarrow)]$$

where $\text{Mod}(P)$ denotes the set of models of P and the data sets of $I^\rightarrow$ and $O^\rightarrow$ depend on the chosen common subalgebra of the models of P .

For any $\Sigma_1 \cup \Sigma_2$ -algebra $A \in \text{Mod}(P)$ and any timed infinite stream $x \in I^\rightarrow$ we define (with the valuation $y \in C$ where $C = I_1 \cup I_2 \cup O_1 \cup O_2$):

$$[\text{ASM1} \otimes \text{ASM2}]_A(x) =$$

$$\{y \mid O: x = y \mid I \wedge y \mid O_1 = [\text{ASM1}]_{A \upharpoonright \Sigma_1} (y \mid I_1) \wedge y \mid O_2 = [\text{ASM2}]_{A \upharpoonright \Sigma_2} (y \mid I_2) \}$$

Here by $y \mid O$ we denote the restriction of the valuation mapping $y \in C$ to the channel set $O \subseteq C$.

To give an example we modify the algebraic state machine AP_ASM for accounts to give a bonus for certain transactions. The machine AP_ASM2 renames the channels in and out of AP_ASM into in_i and out_i and introduces an additional input channel out and an additional output channel in (see Fig. 7). If a message $\text{acc.transact}(m)$ arrives at input channel in_i then depending on a boolean value arriving at channel out a bonus is added to the account acc .

SPEC $\text{AP\_ASM2} =$

based_on $\text{AP\_ASM}[\text{in} \rightarrow \text{in}_i, \text{out} \rightarrow \text{out}_i];$

interface: $\text{out}: \text{Bool}$ **input channel;**
 $\text{in}: \text{In Int}$ **output channel;**

dynamic:

$\{m > 0 \wedge \text{current}(s, \text{acc}) = c \}$
 $\text{in}_i: \text{acc.transact}(m), \text{out}: \text{true} / \text{out}_i: \text{res\_Int}(m \text{ div } 10), \text{in}: \text{islem}(c+m)$
 $\{s' = \text{updCurrent}(s, \text{acc}, c+m+m \text{ div } 10) \};$
 $\}$

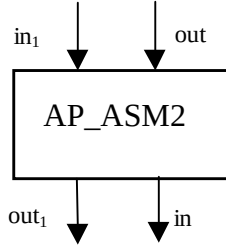


Fig. 7. Refined Account ASM with Bonus

To get a bonus component we compose AP_ASM2 with the machine for finite sets $SetASM1$ (see Fig. 8).

The bonus component is given by

$$AP\_ASM2 \otimes SetASM$$

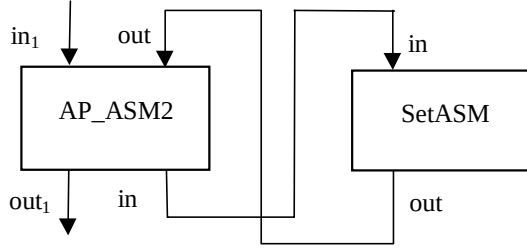


Fig. 8. Parallel Composition with Feedback

Its interaction interface is $(\{in_1\}, \{out_1\})$. The context is given by the specifications SET , $INPUT$, AP_BASE , $AP\_METHODS$, $AP\_RESPONSES$. The state space consists of two constants $s1$: $Set\ Int$, $s2$: $Store$ and the corresponding invariants.

For any model A of the state space specification and any timed input stream $x \in \{in_1\}^\rightarrow$ we get $(y \in C \text{ where } C = \{in_1, out_1, in, out\})$

$$\begin{aligned} [AP\_ASM2 \otimes SetASM]_A(x) &= \{y \mid \{out_1\}: \\ &\quad x = y \mid \{in_1\} \wedge \\ &\quad y \mid \{out_1, in\} = [AP\_ASM2]_A(y \mid \{in_1, out\}) \wedge \\ &\quad y \mid \{out\} = [SetASM]_A(y \mid \{in\}) \} \end{aligned}$$

For example, if the timed input stream begins with

$$\langle a1.transact(m)^{A1} \rangle$$

in a state $A1$ with $\text{current}(s2, a1)^{A1} = 134$ and $134+m \in s1^{A1}$, then the next element of the output stream is $\langle \text{res\_Int}(134+134 \text{ div } 10) \rangle$; this means that the account $a1$ got the bonus 13. If on the contrary, $134+m \notin s1^{A1}$ then no bonus is given.

Note that the composition of AP_ASM2 with SetASM1 has the same set of state transition functions as $\text{AP\_ASM2} \otimes \text{SetASM}$. More generally, one can show that the parallel composition operator preserves implementation equivalence: parallel composition is compositional!

7 Related Work

The design of algebraic state machines was influenced by several specification and design approaches: abstract state machines [18, 19], automata and (classical) state machines [7], component specifications based on stream processing functions [11]. The object-oriented instance of algebraic state machines was developed based on our experience with UML [26].

The idea of algebras as states goes back at least to Gaudel [16] and Ganzinger [15]. Gaudel proposes in her thesis two kinds of operations on algebras, access functions and modifiers. This approach is continued, applied and formalized e.g. in [13], [17]; a similar approach is proposed by [Astesiano, Zucca 95] in terms of a new mathematical structure, called “d-oid”. [15] considers modules as algebras and introduces operators for updating and iterating algebras, - a concept which was earlier used in category theory (cf. e.g. [14]).

The abstract state machines of Gurevich [18] use similar concepts: the state of an abstract state machines is an algebra and there is a fixed assignment operation which allows one to update carrier sets and function symbols. The main difference to algebraic state machines is the concept of communication. Abstract state machines are based on synchronous communication; in each computation step all enabled rules must also fire synchronously. Algebraic state machines are based on asynchronous communication; messages are exchanged asynchronously over communication channels. Moreover, updating an abstract state machine is similar to assignment in imperative languages and not connected with interface signatures as in algebraic state machines. Another difference is that Gurevich uses a model-oriented and not an axiomatic approach. Abstract state machines are not meant to be formalized in a particular logic but should be used in a rigorous (not completely formalized) way.

The use of pre- and postconditions and invariants goes back to Hoare [22] and is well established in programming and design. Good examples are Eiffel [25] and OCL [32] and its use in object-oriented development with UML [30].

Algebraic state machines are similar to I/O-automata [24] in the sense that they also support different input and output channels. However, they generalize I/O-automata by admitting highly structured states with (dynamic) operations as attributes. The same holds for other classical automata types (see. e.g. [7]). Also the notion of states in statecharts [20] does not support operations as attributes. Simple transitions of

statecharts correspond to transitions of algebraic state machines with one input and one output channel as follows.

The guard of a statecharts transition is expressed by the precondition, the input signal corresponds to the message on the input channel, the actions are expressed by the post condition using the function symbols of the context specification. Concurrency in algebraic state machines is expressed by parallel composition of machines and not by concurrent substates as in statecharts.

Another major difference is that automata and statecharts are not suited to specify data whereas algebraic state machines support data type and behavior specification in an integrated way.

An algebraic state machine can be seen as a component specification with an interface given by the interface signature together with the appropriate input and output message specification and an internal behavior specification given by the state machine transitions. In this sense algebraic state machines are similar to component specifications in ROOM [28] or UML-RT [29] which allow one to define interfaces and behavior using statecharts. Algebraic state machines could be used as a formal extension (and as a semantic foundation) of ROOM diagrams.

8 Conclusion

We have demonstrated a straightforward way to base the idea of abstract state machines on algebraic specifications. Of course, the notation is not the most concise, convenient one and could and should be considerably be streamlined. We may improve its readability for instance, by the introduction of diagrams and tables. Furthermore, we have shown how to combine it to two other fundamental ideas namely class diagrams and stream processing functions. Without much technical overhead this leads to a quite interesting compositional specification method for interactive distributed systems.

One critical issue remains. Is it better for the readability of specifications and for their formal analysis to separate the description of the state space and its invariants as much as possible from the dynamic part? We think yes. We believe it is much better to give a classical algebraic specification first that includes the major invariants of the system and then using constants and function identifiers (attributes) of the sorts introduced in this specification to describe the state space.

References

1. M. Abadi, L. Lamport: Composing specifications. Digital Systems Research Center, SRC Report 66, October 1990.
2. D. Ancona, M. Cerioli, E. Zucca: A formal framework with late binding. In J.P. Finance (ed.): *Fundamental Approaches to Software Engineering, FASE '99, Lecture Notes in Computer Science 1577*, pages 30-44, Berlin: Springer, 1999.

3. E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner (eds.): *Algebraic Foundations of Systems Specifications*. Berlin: Springer, 1999.
4. H. Baumeister: Relations as abstract datatypes: An institution to specify relations between algebras. In TAPSOFT '95, *Lecture Notes in Computer Science 915*, pages 756-771, Aarhus, Denmark, May 1995. Springer.
5. H. Baumeister: *Relations between Abstract Datatypes modeled as Abstract Datatypes*, PhD thesis, Universität Saarbrücken, 1999.
6. E. Börger: Why use evolving algebras for hardware and software Engineering. In: M. Bartosek, J. Standek, J. Wiedermann (eds): SOFSEM '95, 22nd Seminar on Current Trends in Theory and Practice of Informatics. *Lecture Notes of Computer Science 1012*, pages, 235-271. Springer, Berlin, 1995.
7. W. Brauer: *Automatentheorie*, Teubner 1984.
8. M. Broy: Views of queues. *Science of Computer Programming 11*, pages 65-86, 1988.
9. M. Broy: Mathematics of software engineering. Invited talk at MPC 95. In: B. Möller (ed.): Mathematics of Program Construction, July 1995, Kloster Irsee, *Lecture Notes of Computer Science 947*, pages, 18-47. Springer, Berlin, 1995.
10. M. Broy: Mathematical system models as a basis of software engineering. J. van Leeuwen (ed.): Computer Science Today. *Lecture Notes of Computer Science 1000*, pages, 292-306. Springer, Berlin, 1995.
11. M. Broy: The specification of system components by state transition diagrams. Technische Universität München, Institut für Informatik, TUM-I9729, Mai 1997.
12. M. Chandy, J. Misra: *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
13. P. Dauchy: *Développement et exploitation d'une spécification algébrique du logiciel embarqué d'un métro*. Thèse, Université de Paris-Sud, Orsay 1992.
14. C.C. Elgot: Monadic computation and iterative algebraic theories. *Proc. Logic Colloquium 73*, pages 175-230, Amsterdam: North-Holland, 1975.
15. H. Ganzinger: Denotational semantics for languages with modules. In: D. Björner (ed.): *TC2 Working Conference of Formal Description of Programming Concepts II*, pages, 3-21. Garmisch, 1982.
16. M.-C. Gaudel: *Correctness Proof of Programming Language Translation*, pages 25-43. 1982.
17. M.-C. Gaudel, C. Khoury, A. Zamulin: Dynamic systems with implicit state. In J.P. Finance (ed.): Fundamental Approaches to Software Engineering, FASE '99, *Lecture Notes in Computer Science*, pages, 114-128. Berlin: Springer, 1999.
18. Y. Gurevich: Evolving Algebra 1993: Lipari Guide. In: E. Börger (ed.): *Specification and Validation Methods*. Oxford University Press 1995.
19. Y. Gurevich: Abstract state machines. In T. Rus (ed.): *AMAST 2000*, Iowa City, 2000, this volume.
20. D. Harel: A visual formalism for complex systems. *Science of Computer Programming 8*, pages 231-274, 1987.
21. R. Hettler: Entity/Relationship-Datenmodellierung in axiomatischen Spezifikations-sprachen. Dissertation, TU München. Reihe Softwaretechnik, Marburg: Tectum Verlag, 1995.
22. C. A. R. Hoare: An axiomatic basis for computer programming. *Comm. ACM 12*, pages 576-583, 1969.
23. L. Lamport: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16(3)*, pages 872-923, 1994.
24. N. Lynch, E. Stark: A proof of the Kahn principle for input/output automata. *Information and Computation 82*, pages 81-92. 1989.
25. B. Meyer: *Object-Oriented Software Construction*. Prentice Hall International, 1988.

26. Rational: *The Unified Modeling Language, Version 1.3*, Rational Software Corporation, <http://www.rational.com>, 1999.
27. M. Broy, C. Facchi, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, K. Stølen: *The Requirement and Design Specification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I/II* Technische Universität München, Institut für Informatik, TUM-I9311 / TUM-I9312, May 1993.
28. B. Selic, G. Gullekson, P.T. Ward: *Real Time Object Oriented Modeling*. Wiley & Sons, 1994.
29. B. Selic, J. Rumbaugh: *Using UML for Modeling Complex Real-Time Systems*. Rational Software Corporation & ObjecTime Ltd., 1998.
30. D. D'Souza, A.C. Wills: *Objects, Components, Frameworks with UML: The Catalysis approach*. Addison-Wesley, 1998.
31. M. Wirsing: Algebraic Specification. *Handbook of Theoretical Computer Science*, Vol. B, pages 675-788, Amsterdam: North Holland, 1990.
32. J. Warmer, A. Kleppe: *The Object Constraint Language: Precise Modeling with UML*. Reading, Mass.: Addison-Wesley, 1999.
33. E. Astesiano, E. Zucca: D-oids: a model for dynamic data types. *Mathematical Structures in Computer Science* 5(2), pages 257-282, 1995.
34. CoFI Language Design Group: CASL - Summary (Version 1.0). <http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary>, 1998.
35. K. Lano: *Formal Object-Oriented Development*. London: Springer, 1995.

Meta Languages in Algebraic Compilers^{*}

Eric Van Wyk

Oxford University Computing Laboratory

Abstract. Algebraic compilers provide a powerful and convenient mechanism for specifying language translators. With each source language operation one associates a computation for constructing its target language image; these associated computations, called *derived operations*, are expressed in terms of operations from the target language. Sometimes the target language operations are not powerful enough to specify the required computations and one may then need to extend the target language algebras with more computationally expressive operations. A better solution is to package them in a *meta language* which can be automatically composed with the target language operations to ensure that all operations needed or desired for performing a translation are provided. In this paper, we show how imperative and functional meta languages can be composed with a target language in an example which implements a temporal logic model checker as an algebraic compiler and show how meta languages can be seen as components to be combined with a source and target language to generate an algebraic compiler.

1 Introduction

Attribute grammars [7,1] and algebraic compilers [9] provide powerful and convenient mechanisms for specifying language translators. In both, one associates with each operation in the source language computations for constructing the target language images of source constructs created by the operation. The complexity of these computations contributes to the complexity of the entire language translator specification, and we are thus interested in means of reducing the specification's complexity by writing these computations in languages appropriate to the translation task at hand. These languages must be computationally expressive enough to perform the necessary computations, and should provide convenient programming constructs which simplify the specification process for the translator implementor. Since algebraic compilers provide solid mathematical framework which provide a clear distinction between the target language and the language used to specify the translation, they provide a better context in which to explore the issues of *meta languages*.

An algebraic compiler $\mathcal{C} : L_S \rightarrow L_T$ is a *language-to-language* translator that uses an algorithm for homomorphism computation to embed a source language L_S into a target language L_T . The computations associated with each source language operation that define an algebraic compiler are written in terms using the

^{*} This work is funded by Microsoft Research.

operations from the target language and are called *derived operations*. In some cases, the operations provided by the target language are not expressive enough to correctly specify the translation or exist at such a low level of abstraction, with respect to the source language, that the specification is excessively difficult to read and write. In such cases, the target language is extended with additional operations to make the translation possible or more easily specifiable. In this paper, we explore how different *meta languages* can be used in conjunction with operations of the target language, to correctly and conveniently specify translators implemented as algebraic compilers without extending the target language.

As an example, we develop a model checker for the temporal logic CTL (computation tree logic) [3] as an algebraic compiler which maps the source language CTL into a target language of satisfiability sets. Since the operations in the target language of sets are not powerful enough to specify general computations, we must use a meta language to provide a more computationally expressive language in which to specify this translation. We show how both functional and imperative style meta languages can be used in the specification, thus giving the language implementor some choice in choosing an appropriate meta language.

Section 2 describes CTL and model checking. In Section 3 we define algebraic languages and compilers and show how CTL and models can be specified as algebraic languages. Section 4 discusses meta languages in algebraic compilers and specifically the meta languages used to implement a model checker as an algebraic compiler. Section 5 provides the specification of the model checker as an algebraic compiler using both a functional and an imperative meta language. Section 6 contains some comments on meta languages in attribute grammars, domain specific meta languages, and future work.

2 Model Checking

We present the problem of model checking a temporal logic as a language translation problem and implement two solutions as generalized homomorphisms using different meta languages. Model checking [3] is a formal technique used to verify the correctness of a system according to a given correctness specification. Systems are represented as labeled finite state transition systems called *Kripke models* [8] or simply *models*. Correctness properties are defined by formulas written in a temporal logic. In this paper, we use CTL [3], a propositional, branching-time temporal logic as our example. A model checking algorithm determines which states in a model satisfy a given temporal logic formula, and can thus be seen as a language translator which maps formulas in the temporal logic language to sets in a set language defined by the model.

A model M is a tuple $M = \langle S, E, P: AP \rightarrow 2^S \rangle$, where S is a finite set of states, $S = \{s_1, s_2, \dots, s_m\}$, and E defines directed edges between states and is a binary relation on S , $E \subseteq S \times S$, such that $\forall s \in S, \exists t \in S, (s, t) \in E$, that is, every state has a successor. For each $s \in S$ we use the notation $\text{succ}(s) = \{s' \in S \mid (s, s') \in E\}$. A *path* is an infinite sequence of states $(s_0, s_1, s_2, \dots)$ such that $\forall i, i \geq 0, (s_i, s_{i+1}) \in E$. AP is a finite set of *atomic propositions*,

$AP = \{p_1, p_2, \dots, p_n\}$, P is a proposition labeling function that maps an *atomic proposition* in AP to the set of states in S on which that proposition is *true*. Figure 1 shows a model [3] for two processes competing for entrance into a critical

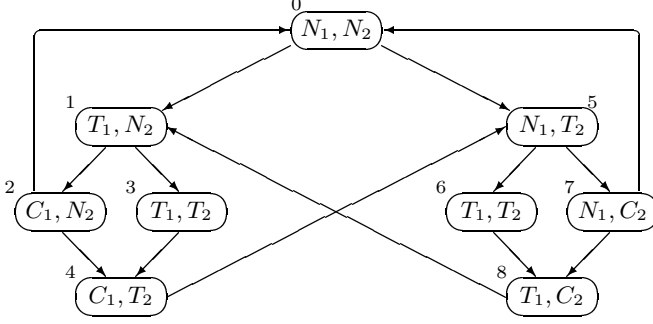


Fig. 1. Model Example

section. The atomic propositions T_i , N_i , and C_i denote process i , $1 \leq i \leq 2$, trying to enter the critical section, not trying to enter the critical section, and executing in the critical section, respectively.

The set of well-formed CTL formulas is described by the rules [3]:

1. The logical constants, *true* and *false* are CTL formulas.
2. Every atomic proposition, $p \in AP$, is a CTL formula.
3. If f_1 and f_2 are CTL formulas, then so are $\neg f_1$, $f_1 \wedge f_2$, $AX f_1$, $EX f_1$, $A[f_1 U f_2]$, and $E[f_1 U f_2]$.

As in [3], we define the satisfaction relation $\models$ of a CTL formula f on a state s in M , denoted $s \models f$ or $M, s \models f$ and read “ s satisfies f ”, as follows:

$$\begin{aligned}
 s &\models p \text{ iff } s \in P(p) \\
 s &\models \neg f \text{ iff not } s \models f \\
 s &\models f_1 \wedge f_2 \text{ iff } s \models f_1 \text{ and } s \models f_2 \\
 s &\models AX f_1 \text{ iff } \forall (s, t) \in E, t \models f_1 \\
 s &\models EX f_1 \text{ iff } \exists (s, t) \in E, t \models f_1 \\
 s &\models A[f_1 U f_2] \text{ iff } \forall \text{ paths } (s_0, s_1, s_2, \dots), s = s_0 \text{ and} \\
 &\quad \exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow s_j \models f_1]] \\
 s &\models E[f_1 U f_2] \text{ iff } \exists \text{ a path } (s_0, s_1, s_2, \dots), s = s_0 \text{ and} \\
 &\quad \exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow s_j \models f_1]]
 \end{aligned}$$

The set of states $\{s \in S \mid M, s \models f\}$ is called the *satisfiability set* of the formula f for model M . For the model in Figure 1, we can express the mutual exclusion property that both processes should not be in the critical section at the same time by the CTL formula $\neg(C_1 \wedge C_2)$. The absence of starvation property, which

states that if a process is trying to enter the critical section it will eventually be able to do so, is described for process i by the formula $\neg T_i \vee A[\text{true} \cup C_i]$. The model checker verifies that these properties hold on all states in the model.

We present both a functional and imperative version of a CTL model checker implemented as an algebraic compiler [9] $\mathcal{MC} : L_S \rightarrow L_T$ where the source language L_S is CTL and the target language L_T is a language describing the subsets of the states of the model M . The algebraic compiler $\mathcal{MC}$ translates a CTL formula f , to the set of states, S' , on which the formula f holds. That is, $\mathcal{MC}(f) = S'$ where $S' = \{s \in S \mid M, s \models f\}$.

3 Algebraic Compilers

3.1 Σ -Algebras and Σ -Languages

An *operator scheme* is a tuple $\Sigma = \langle S, Op, \sigma \rangle$ where S is a set of sorts, Op is a set of operator names, and σ is a mapping defining the signatures of the operator names in Op over the sorts in S . That is, $\sigma: Op \rightarrow S^* \times S$ such that if, for example, s_0, s_1 , and s_2 are sorts S and op is an operator name in Op which stands for operations which take an element of sort s_1 and an element of sort s_2 and generates an element of sort s_0 , then $\sigma(op) = s_1 \times s_2 \rightarrow s_0$.

A Σ -algebra is a family of non-empty sets, called the *carrier sets*, indexed by the sorts S of Σ and a set of Op named operations over the elements of these sets whose signatures are given by σ . There may be many different algebras for the same operator scheme Σ . These algebras are called *similar* and are members of the same *class of similarity*, denoted $\mathcal{C}(\Sigma)$. An interesting member of $\mathcal{C}(\Sigma)$ is the *word* or *term* algebra for Σ . This algebra is parameterized by a set of variables $V = \{V_s\}_{s \in S}$ and is denoted $W_\Sigma(V)$. Its carrier sets contain words formed from the variables of V and operator names of Op and its operators construct such words according to the operations signatures defined by σ [4]. Variables in V are called *generators* and V is thus said to *generate* $W_\Sigma(V)$.

A Σ -language [9] L is defined as the tuple $\langle \mathcal{A}^{sem}, \mathcal{A}^{syn}, \mathcal{L}: \mathcal{A}^{sem} \rightarrow \mathcal{A}^{syn} \rangle$ where $\mathcal{A}^{sem}$ is a Σ -algebra which is the language semantics, $\mathcal{A}^{syn}$ is a Σ word algebra which is the language syntax, and $\mathcal{L}$ is a partial mapping called the *language learning function* [9,10]. $\mathcal{L}$ maps semantic constructs in $\mathcal{A}^{sem}$ to their expressions as syntactic constructs in $\mathcal{A}^{syn}$ such that there exists a complementary homomorphism $\mathcal{E}: \mathcal{A}^{syn} \rightarrow \mathcal{A}^{sem}$ where if $\mathcal{L}(\alpha)$ is defined, then $\mathcal{E}(\mathcal{L}(\alpha)) = \alpha$, $\alpha \in \mathcal{A}^{sem}$. $\mathcal{E}$ is called the *language evaluation function* and maps expressions in $\mathcal{A}^{syn}$ to their semantic constructs in $\mathcal{A}^{sem}$.

CTL as a Σ -Language CTL can be specified as the Σ -language $L_{ctl} = \langle \mathcal{A}_{ctl}^{sem}, \mathcal{A}_{ctl}^{syn}, \mathcal{L}_{ctl} \rangle$ [12] using the operator scheme $\Sigma_{ctl} = \langle S_{ctl}, Op_{ctl}, \sigma_{ctl} \rangle$ where $S_{ctl} = \{F\}$, the set of sorts containing only one sort for “formula”, $Op_{ctl} = \{\text{true}, \text{false}, \text{not}, \text{and}, \text{or}, \text{ax}, \text{ex}, \text{au}, \text{eu}\}$, and σ_{ctl} is defined below:

$$\begin{array}{lll}
\sigma_{ctl}(true) = \emptyset \rightarrow F & \sigma_{ctl}(not) = & F \rightarrow F & \sigma_{ctl}(ax) = & F \rightarrow F \\
\sigma_{ctl}(false) = \emptyset \rightarrow F & \sigma_{ctl}(and) = & F \times F \rightarrow F & \sigma_{ctl}(ex) = & F \rightarrow F \\
& \sigma_{ctl}(or) = & F \times F \rightarrow F & \sigma_{ctl}(au) = & F \times F \rightarrow F \\
& & & \sigma_{ctl}(eu) = & F \times F \rightarrow F
\end{array}$$

As CTL formulas are written using atomic propositions from a specific model M , the syntax algebra $\mathcal{A}_{ctl}^{syn}$ is parameterized by the set of atomic propositions AP from M and is denoted as $\mathcal{A}_{ctl}^{syn}(AP)$. For example, the formula $\neg(C_1 \wedge C_2)$ shown above has variables C_1 and C_2 from AP of the above model and the $\wedge$ and $\neg$ operations construct the CTL formula (in the syntax word algebra) from these variables. The syntax (word) algebra $\mathcal{A}_{ctl}^{syn}(AP)$ has as its carrier set all possible CTL formulas written using the atomic propositions in AP . The operations of this algebra construct formulas (words, if you like) from variables and operator names. The set of variables AP generates the algebra $\mathcal{A}_{ctl}^{syn}(AP)$.

Just as the syntactic algebra $\mathcal{A}_{ctl}^{syn}(AP)$ is parameterized by the atomic propositions AP of the model M , the semantic algebra $\mathcal{A}_{ctl}^{sem}$ is also parameterized by M in that the carrier set of the semantic algebra $\mathcal{A}_{ctl}^{sem}$ is the power set of the set of states of the model M . The operations in this algebra, while *similar* (that is, having the same signature) to those in $\mathcal{A}_{ctl}^{syn}$, operate on sets, not formulas, since the meaning of a CTL formula is in fact its satisfiability set. Although the operations in the word algebra $\mathcal{A}_{ctl}^{syn}(AP)$ are easily defined as simply concatenating operation names and operands together, the operations in the semantic algebra $\mathcal{A}_{ctl}^{sem}$ are not so simply defined. We will thus name these operations in $\mathcal{A}_{ctl}^{sem}$ and define them individually. The operation names $\{true, false, not, and, or, ax, ex, au, eu\}$ in Op_{ctl} are instantiated in $\mathcal{A}_{ctl}^{sem}$ by the respective operations $\{\mathcal{S}, \emptyset, \mathcal{C}, \cap, \cup, Next_{all}, Next_{some}, lfp_{all}, lfp_{some}\}$ where

- $\mathcal{S}$ is the constant set of all states in M and $\emptyset$ is the constant empty set.
- $\mathcal{C}$ is the unary operator that produces the complement in $\mathcal{S}$ of its argument.
- $\cap$ and $\cup$ are the binary set union and intersection operators.
- For $\alpha \in \mathcal{S}_M$ the unary operators $Next_{all}(\alpha)$ and $Next_{some}(\alpha)$ are defined by the equalities $Next_{all}(\alpha) = \{s \in \mathcal{S} | successors(s) \subseteq \alpha\}$ and, $Next_{some}(\alpha) = \{s \in \mathcal{S} | successors(s) \cap \alpha \neq \emptyset\}$, respectively, where $successors(s)$ denotes the successors of the state s in the model M .
- lfp_{all} and lfp_{some} are inspired by the Y operator for fixed point construction [5]. For $\alpha, \beta \in 2^{\mathcal{S}}$, $lfp_{all}(\alpha, \beta)$ computes the least fixed point of the equation $Z = \beta \cup (\alpha \cap \{s \in \mathcal{S} | successors(s) \subseteq (\alpha \cap Z)\})$ and $lfp_{some}(\alpha, \beta)$ computes the least fixed point of the equation $Z = \beta \cup (\alpha \cap \{s \in \mathcal{S} | (successors(s) \cap \alpha \cap Z) \neq \emptyset\})$ [3].

Although the algebra $\mathcal{A}_{ctl}^{sem}$ exists, it is not used directly in the model checking process. It is only used to explain CTL as an Σ_{ctl} -language.

A Model as an Σ -Language As the target language of our algebraic model checker, we develop a Σ -language based on sets which is parameterized by a specific model. For a model M , $L_M = \langle \mathcal{A}_M^{sem}, \mathcal{A}_M^{syn}, \mathcal{L}_M \rangle$ using operator scheme $\Sigma_M = \langle \mathcal{S}_M, Op_M, \sigma_M \rangle$ where $\mathcal{S}_{ctl} = \{Set, Node, Boole\}$, $Op_{ctl} = \{\emptyset, \cup, \cap, \setminus, succ, \subseteq, =, \in, \{\ \} \}$, and σ_M is defined below:

$$\begin{aligned}
\sigma_M(\emptyset) &= \emptyset \rightarrow Set & \sigma_M(\subseteq) &= Set \times Set \rightarrow Boole \\
\sigma_M(S) &= \emptyset \rightarrow Set & \sigma_M(=) &= Set \times Set \rightarrow Boole \\
\sigma_M(\cup) &= Set \times Set \rightarrow Set & \sigma_M(\in) &= Set \times Node \rightarrow Boole
\end{aligned}$$

$$\begin{aligned}
\sigma_M(\cap) &= Set \times Set \rightarrow Set & \sigma_M(succ) &= Node \rightarrow Set \\
\sigma_M(\backslash) &= Set \times Set \rightarrow Set & \sigma_M(\{ \}) &= Node \rightarrow Set
\end{aligned}$$

The operators here are mostly self-descriptive. $\emptyset$ and S generate respectively the empty set and the full set of states S . The binary operators $\cup$, $\cap$, and $\backslash$ are respectively set union, intersection and difference. We also have the subset ($\subseteq$), set equality ($=$), and membership operations ($\in$) and the successor function *succ* and singleton set creation function denoted by $\{ \}$. These operators build set expressions in the syntax algebra $\mathcal{A}_M^{syn}$ and sets in the semantic algebra $\mathcal{A}_M^{sem}$.

3.2 Algebraic Compilers

An *algebraic compiler* [9,10] $\mathcal{C}: L_S \rightarrow L_T$ which maps the language $L_S = \langle \mathcal{A}_S^{syn}, \mathcal{A}_S^{sem}, \mathcal{L}_S \rangle$ into the language $L_T = \langle \mathcal{A}_T^{syn}, \mathcal{A}_T^{sem}, \mathcal{L}_T \rangle$ is a pair of (generalized) homomorphisms ($H_{syn}: \mathcal{A}_S^{syn} \rightarrow \mathcal{A}_T^{syn}$, $H_{sem}: \mathcal{A}_S^{sem} \rightarrow \mathcal{A}_T^{sem}$) defined such that the diagram in Figure 2 commutes. In general, the operator schemes of the

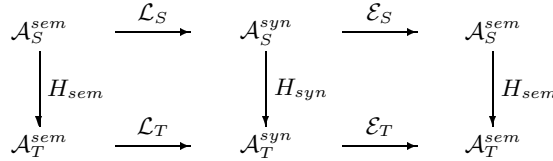


Fig. 2. An algebraic compiler.

algebras in these two languages may not be similar, as is the case with the operator schemes Σ_{ctl} and Σ_M for the languages L_{ctl} and L_M we intend to use in our model checker. Thus, a homomorphism, which associates a single target algebra operation with each source algebra operation is not possible. Instead, for each source algebra operation, we must build an appropriate operation from several target algebra operations. Such operations are called *derived operations*. Derived operations are written using words from the target word algebra using a set of *meta variables*. We will use subscripted versions of the sort names from the source language operator scheme as meta variables. The word “ $\mathcal{S} \setminus F_1$ ”, is a word in the algebra $\mathcal{A}_M^{syn}(\{F_1\})$ which represents the unary derived operation for taking the complement of a set with respect to the full set of states S . The meta variable F_1 is the “formal parameter” of the derived operation. We will associate this derived operation with the CTL operation *not* since given the satisfiability set of a formula f , it will generate the satisfiability set of the formula *not* f .

To define a generalized homomorphism [6] H from algebra $\mathcal{A}_{\Sigma_S}$ with operator scheme $\Sigma_S = \langle S_S, Op_S, \sigma_S \rangle$ to algebra $\mathcal{A}_{\Sigma_T}$ with the possibly dissimilar operator scheme $\Sigma_T = \langle S_T, Op_T, \sigma_T \rangle$ we must define two mappings:

1. a *sort map*, $sm: S_S \rightarrow S_T$ which maps source algebra sorts to target algebra sorts. In a generalized homomorphism, an object of sort s of Σ_S will be mapped to an object of sort $sm(s)$ of Σ_T .
2. a *operator map*, $om: Op_S \rightarrow W_{\Sigma_T}(S'_S)$, which maps operators in the source algebra to derived operations written as words in the target syntax algebra with meta variables S'_S – the source sorts with subscripts.

The derived operations, which take operands from the target algebra, have the same signatures as their counterparts in the source algebra, and thus we implicitly create an intermediate, hybrid algebra $\mathcal{A}_{\Sigma_S}^{ST}$ which has the same operator scheme Σ_S as the source algebra, but whose carrier sets are populated by values from the target algebra and whose operations are the derived operations defined by the operator map om . The generalized homomorphism $H: \mathcal{A}_{\Sigma_S} \rightarrow \mathcal{A}_{\Sigma_T}$ is thus the composition of an *embedding* homomorphism from $\mathcal{A}_{\Sigma_S}$ to the intermediate algebra $\mathcal{A}_{\Sigma_S}^{ST}$, ($em: \mathcal{A}_{\Sigma_S} \rightarrow \mathcal{A}_{\Sigma_S}^{ST}$) with an identity *injection mapping* from the intermediate algebra to $\mathcal{A}_{\Sigma_T}$, ($im: \mathcal{A}_{\Sigma_S}^{ST} \rightarrow \mathcal{A}_{\Sigma_T}$) [9,17]. The mapping im is an identity mapping that maps elements in sort $s, s \in S_S$ in $\mathcal{A}_{\Sigma_S}^{ST}$ to the same value in sort $sm(s) \in S_T$ in $\mathcal{A}_{\Sigma_T}$. Thus $H = im \circ em$. Since both the syntax and semantic generalized homomorphisms of Figure 2 are implemented in this manner, the intermediate algebras form an intermediate Σ -language and thus, the diagram of Figure 2 becomes the commutative diagram in Figure 3.

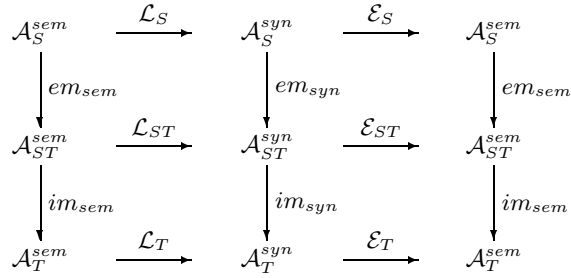


Fig. 3. An algebraic compiler with the intermediate language displayed.

Given a mapping g which maps generators of the source algebra into the target algebra, $g = \{g_s: s \rightarrow sm(s)\}_{s \in S_S}$, g can be uniquely extended to a homomorphism $H: \mathcal{A}_{\Sigma_S} \rightarrow \mathcal{A}_{\Sigma_T}$ [6,9]. The algorithm for implementing a generalized homomorphism from a Σ_S algebra generated by $G = \{g_s\}_{s \in S_S}$ is

$$\begin{aligned}
H(a) = & \textbf{if } a \in g_s \text{ for some } s \in S_S \textbf{ then } g_s(a) \\
& \textbf{else if } a = f(a_1, a_2, \dots, a_n) \text{ for some } f \in Op_S \\
& \textbf{then } om(f)(h(a_1), h(a_2), \dots, h(a_n)).
\end{aligned}$$

This is all made clear by examining it in the context of our model checker as an algebraic compiler. For starters, the sort map sm simply maps the sort F in Σ_{ctl} to the sort Set in Σ_M . The generators G are the set of atomic propositions, $G_F = AP$, and g_F is the function P which maps atomic propositions to their satisfiability sets. What is left then, is to define the operator map om which maps CTL operators in Op_{ctl} to derived operations over satisfiability sets. We saw above how the word “ $S \setminus F_1$ ” could be used to define the derived operation for the CTL operation $not: F_1 \rightarrow F_0$. The use of the indexed sort name F (F_1) as the meta variable is to show the correspondence between the parameters of the source and derived operations. The subscripts are used to distinguish between multiple parameters of the same sort, different sorts will have different names.

Consider now, the CTL operation ax . We cannot write a correct derived operation using only the operators from the target language. We need additional constructs with which to compose a derived operation. It is at this point that we can begin to speak of *meta languages* used in the specification of algebraic compilers instead of just *meta variables*. By introducing some functional language constructs into the language in which we write derived operations, we may like to write the derived operation for ax as

$$om(ax: F_1 \rightarrow F_0) = \text{filter } (\lambda n . succ(n) \subseteq F_1) S$$

where “filter” is a generic operation which applies a boolean function (given by the λ -expression) to each element of a container type, returning a similar container type which contains only those elements from the original which, when provided to the boolean function, generate a value of *true*. Where F_1 is the satisfiability set of a CTL formula f , the derived operation denoted by this term will compute the satisfiability set of the CTL formula $ax f$, by extracting from S , those states that satisfy the condition that all of their successors satisfy f .

Instead of extending the target algebra with these operations, we show in the following section how a meta language containing these constructs can be used in conjunction with the target language to write the appropriate derived operations. The advantage of keeping the meta language separate from the target language is that we can populate an algebraic language processing environment with several reusable meta languages which a language designer may use to build translators.

3.3 Evaluation of Derived Operations

Derived operations are specified by words from the target language syntax algebra $\mathcal{A}_T^{syn}(S'_S)$ over a subscripted set of meta variables from the source signature set of sorts S_S . In Figure 3, the same words from $\mathcal{A}_T^{syn}(S'_S)$ are used to define the operations of the syntax algebra $\mathcal{A}_{ST}^{syn}$ and the semantics algebra $\mathcal{A}_{ST}^{sem}$. Thus, we could build a generalized homomorphism $h': \mathcal{A}_S^{syn} \rightarrow \mathcal{A}_{ST}^{sem}$ which maps words

in $\mathcal{A}_S^{syn}$ directly to values in $\mathcal{A}_{ST}^{sem}$. Thus, h' is the composition of the embedding morphism $em_{syn}: \mathcal{A}_S^{syn} \rightarrow \mathcal{A}_{ST}^{sem}$ and the L_{ST} evaluation function $\mathcal{E}_{ST}$, i.e. $h' = em_{syn} \circ \mathcal{E}_{ST}$. In the case of our model checker, such a homomorphism would map CTL formulas directly to their satisfiability sets in the intermediate semantic algebra. For efficiency reasons this may be desirable and is often the way we will actually implement model checkers as algebraic compilers.

4 Meta Languages in Algebraic Compilers

A meta language L_{ML} used in an algebraic compiler is essentially a parameterized Σ -language. Like all Σ -languages, it has an operator scheme $\Sigma_{ML} = \langle S_{ML}, Op_{ML}, \sigma_{ML} \rangle$ where S_{ML} and Op_{ML} are a set of sorts and operator names as seen above. The signatures of these operator names, however, may include parameters as well as sorts from S_{ML} . That is, $\sigma_{ML}: Op_{ML} \rightarrow PS_{ML}^* \times PS_{ML}$, where PS_{ML} where $PS_{ML} = S_{ML} \cup Param$, $Param$ is a set of parameter names. The meta language has additional constructs that we will use to write the derived operations of the algebraic compiler. In the functional instance of the model checker, these meta language operations will include the “filter” and λ -expression operators we saw above, in the “imperative” instance, the meta language constructs will include *if* and *while* statements, assignment statements, and a *for each* loop operation. These meta operations, in combination with the target language operations of set intersection, union, membership, etc., are used to write the derived operations specifying the model checker.

To write derived operations using meta (L_{ML}) and target (L_T) language operations, an *instantiation* of the meta language is created (by the language processing environment) from these two languages. This language is denoted $L_{MLT} = \langle \mathcal{A}_{MLT}^{sem}, \mathcal{A}_{MLT}^{syn}, \mathcal{L}_{MLT} \rangle$ with operator scheme Σ_{MLT} . To instantiate a meta language the following tasks must be performed:

1. Instantiate the operator scheme Σ_{MLT} . $\Sigma_{MLT} = \langle S_{MLT}, Op_{MLT}, \sigma_{MLT} \rangle$ where the set of sorts S_{MLT} is the union of the meta and target sorts $S_{ML} \cup S_T$, the operator names Op_{MLT} are the union of meta and target operator names $Op_{ML} \cup Op_T$, and signatures in σ_{MLT} are created by replacing parameters in σ_{ML} signatures with sort names in S_T and adding the target languages signatures in σ_T . In our model checker, the target language sorts *Node* and *Set* replace the parameters in the meta language signatures.
2. Instantiate the syntax algebra $\mathcal{A}_{MLT}^{syn}$. We must instantiate the operations for the syntax algebra, but these can be automatically constructed using a prefix format for these “word constructing” operations.
3. Instantiate the semantic algebra $\mathcal{A}_{MLT}^{sem}$. We must also instantiate the operations of this algebra. Either they are explicitly constructed for the new types, a kind of ad hoc polymorphism, or, preferably, the existing meta language operations are *generic* (polymorphic or polytypic) [2] and can thus automatically work on the data-types from the target algebra.

Derived operations for the generalized homomorphism are now written in $\mathcal{A}_{MLT}^{syn}(S'_S)$, the instantiated meta language word algebra with meta variables

S'_S , instead of the syntax algebra $\mathcal{A}_{ST}^{syn}(S'_S)$ of the intermediate hybrid language L_{ST} as done before. Thus, the operator map om used in defining the generalized homomorphism has the signature $om : Op_S \rightarrow \mathcal{A}_{ML^T}^{syn}(S'_S)$. The sort map sm is the same as before, so that target images of source language constructs are still objects of sorts in the target language, not the meta language.

When building such an algebraic compiler the hybrid intermediate language L_{ST} from Figure 3 is replaced by the hybrid intermediate language $L_{SML^T} = \langle \mathcal{A}_{SML^T}^{sem}, \mathcal{A}_{SML^T}^{syn}, \mathcal{L}_{SML^T} \rangle$ as shown in Figure 4. Like L_{ST} , this language has the same operator scheme Σ_S as the source language, but has operations built using the operations from L_{ML^T} . The embedding morphisms em_{syn} and em_{sem} in Figure 4 are computed in the same manner as those in Figure 3. We also add an extra pair of identity injection mappings between L_{SML^T} and L_{ML^T} .

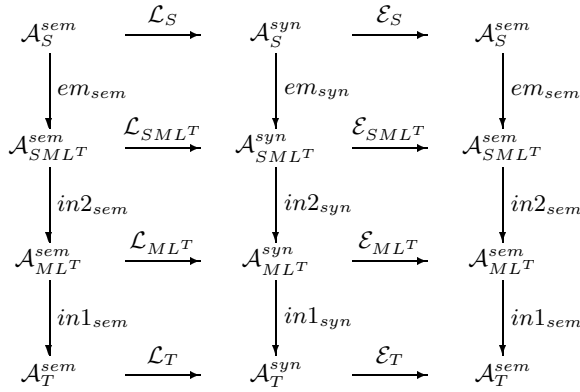


Fig. 4. An algebraic compiler with a meta language level.

Just as the intermediate hybrid language L_{ST} in Figure 3 is automatically created, so it $L_{SML^T} = \langle \mathcal{A}_{SML^T}^{sem}, \mathcal{A}_{SML^T}^{syn}, \mathcal{L}_{SML^T} \rangle$. However, we do need to explicitly create the meta language L_{ML^T} . But, this makes sense, whereas before we specified the source and target language of the algebraic compiler and wrote derived operations in the target syntax algebra with meta variables, we must now specify the meta language we wish to use as well. The derived operations are then written in the instantiated meta language syntax algebra. An appropriate set of algebraic language processing tools can automatically instantiate the meta language, provided the existing meta language operations are generic, but we must at least specify which meta language is to be composed with the selected target language in order to write derived operations and generate the algebraic compiler from these specifications.

4.1 A Functional Meta Language

As alluded to above, we can use a functional meta language in specifying our algebraic model checker $MC: L_{ctl} \rightarrow L_M$. This allows us write derived operations for the temporal logic operators ax , ex , au , and eu using functional language constructs and thus provide concise specifications for our model checker. Although a full functional meta language would have many higher order functions, like *map* and *fold*, we only describe here the operations which are used in our algebraic specification. We do however use λ expressions and higher order functions *filter*, *limit* and *iterate* which are defined below.

Our functional meta language $L_{FM} = \langle \mathcal{A}_{FM}^{sem}, \mathcal{A}_{FM}^{syn}, \mathcal{L}_{FM} \rangle$ has operator scheme $\Sigma_{FM} = \langle S_{FM} = \{Boole, Var, Func(\rightarrow), List([])\}, Op_{FM} = \{not, and, filter, \lambda, limit, iterate\}, \sigma_{FM} \rangle$, where σ_{FM} is defined below:

$$\begin{aligned} \sigma_{FM}(not) &= Boole && \rightarrow Boole \\ \sigma_{FM}(and) &= Boole \times Boole && \rightarrow Boole \\ \sigma_{FM}(filter) &= (a \rightarrow Boole) \times b && \rightarrow b \\ \sigma_{FM}(\lambda) &= Var_b \times a && \rightarrow (b \rightarrow a) \\ \sigma_{FM}(limit) &= [a] && \rightarrow a \\ \sigma_{FM}(iterate) &= (a \rightarrow a) \times a && \rightarrow [a] \end{aligned}$$

The *Boole* sort is for boolean variables, *Var* for variables used in λ -expressions, *Func* for functions between two types, denoted $a \rightarrow b$ for respective source and target types a and b , and *List*, denoted $[a]$ for lists of elements of type a . *filter* is a generic operation which applies a boolean function to each element (parameter a) of a container type (parameter b), and constructs the container type with only those original elements which evaluate to *true* under the boolean function. λ is the operation for creating functions from λ -expressions. The parameter a in this signature represents an expression of type a with a free variable of type b which when combined with a variable of type b , (Var_b) generates a function of type $b \rightarrow a$. *limit* is a function which lazily evaluates a list of elements, returning the first element in the list which is followed by a element of the same value (limit $[1, 2, 3, 3, \dots]$ evaluates to 3). *iterate* is also lazy and repeatedly applies a unary function first using a given initial value and then to the value returned from the previous application. That is, *iterate* f $x = x$ cons (*iterate* f (f x)) (for example *iterate* *inc* 3 = $[3, 4, 5, 6, \dots]$).

We can instantiate this meta language with the model language L_M by writing new operator signatures by replacing the parameters a and b in σ_{FM} with sort names *Set* and *Node* from the operator scheme Σ_M of L_M . Since the *filter* operation is generic [2], we do not need to explicitly implement versions of this function for sets.

4.2 An Imperative Meta Language

We can similarly design an imperative meta language $L_{IM} = \langle \mathcal{A}_{IM}^{sem}, \mathcal{A}_{IM}^{syn}, \mathcal{L}_{IM} \rangle$ that has operator scheme $\Sigma_{IM} = \langle S_{IM}, Op_{IM}, \sigma_{IM} \rangle$. The sort set contains sorts $S_{IM} = \{Expr, Stmt, StmtList, Var, Boole\}$ for expressions, statements,

statement lists, etc., as are familiar in imperative languages. A set of operators Op_{IM} would thus include the set $\{if, while, assign, block, for\ each, not, and, \dots\}$. These operator's signatures and others as defined by σ_{IM} are shown below:

$$\begin{aligned}
\sigma_{IM}(not) &= Boole && \rightarrow Boole \\
\sigma_{IM}(and) &= Boole \times Boole && \rightarrow Boole \\
\sigma_{IM}(foreach) &= Var \times Expr \times Stmt && \rightarrow Stmt \\
\sigma_{IM}(if) &= Boole \times Stmt && \rightarrow Stmt \\
\sigma_{IM}(while) &= Boole \times Stmt && \rightarrow Stmt \\
\sigma_{IM}(assign) &= Var \times Expr && \rightarrow Stmt \\
\sigma_{IM}(block) &= StmtList && \rightarrow Stmt \\
\\
\sigma_{IM}(list_1) &= Stmt && \rightarrow StmtList \\
\sigma_{IM}(list_2) &= StmtList \times Stmt && \rightarrow StmtList \\
\sigma_{IM}(expr_1) &= Expr && \rightarrow a \\
\sigma_{IM}(expr_2) &= Expr && \rightarrow Boole \\
\sigma_{IM}(valof) &= Stmt && \rightarrow Expr
\end{aligned}$$

The familiar imperative language operations are present here. Of interest is the generic *for each* operation which will iterate through all elements of a container type, and perform some statement for each element and the *valof* operation which embeds statements in expressions using the value of the last assignment.

5 Model Checker Specification

In this section we can show the specifications for the algebraic model checker using the functional and imperative meta languages. We will write the translation specifications for each CTL operation $op \in Op_{ctl}$, by writing the signature of the operation, $\sigma_{ctl}(op)$, followed by its derived operation in the target, $om(op)$, but we will drop the om for convenience. The operation's signatures are written with the output sort of each operation to the left and the operation name split between the input sorts in a BNF notation. (In fact, some algebraic tools like TICS [11] use this specification to generate a parser for the source language.) The meta variables used in the derived operations are indexed source language sorts found in the source operation signature. In the derived operations, a meta variable for an input sort represents the target image of the corresponding source language component. These specifications are processed by an algebraic language processing environment to automatically generate the model checker [12,13].

5.1 Functional Meta Language Specification

The functional version of the algebraic model checker maps CTL formulas in $\mathcal{A}_{ctl}^{syn}(AP)$ to their satisfiability sets. For the non-temporal operators in L_{ctl} we have straightforward derived operations shown below:

$$\begin{array}{llll}
F_0 ::= \text{true} & F_0 ::= \text{false} & F_0 ::= \text{not } F_1 & F_0 ::= F_1 \text{ and } F_2 \\
S & \emptyset & S \setminus F_1 & F_1 \cap F_2
\end{array}$$

The operation *true* has the derived operation S (shown directly below it) indicating that the satisfiability set of *true* is the full set of states S in the model M ; *false* has derived operation $\emptyset$ indicating that the satisfiability set of *false* is the empty set. The derived operation associated with *not* shows that the satisfiability of *not* f is the set difference of S and the satisfiability set of f , denoted by the sort name F_1 . Similarly, *and* is defined by the intersection of the satisfiability sets of the two sub formulas, respectively denoted F_1 and F_2 .

In the derived operation for *ax*, seen below, we see the use of some meta language constructs. Here, we define the satisfiability set of *ax* f by filtering the set of states by a function which selects only those nodes such that all of their successors are in the satisfiability set of f .

$$F_0 ::= \text{ax } F_1 \qquad \text{filter } (\lambda n . \text{succ}(n) \subseteq F_1) S$$

The derived operation for *au* is similar, but uses the *limit* and *iterate* operations to implement a type of least fixed point operator of the function specified by the λ -expression.

$$\begin{array}{l}
F_0 ::= a [F_1 \text{ u } F_2] \\
\text{limit } (\text{iterate } (\lambda z . z \cup (\text{filter } (\lambda n . (\text{succs}(n) \subseteq z)) F_1)) F_2)
\end{array}$$

The atomic propositions, specified as variables AP in $\mathcal{A}_{ctl}^{syn}(AP)$, are mapped to their satisfiability set by P , the model labeling function.

$$F_0 ::= p \qquad P(p)$$

5.2 Imperative Meta Language Specification

Since the non-temporal CTL operators do not use any meta language constructs in their derived operations, they are the same here as in the functional specification. Thus, we show only the temporal operators *ax* and *au*. We use an additional meta variables $\$temp_i$ in these derived operations which are replaced by a new temporary variables for each use of a derived operation.

$$\begin{array}{ll}
F_0 ::= \text{ax } F_1 & F_0 ::= a [F_1 \text{ u } F_2] \\
\text{valof } \{ & \text{valof } \{ \\
\quad \$temp := \emptyset & \quad \$temp_1 := \emptyset; \$temp_2 := F_2 ; \\
\quad \text{for each } n \text{ in } S & \quad \text{while } (\text{not } \$temp_1 == \$temp_2) \{ \\
\quad \quad \text{if } (\text{succ}(n) \subseteq (F_1) \text{ then} & \quad \quad \$temp_1 := \$temp_2 ; \\
\quad \quad \quad \$temp := \$temp \cup \{ n \} & \quad \quad \text{for each } n \text{ in } F_1 \text{ do} \\
F_0 := \$temp \} & \quad \quad \text{if } (\text{succ}(n) \subseteq \$temp_1) \text{ then} \\
& \quad \quad \quad \$temp_2 := \$temp_2 \cup \{ n \} \} \\
& \quad F_0 := \$temp_1 \}
\end{array}$$

These derived operations are the imperative versions of the functional derived operations given above in Section 5.1. Here, the *while* and *for each* operators are used to implement a least fixed point operation to compute satisfiability sets.

6 Comments and Future Work

The meta languages described here are just the required subsets of general purpose meta languages which would populate an algebraic language processing environment. Meta languages should be reusable components in such an environment so that algebraic compiler designers can choose from a collection of existing meta languages in which to write their translator specifications. A well-stocked environment would have functional and imperative style meta languages giving the language designer some choice based on personal preference of language style.

More importantly, however, we would also expect this environment to contain *domain specific meta languages* [18] with specialized constructs to address issues found in specific domains commonly encountered in language processing as well as other domains, such as temporal logic model checking, which also have solutions as algebraic compilers. Traditional language processing tasks with specific domains include type checking, optimization and parallelization, and code generation. In a type checker, for example, the target algebras would have operators for the base types and type constructors and carrier sets containing types or type expressions. A domain specific meta language for type checking which has specific constructs for managing symbol tables and environments would be helpful to the implementor and reusable in different compilers. In the case of the model checker, a domain specific meta language would include a least fixed point operator, since this domain would make good use of such a construct.

We opened this paper with a mention of attribute grammars and comment here on meta languages within attribute grammars since they take a slightly different form than in algebraic compilers. Algebraic compilers rely on an explicit definition of the target language and use target language operations for writing derived operations. These operations thus provide a starting point for adding meta language features. Attribute grammars, to their detriment, make no explicit mention of the target language and thus do not have a set of target language operations to provide as a starting point for writing semantic functions for defining attribute values. Instead, they provide a single general purpose language for writing semantic functions. This language doesn't suffer the expressiveness problems we saw above, but it does lock the user into a single "meta language" for defining attribute values. We have thus argued [18] that a choice of domain specific meta languages in attribute grammars is also desirable for many of the same reasons as they are beneficial in algebraic compilers.

We are pursuing this work in an effort to find appropriate meta languages for defining language constructs for the Intentional Programming (IP) [15] system under development at Microsoft. IP is an extensible programming environment which allows programmers to define their own language constructs, called *intentions*, and add them to their programming environment. We are interested in exploring different meta languages, in the broad sense of the term, for defining such intentions. Since the same domains of type checking, optimization, code generation, etc., are encountered in IP, domain specific meta languages will be useful in this system as well. They are especially important here since appropri-

ate domain specific meta languages raise the level of abstraction in which the intention designer works and will thus make designing intentions a more reasonable process that experienced programmers could perform to create their own language extensions. To experiment with different meta languages, we are currently developing a set of lightweight prototype tools using algebraic compilers and attribute grammars which use domain specific meta languages.

Our choice of model checking as an example isn't as esoteric as it may appear. Model checking has been used to perform data flow analysis on program control and data flow graphs [16] and to find optimization and parallelization opportunities in program dependency graphs [14]. In both cases, temporal logic acts as a specification language for certain patterns in a graph representation of the program which are found by a model checker. Thus, temporal logic does have applications as a domain specific meta language in algebraic compilers, attribute grammars and IP.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
2. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.
3. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.
4. P.M. Cohn. *Universal Algebra*. Reidel, London, 1981.
5. M. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1988.
6. P.J. Higgins. Algebras with a scheme of operators. *Mathematische Nachrichten*, 27:115–132, 1963/64.
7. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in 5(2):95–96, 1971.
8. S. Kripke. Semantical analysis of modal logic i: Normal modal propositional calculi. *Zeitschrift f. Math. Logik und Grundlagen d. Math.*, 9, 1963.
9. T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.
10. T. Rus. Algebraic processing of programming languages. *Theoretical Computer Science*, 199(1):105–143, 1998.
11. T. Rus, T. Halverson, E. Van Wyk, and R. Kooima. An algebraic language processing environment. In Michael Johnson, editor, *LNCS 1349*, pages 581–585, Sydney, Australia, 1997.
12. T. Rus and E. Van Wyk. Algebraic implementation of model checking algorithms. In *Third AMAST Workshop on Real-Time Systems, Proceedings*, pages 267–279, March 6 1996. Available from URL: <http://www.comlab.ox.ac.uk/oucl/work/eric.van.wyk/>
13. T. Rus and E. Van Wyk. Integrating temporal logics and model checking algorithms. In *Fourth AMAST Workshop on Real-Time Systems, Proceedings, LNCS 1231*. Springer-Verlag, May 21 1997.

14. T. Rus and E. Van Wyk. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, 8(4):459–471, 1998.
15. C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1., 1996. Available from URL:
<http://www.research.microsoft.com/research/ip/>
16. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115–139, 1993.
17. E. Van Wyk. *Semantic Processing by Macro Processors*. PhD thesis, The University of Iowa, Iowa City, Iowa, 52240 USA, July 1998.
18. E. Van Wyk. Domain specific meta languages. In *ACM Symposium on Applied Computing*, March 19–21 2000.

Random Access to Abstract Data Types

Martin Erwig

FernUniversität Hagen, Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract. We show how to define recursion operators for random access data types, that is, ADTs that offer random access to their elements, and how algorithms on arrays and on graphs can be expressed by these operators. The approach is essentially based on a representation of ADTs as bialgebras that allows catamorphisms between ADTs to be defined by composing one ADT's algebra with the other ADT's coalgebra. The extension to indexed data types enables the development of specific recursion schemes, which are, in particular, suited to express a large class of graph algorithms.

Keywords: Category Theory, ADT, Catamorphism, Graph Algorithm

1 Introduction

In [6] we have proposed to model abstract data types as bialgebras, that is, as (algebra, coalgebra)-pairs with a common carrier. In this approach a program on an ADT D can be defined by a mapping to another ADT D' , and such a mapping, called *metamorphism*, is essentially given by composing the algebra of D' with the coalgebra of D . This offers much freedom in specifying ADTs and mappings between them. It also provides a new programming style encouraging the compositional use of ADTs. The proposed approach essentially uses existing concepts, such as algebra and coalgebra, on a higher level of abstraction, and this is the reason that all the laws developed for algebraic data types can still be used for program transformation and optimization in this extended framework. But in addition to this, the “programming by ADT composition” style offers some new optimization opportunities: for example, since intermediate ADTs are intrinsically used in a single-threaded way, a compiler can automatically insert efficient update-in-place implementations for them [7].

ADTs (as well as algebraic data types) are restricted in the sense that the decomposition (order) cannot be controlled from the outside. In other words, the decomposition of ADT values is completely determined by themselves in advance. This makes the treatment of some data types, such as arrays or graphs, difficult. In these data types the decomposition is often controlled by explicitly given indices (respectively, nodes) telling which parts of the ADT value are to be processed next. We call data types that offer such an index access *random access data types* or simply *indexed data types*. Index access behavior can, in

principle, be realized in the ADT approach by appropriately defining new, compound ADTs that contain the ADT values (array, graph) to be indexed as well as the index values. However, this results in rather complex definitions that are difficult to comprehend.

A different solution is proposed in this paper: first, we generalize the definition of ADT to IDT (*indexed data type*). Essentially, this means to extend the argument type of the destructor so that it has explicit access to index values. This leads to a definition of an IDT as a *trialgebra*. Second, the definition of metamorphism is generalized to take into account the use and dynamic generation of index values. This generalization comes in two flavors: first, for data types like arrays having to deal with only one index value at a time, a simple construction, called *exomorphism*, suffices. However, in the more general case, for example, when dealing with graphs, collections of index values must be handled, and this requires a much more involved definition in which two data types, the primary one and an auxiliary one for storing index values, are processed simultaneously. This general mapping is called *sychromorphism*.

The paper is structured as follows: after describing related work in the next section, we briefly review the general categorical approach to data types in Section 3 followed by an introduction to our bialgebra approach to abstract data types in Section 4. The generalization to indexed data types is then described in Section 5. A simple way to map between IDTs with is shown in Section 6, and the development of a more powerful kind of morphisms is presented in Section 7. Conclusions in Section 8 complete the paper.

2 Related Work

The so-called Bird/Meertens formalism [1,16] is concerned with the derivation of programs from specifications. Essential in that approach is the use of a few powerful operators, like *catamorphisms* (also called *fold* or *reduce*), instead of general recursion. Their work is originally based on lists only, but it has been extended to arbitrary inductively defined data types [15,17,18,19,9]: a data type is given by a morphism which is a fixed point of a functor defining the signature of the data type. Since fixed points are initial objects, homomorphisms to other data types are uniquely defined, and this makes it possible to specify a program on a data type by simply selecting an appropriate target data type.

Work on program optimization has profited a lot from the categorical approach: when programs are expressed as catamorphisms (or even better as hylo-morphisms), powerful fusion laws can be used to eliminate intermediate data structures [13,19,20].

The categorical framework has been almost always applied to algebraic data types, that is, data types that are just given by free term structures. The only general approach for expressing catamorphisms over non-free data types we know of is the work of Fokkinga [11,10]. The idea is to represent terms by combinators called *transformers* and to represent an equation by a pair of transformers. Several properties of transformers are investigated, and it is shown how transform-

ers can be combined to yield new transformers thus resulting in a variable-free language for expressing equations. The use of transformers is demonstrated in showing the equivalence of two different stack implementations. Although this works for sub-algebras that satisfy certain laws, one cannot map into algebras with less structure [10,11,14]. This innocent looking restriction means a rather severe limitation of expressiveness: for instance, a program for counting the elements of a set *cannot* be expressed as a catamorphism. This restriction was lifted by the proposal we made in [6].

Some work has been done for specific abstract data types. Interestingly, these are always indexed data types in our sense: Chuang presents in [3] three different views of arrays and defines for each view corresponding fold operations. The first exploits the indexing facilities of arrays, whereas the second views arrays as sequences. In the third view arrays are treated as mappings. We can recover the first two of these views by an IDT based on appropriate coalgebras, and programs on arrays can be conveniently expressed by exomorphisms.

Gibbons [12] defines a data type for directed acyclic multi-graphs. With a careful choice of operations, which obey certain algebraic laws, the definition of graph catamorphisms becomes feasible, and some functions on graphs, such as reversing the edges of a graph, can be expressed as graph catamorphisms. However, the whole approach is very limited since it applies only to acyclic graphs having no edge labels. We have presented a more general view of graphs in [5]. In that paper an important aspect was the definition of a couple of fold operations that can be used to express operations, such as graph reversal, depth first search, evaluation of expression DAGs, or computing all simple paths in a graphs. Two theorems for program fusion were presented that allow the removal of intermediate search trees as well as intermediate graph structures. We can express graph algorithms by synchomorphisms, including depth-first and breadth-first search and even Dijkstra's shortest path algorithm. The view on graph algorithms that is provided by synchomorphisms is similar in spirit to the fixed set of graph exploration operators that were identified in [4].

3 Categorical Data Types

In this section we give a very brief review of the categorical framework for modeling data types. More detailed introductions can be found, for example, in [2,10,17,20]. Examples follow in later sections.

Our default category $\mathcal{C}$ is **CPO**, whose objects are complete partially ordered sets with a least element $\perp$ and whose morphisms are continuous functions. Working in **CPO** guarantees the existence of least fixed points for recursive equations, such as for hylomorphisms and those of Sections 6 and 7.1. We consider polynomial endofunctors on $\mathcal{C}$ which are built by the four basic functors *identity* ($I\ A = A$ and $I\ f = f$), *constant* ($\underline{A}\ B = A$ and $\underline{A}\ f = \text{id}_A$), *product* ($A \times B = \{(x, y) \mid x \in A, y \in B\}$), and *separated sum* ($A + B = \{1\} \times A \cup \{2\} \times B \cup \{\perp\}$). The definition of $\times$ and $+$ on functions is given below with several additional operations:

$$\begin{array}{ll}
(f + g) (1, x) = (1, f x) & (f \times g) (x, y) = (f x, g y) \\
(f + g) (2, y) = (2, g y) & \langle f, g \rangle x = (f x, g x) \\
(f + g) \perp = \perp & \pi_1 (x, y) = x \\
[f, g] (1, x) = f x & \pi_2 (x, y) = y \\
[f, g] (2, y) = g y & \iota_1 x = (1, x) \\
[f, g] \perp = \perp & \iota_2 y = (2, y)
\end{array}$$

For an object x we denote its constant function by $\underline{x}$, that is, $\underline{x} y = x$. (Note also that function application binds strongest, and $\times$ binds stronger than $+$, which in turn binds stronger than composition “ $\circ$ ”.)

Separated sum and product are bifunctors that map from the product category $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$. Fixing one parameter of a bifunctor yields a monofunctor: the (left) *section* of a bifunctor F and an object A is defined as $F_A(B) = F(A, B)$. Thus, for example, $\times_A$ is a monofunctor which takes an object B and maps it to the product $A \times B$.

We will later need the following functors:

$$\begin{array}{ll}
O_A = \underline{1} + \underline{A} & L_A = \underline{1} + \underline{A} \times I \\
P_A = \underline{A} \times I & Q = \underline{1} + I \times I
\end{array}$$

We usually denote $(1, x)$ and $(2, x)$ by ${}_L x$ and ${}_R x$, and we use abbreviations, such as ${}_{LR} x$ for ${}_L({}_R x)$.

Let $F : \mathcal{C} \rightarrow \mathcal{C}$. Then an F -algebra is a morphism $\alpha : F(A) \rightarrow A$. Object A is called the *carrier* of the algebra. We can extract the carrier of an algebra with the forgetful functor U , that is, $U(\alpha) = A$. Dually, an F -coalgebra is a morphism $\varphi : A \rightarrow F(A)$. An F -homomorphism from algebra $\alpha : F(A) \rightarrow A$ to algebra $\beta : F(B) \rightarrow B$ is a morphism $h : A \rightarrow B$ in $\mathcal{C}$ that satisfies $h \circ \alpha = \beta \circ F(h)$.

The category of F -algebras $\mathbf{Alg}(F)$ has as objects F -algebras and as arrows F -homomorphisms. If F is a polynomial functor on **CPO**, $\mathbf{Alg}(F)$ has an initial object, which is denoted by in_F . This means that $in_F : F(T) \rightarrow T$ is an F -algebra with carrier $T = U(in_F)$. For example, the algebraic data type of cons-lists with constructors $[Nil, Cons] : L_{list\ A} \rightarrow list\ A$ is nothing but the initial algebra in_{LA} . Dually, $\mathbf{CoAlg}(F)$ has a terminal object, denoted by out_F , and $out_F : T \rightarrow F(T)$ is an F -coalgebra with the same carrier T as in_F ; in_F and out_F are each other's inverses, and they define an isomorphism $T \cong F(T)$ in **CPO**.

Initial and terminal objects are unique up to isomorphism, and they are characterized by having exactly one morphism to, respectively, from, all other objects. This means that for each F -algebra α in $\mathbf{Alg}(F)$ there is exactly one F -homomorphism $h : in_F \rightarrow \alpha$. Since h is uniquely determined by α , it is conveniently denoted by $\llbracket \alpha \rrbracket_F$; h is called a *catamorphism* [17]. Dually, for each F -coalgebra φ in $\mathbf{CoAlg}(F)$ there is exactly one homomorphism $h : \varphi \rightarrow out_F$, which is denoted by $\llbracket \varphi \rrbracket_F$ and which is called an *anamorphism*. A *hylomorphism* is essentially the composition of a catamorphism with an anamorphism.

Formally, a hylomorphism $\llbracket \alpha, \varphi \rrbracket_F$ is defined as the least morphism h satisfying:

$$h = \alpha \circ F(h) \circ \varphi \quad (\text{HyloDef})$$

Several laws for $\{\text{cata}, \text{ana}, \text{hylo}\}$ -morphisms can be found in [17,20]. The most important result is the fusion rule for hylomorphisms:

$$\llbracket \alpha, \varphi \rrbracket_F \circ \llbracket \beta, \psi \rrbracket_F = \llbracket \alpha, \psi \rrbracket_F \Leftarrow \varphi \circ \beta = \text{id} \quad (\text{HyloFusion})$$

4 Abstract Data Types and Metamorphisms

We define an ADT to be a pair $(\alpha, \bar{\alpha})$ where α is an F -algebra, $\bar{\alpha}$ is an H -coalgebra, and $U(\alpha) = U(\bar{\alpha})$. Such an algebra/coalgebra-pair with a common carrier is called an F, H -bialgebra [11] (where an F, G -bialgebra is a special case of an F, G -dialgebra, that is, $\mathbf{BiAlg}(F, G) = \mathbf{DiAlg}([F, I], [I, G])$ [10]. Working with bialgebras is sufficient for our purposes and makes the separation of constructors and destructors more explicit.) Given an ADT $D = (\alpha, \bar{\alpha})$, we call α the *constructor* of D and $\bar{\alpha}$ the *destructor* of D .

Let us consider two examples. First of all, algebraic data types can be regarded as ADTs by taking the initial algebra as constructor and its inverse as destructor. For example, ADT $List = (in_{L_A}, out_{L_A})$ is an L_A, L_A -bialgebra.

As an example for a non-algebraic type consider an ADT for sets. To define sets based on the “cons”-view given by L_A we take in_{L_A} as the constructor, and the destructor must be defined so that a value is retrieved from a set at most once. This can be realized by splitting off an arbitrary element (for example, the first one) and removing all occurrences of this element in the returned set. With a function *filter* that takes a predicate p and selects a sublist of elements for which p yields true we can first define a further function *remove*:

$$remove(x, l) = filter(\neq x) l$$

Here, the partial application $(\neq x)$ denotes the function $\lambda y. y \neq x$, that is, the predicate that yields true for all values that are not equal to x .

Thus, we can define the set destructor and the set ADT by:

$$\begin{aligned} deset &= I + \langle \pi_1, remove \rangle \circ out_{L_A} \\ Set &= (in_{L_A}, deset) \end{aligned}$$

Note that the definition works only for types A for which equality is defined.

Let $D = (\alpha, \bar{\alpha})$ be an F, H -bialgebra, let $C = (\beta, \bar{\beta})$ be a K, J -bialgebra, and let $D' = (\varphi, \bar{\varphi})$ be an M, N -bialgebra.

Given a natural transformation $f : H \rightarrowtail M$, the *f-metamorphism* from D to D' is defined as the least solution of the equation

$$h = \varphi \circ f \circ H(h) \circ \bar{\alpha} \quad (\text{MetaDef})$$

and is denoted by $D \xrightarrow{f} D'$ (we write $D \rightsquigarrow D'$ if $f = \text{id}$). We call D/D' the *source/target* and f the *map* of the metamorphism. This definition says that a metamorphism from D to D' is essentially a hyloorphism:

$$D \xrightarrow{f} D' = \llbracket \varphi \circ f, \overline{\alpha} \rrbracket_H \quad (\text{MetaHylo})$$

As an important special case, metamorphisms from algebraic data types reduce to catamorphisms, that is,

$$D \rightsquigarrow D' = \llbracket \varphi \rrbracket_H \quad \Leftarrow \quad D = (\text{in}_H, \text{out}_H) \quad (\text{MetaAlg})$$

Let us consider a few examples. Metamorphisms for algebraic data types translate directly from the corresponding catamorphisms. For instance, if we represent the natural numbers by $\text{Nat} = [\text{Zero}, \text{Succ}] = \text{in}_{\underline{1}+I}$, the length of a list can be computed by the metamorphism

$$\text{length} = \text{List} \xrightarrow{I+\pi_2} \text{Nat}$$

Since metamorphisms are based on explicitly defined destructors, we can also count the number of elements in a set:

$$\text{card} = \text{Set} \xrightarrow{I+\pi_2} \text{Nat}$$

The composition of two metamorphisms $C \rightsquigarrow D'$ and $D \rightsquigarrow C$ filters the values of D “through” C before putting them into D' . We thus define the C -*filter* from D to D' as:

$$D \xrightarrow{f} C \xrightarrow{g} D' = C \xrightarrow{g} D' \circ D \xrightarrow{f} C \quad (\text{Filter})$$

Here D and D' are called the *source* and *target* of the filter, and C is called the *filter data type*. Again, we omit f and g if they are just identities.

ADT filters provide a convenient way for expressing certain algorithms, for example,

$$\begin{array}{ll} \text{List} \rightsquigarrow \text{Set} \rightsquigarrow \text{List} & \text{Remove duplicates} \\ \text{List} \rightsquigarrow \text{Heap} \rightsquigarrow \text{List} & \text{Heapsort} \end{array}$$

As for algebraic data types there are several laws for ADTs, see [6,7]. One important result is a generalization of the fusion law for algebraic data types (recall that $C = (\beta, \overline{\beta})$):

Theorem 1 (ADT Fusion). $\overline{\beta} \circ \beta = \text{id} \implies D \rightsquigarrow C \rightsquigarrow D' = D \rightsquigarrow D'$ □

Another very general relationship can be obtained by deriving the “free theorem” [21] for the type of metamorphisms.

Theorem 2 (FreeMeta). *If l is strict, then for any two F, H -bialgebras $D = (\alpha, \overline{\alpha})$ and $D' = (\alpha, \overline{\gamma})$ and two M, N -bialgebras $C = (\varphi, \overline{\varphi})$ and $C' = (\delta, \overline{\varphi})$:*

$$l \circ \varphi = \delta \circ H(l) \wedge \overline{\gamma} \circ r = H(r) \circ \overline{\alpha} \implies l \circ (D \rightsquigarrow C) = (D' \rightsquigarrow C') \circ r \quad \square$$

This general law can be instantiated to many different useful program transformation rules (see [7]).

5 Indexed Data Types

When an ADT is processed by a metamorphism, the decomposition is completely controlled by the ADT itself, that is, the definition of the coalgebra completely determines the decomposition order. (The same is, of course, true for algebraic data types where catamorphisms just follow the term construction.) For some applications, however, it is very useful to have external control over the decomposition of the involved ADT. Consider the simple task of deleting a specific number x from an integer set s . Of course, we can express this by a set-catamorphism that selects from s all elements that are not equal to x , but an even simpler solution is, instead of blindly decomposing all numbers from s , to directly ask for the specific decomposition (x, s') . Then the result is simply given by s' .

This example raises several issues: first, the destructor of such an ADT is not any more simply of type $A \rightarrow H(A)$, but rather of type $G(A) \rightarrow H(A)$ to account for additional arguments (“indices”) for the decomposition. We will therefore extend the definition of ADT into an “IDT”. Second, the requested decomposition might not be possible at all, for example, in the above example x might not be contained in s . This affects the definition of mappings from such IDTs, which has to handle such cases. Finally, we need a way to specify how index values are generated during (or fed into) the decomposition process. In the simplest case the next index can be computed by a function parameter; we will consider this case in Section 6. The more general case is treated in Section 7.

We start by generalizing the definition of ADT. An *indexed data type* (IDT) is a pair $D = (\alpha, \bar{\alpha})$ where α is an F -algebra, $\bar{\alpha}$ is a G, H -dialgebra, and $U(\alpha) = U(\bar{\alpha})$. We call D an *F, G, H -trialgebra*. Again, α is the *constructor* and $\bar{\alpha}$ is the *destructor* of D . As an example consider the above set ADT with random access to its elements: we have $\alpha = in_{L_A}$ and $\bar{\alpha} = extract$ with

$$\begin{aligned} split(x, s) &= \langle filter(=x), filter(\neq x) \rangle s \\ extract(x, s) &= \begin{cases} \text{L}() & \text{if } \pi_1 \circ split(x, s) = Nil \\ \text{R}((hd \times I \circ split)(x, s)) & \text{otherwise} \end{cases} \end{aligned}$$

In this example we have $F = H = L_A$ and $G = P_A$. The example also illustrates that we use the term “index” in a rather broad sense: an index can be any value that controls the decomposition of an IDT. Hence, “index” is just a name for a particular role of a type.

Arrays are probably the most prominent representatives of IDTs; they are typically used whenever indexed access is needed. A simple array constructor is given by $in_{L_{(X \times A)}}$ where X and A denote the index type and the type of stored elements. The treatment of duplicate index entries can happen within the array destructor. A simple approach is to define a function *dearr* that takes an index i and an array a (which is represented by a list of pairs) and returns the first pair (i, x) and the array without all those pairs (j, x') for which $j = i$. (This realizes the behavior that newer entries in the array “overwrite” older ones.) If the index

is not contained in the array, the unit value $() : \mathbf{1}$ is returned. Thus, *dearr* is a $P_X, L_{(X \times A)}$ -dialgebra, and the IDT defined by $Array = (in_{L_{(X \times A)}}, dearr)$ is an $L_{(X \times A)}, P_X, L_{(X \times A)}$ -trialgebra.

Graphs are another example for IDTs. In the inductive view of directed graphs we have proposed in [5] graphs can be constructed by two constructors: *empty*, which denotes the empty graph without any nodes, and *embed*, which extends a graph by a *node context*, that is, a labeled node together with its incoming and outgoing edges. To stay with polynomial functors we need a functor $_{(k)}$ for denoting lists of length not greater than k : $X^{(0)} = \mathbf{1}$ and $X^{(k+1)} = \mathbf{1} + X^{(k)} \times X$.

Now the type of node contexts for node type X and label type Y is given by the following bifunctor:

$$Ctx(X, Y) = X^{(k)} \times X \times Y \times X^{(k)}$$

that is, a four-tuple consisting of a list of nodes (predecessors), a node, a label, and another list of nodes (successors). The type of graphs of bounded in- and out-degree of k is then defined by the following ternary functor:

$$Gr(X, Y, G) = \mathbf{1} + (Ctx(X, Y) \times G)$$

Then the graph constructor is given by a $Gr_{X,Y}$ -algebra $[empty, embed]$. (For a precise semantics of *empty* and *embed*, see [5].)

The graph destructor *degraph* essentially retrieves and removes a specific node context from the graph. This means, given a node x and a graph g , *degraph*(x, g) returns a pair (c, g') where $c = (p, x, l, s)$ is the context of x and where g' is g without x and its incident edges. If x is not contained in g , *degraph* yields $()$. Thus, $degraph : X \times G \rightarrow Gr_{X,Y}(G)$, and we obtain a graph IDT by the $Gr_{X,Y}, P_X, Gr_{X,Y}$ -trialgebra

$$Graph = ([empty, embed], degraph)$$

6 Exomorphisms

The recursion in a metamorphism h is realized by applying $H(h)$ to the result of $\bar{\alpha}$ which works fine because $\bar{\alpha}$ has type $A \rightarrow H(A)$. Since the destructor of an IDT is a G, H -dialgebra, that is, $\bar{\alpha} : G(A) \rightarrow H(A)$, we *cannot* simply express the recursion by $H(h) \circ \bar{\alpha}$ since $\bar{\alpha}$, and thus h , too, applies to $G(A)$ -values and not simply A -values. Therefore, we have to prepare the recursion by first applying a function $g : H(A) \rightarrow H(G(A))$ which, in fact, supplies index values for all the recursive occurrences of A -values.

Let $D = (\alpha, \bar{\alpha})$ be an F, G, H -trialgebra with $A = U(\alpha) = U(\bar{\alpha})$, and let $D' = (\varphi, \bar{\varphi})$ be an M, N -bialgebra. Given two functions

$$f : H(C) \rightarrow M(C) \qquad g : H(A) \rightarrow H(G(A))$$

we define the *exomorphism* from D to D' as the least morphism satisfying:

$$h = \varphi \circ f \circ H(h) \circ g \circ \bar{\alpha}$$

We denote h by $D \xrightarrow{f}_g D'$. Since g is a parameter of the exomorphism, it provides control over the IDT decomposition from the “outside”.

As a simple example, consider a store of linked lists implemented with arrays: each array cell consists of a pair (x, p) representing a cell where p is an integer pointing to the next cell. With the function $g = I + \langle \pi_1, \langle \pi_2 \circ \pi_1, \pi_2 \rangle \rangle$ that pairs the pointer of the decomposed array cell $(\pi_2 \circ \pi_1)$ with the remaining array (π_2) and pairs this with the found list entry (π_1) , we can retrieve the list stored in A beginning at position i by

$$(Array \xrightarrow{g} List)(i, A)$$

7 Synchronomorphisms

We have already seen that the next index value depends, in general, on preceding decompositions. This means that index generation happens dynamically; it must be performed “on the fly” during the decomposition of the IDT, hence, the sequence of indices is generally not known in advance.

The limitations of exomorphisms are mainly due to their inability to handle more than one index at a time, that is, we are missing an option to intermediately store collections of indices. Now ADTs themselves are suited very well for this index buffering, and when we are going to define a recursion scheme for IDTs in Section 7.1, this will in fact turn out to be a scheme for processing the IDT with the buffer ADT hand in hand. In Section 7.2 we present some examples.

7.1 Buffered Decomposition of IDTs

A synchronomorphism takes three arguments: a source IDT, a target ADT, and a buffer ADT for storing and delivering index values. A synchronomorphism informally works as follows: the IDT is decomposed, and (i) from the result some fresh index values are computed that are inserted into the buffer ADT, and (ii) a part of the result is inserted into the target ADT. Immediately after that the buffer is requested to yield a new index which is then used in the next iteration to decompose the remaining IDT-value.

Let $D = (\alpha, \bar{\alpha})$ be an F, G, H -trialgebra with $A = U(\alpha) = U(\bar{\alpha})$, let $D'' = (\beta, \bar{\beta})$ be a K, J -bialgebra with $B = U(\beta) = U(\bar{\beta})$, and let $D' = (\varphi, \bar{\varphi})$ be an M, N -bialgebra with $C = U(\varphi) = U(\bar{\varphi})$. X is the type of index values. It is shared between the types of D and D'' , and we assume that all functors F, G, H, K and J are left sections of bifunctors having X fixed as their first argument.

Recall the roles of the functors: F and G define the argument type of the source ADT constructor and destructor, respectively, and H defines the result type of the source ADT destructor. K defines the argument type of the buffer ADT constructor, and J defines the result type of the buffer ADT destructor. This means that D and D'' carry index values, whereas the target ADT D' does, in general, not.

In the following we use variable names that indicate their type: for example, x_G is an element of $G(A)$, and x_{HG} an element of $H(G(A))$. We develop the definition of synchronomorphisms step by step, collecting requirements and incrementally fixing design decisions. The construction is summarized in Figure 1.

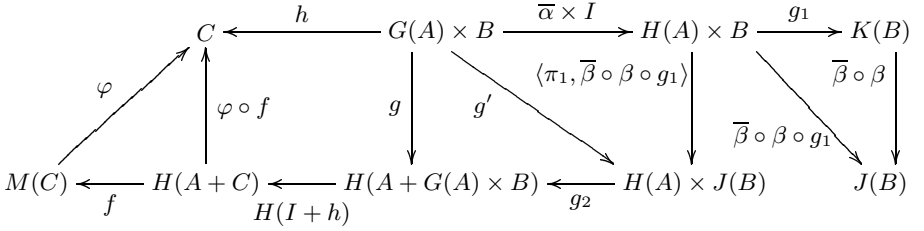


Fig. 1. Categorical Definition of Synchronomorphisms.

First, a synchronomorphism (h) takes an IDT-argument and a buffer and produces a value of the target ADT. Therefore, h has the following type:

$$h : G(A) \times B \rightarrow C$$

Since the IDT-destructor yields an element x_H , we have to apply a function g to x_H to enable the recursive application of the synchronomorphism. After the recursive application by H we apply a function f extracting relevant information to be aggregated by the constructor φ of the target ADT D' .

Next we explain how to obtain a suitable definition for g . The synchronomorphism has to perform the following steps:

1. Initially, decompose the IDT with the supplied index, that is, $x_H = \bar{\alpha}(x_G)$.
2. Extract fresh index values j from x_H to be inserted into the buffer. How this should be done is application-specific, and it is specified by a parameter function g_1 .

3. Insert the fresh index values into the buffer, and retrieve the next index value(s) i from the buffer for further decomposition of the IDT. We thus obtain something like

$$i = \dots \circ \bar{\beta} \circ \beta \circ g_1(x_H)$$

The dots “...” indicate that $\bar{\beta}$ actually yields a value x_J of which i is, in general, only a part.

Note that g_1 not only has to extract fresh index values, but also has to arrange them properly around the buffer so that β can be applied. Thus, the result must be of type $K(B)$, and we get:

$$g_1 : H(A) \times B \rightarrow K(B)$$

We can now compose all three steps. With:

$$\begin{aligned}\bar{\alpha} \times I &: G(A) \times B \rightarrow H(A) \times B \\ \bar{\beta} \circ \beta \circ g_1 &: H(A) \times B \rightarrow J(B)\end{aligned}$$

we obtain $g' = \langle \pi_1, \bar{\beta} \circ \beta \circ g_1 \rangle \circ \bar{\alpha} \times I$ of type

$$g' : G(A) \times B \rightarrow H(A) \times J(B)$$

Now two things remain to be done:

4. Combine the remaining IDT(s) d (as part of x_H) and the next index values i into a value x_{HG} that allows d to be decomposed indexed by i (in a recursion structure specified by H). Again this is application-specific and should therefore be specified by a parameter function, say, $g'_2 : H(A) \times J(B) \rightarrow H(G(A))$.

5. Select the resulting buffer b (from x_J) for distribution into x_{HG} , the structure containing the remaining IDT/next index combinations. The selection is directed by the application and requires a further function, say, $g''_2 : J(B) \rightarrow B$ (distribution into x_{HG} could be achieved by H).

Now a problem occurs if x_J does not contain a buffer value at all. This usually will occur if the buffer is exhausted (then x_J will be, for example, $()$). Now in order to not complicate the typings further it seems best to combine g'_2 and g''_2 into one function: $g_2 : H(A) \times J(B) \rightarrow H(G(A) \times B)$ that is supplied by the programmer and that handles all the above cases internally.

But as x_J is not guaranteed to contain buffer values, it might as well fail to produce next index values (again, for example, in the case the buffer is exhausted). In that case we cannot distribute a buffer, and we cannot even build a value x_{GH} . Then we simply pass x_H so that the value can be used by f and φ . Thus we have $g_2 : H(A) \times J(B) \rightarrow H(A) + H(G(A) \times B)$ which can be also written by moving the sum into H :

$$g_2 : H(A) \times J(B) \rightarrow H(A + G(A) \times B)$$

We have not yet discussed the case when $\bar{\alpha}$ fails to produce a new IDT value. This could well happen if the required index decomposition is not possible. In that case the old, undecomposed IDT value (from x_G) should be taken and combined with the buffer (if available, otherwise the recursion stops). But since x_G is not available any more, the easiest solution is to rely on appropriately adapted definitions of H and $\bar{\alpha}$, that is, instead of simply returning $()$, $\bar{\alpha}$ could well be defined to return its argument unchanged whenever decomposition is not possible.

Then (the type of) g_2 need not be changed, and the necessary transformation has to be specified in the definition of g_2 . To summarize, there are four cases to be considered by g_2 depending on the success of $\bar{\alpha}$ and $\bar{\beta}$ in producing new IDT, respectively, index/buffer values:

$\bar{\alpha}$	$\bar{\beta}$	description of case
fail	fail	immediately stop recursion
fail	ok	preserve (old) x_G and continue recursion
ok	fail	just pass x_H and stop decomposition
ok	ok	normal recursion

Whenever $\bar{\beta}$ is successful, the result type of g_2 is $H(G(A) \times B)$, otherwise it is $H(A)$.

Now we can formally define a synchromorphism. Given the functions

$$\begin{aligned} f &: H(A + C) \rightarrow M(C) \\ g_1 &: H(A) \times B \rightarrow K(B) \\ g_2 &: H(A) \times J(B) \rightarrow H(A + G(A) \times B) \end{aligned}$$

the D, D'' -synchromorphism to D' is defined the least solution of the following equation

$$h = \varphi \circ f \circ H(I + h) \circ g$$

where

$$g = g_2 \circ \langle \pi_1, \bar{\beta} \circ \beta \circ g_1 \rangle \circ \bar{\alpha} \times I$$

We denote the synchromorphism h by $D' \xleftarrow[f]{g_1} D \xrightarrow[g_2]{g_1} D''$.

7.2 Examples

Let us begin with expressing depth-first search (dfs) as a synchromorphism. Roughly spoken, dfs decomposes a graph by extracting a particular node context c , pushing the successors from c onto a stack, and extracting the top of the stack to continue graph decomposition. In addition, part of c is aggregated in a target ADT, for example, the visited nodes are put into a list.

Thus, we need a stack buffer with a constructor that can insert lists of nodes. We can use the following ADT defined as a Q, L_X -bialgebra ($++$ is the function for concatenating two lists):

$$Stack = ([Nil, ++], out_{L_X})$$

We can use the ADT *List* for collecting visited nodes, but we have to account for the case that a visited node is not available for insertion into the result list whenever the graph decomposition fails. Therefore, we use the “option” or “maybe”

type O_A to wrap nodes. Finally, we cannot directly use the $Gr_{X,Y}, P_X, Gr_{X,Y}$ -trialgebra *Graph* from Section 5 as IDT since we require the destructor to pass the argument graph if destruction is not possible. We therefore redefine

$$\begin{aligned} \text{degraph}'(x, g) &= (\underline{g} + I) (\text{degraph}(x, g)) \\ \text{Graph} &= ([\text{empty}, \text{embed}], \text{degraph}') \end{aligned}$$

To summarize we have the functors $G = P_X$, $H = I + Ctx_{X,Y} \times I$, $K = Q$, $J = L_X$, and $M = L_{1+X} (= \underline{1} + (\underline{1} + \underline{X}) \times I)$. This gives the following types (A , B , and C are the carriers of graphs, stacks, and lists, and X and Y are the types of nodes and node labels):

$$\begin{aligned} G(A) &= X \times A & H(A) &= A + Ctx_{X,Y} \times A \\ K(B) &= \underline{1} + B \times B & J(B) &= \underline{1} + X \times B \\ M(C) &= \underline{1} + (\underline{1} + X) \times C \end{aligned}$$

Next we define the functions g_1, g_2 and f : g_1 pairs the successors (π_4) of the extracted context (π_1) or an empty list with the buffer and therefore always returns the second alternative (ι_2) of $K(B)$.

$$g_1 = \iota_2 \circ ([\underline{Nil}, \pi_4 \circ \pi_1]) \times I$$

g_2 combines the remaining graph and the next index and distributes fresh indices into the remaining stack. It also has to preserve decomposed values for insertion into D' ; g_2 actually controls the different cases of the recursion:

- (i) If graph decomposition has failed and the stack is exhausted, the recursion is stopped; the graph, which is passed only for typing reasons, will be ignored by f .
- (ii) If graph decomposition failed and a new index i is available, proceed with decomposing the old graph (delivered by the modified *degraph'*) at i .
- (iii) If graph decomposition yields context c and remaining graph g but the stack is exhausted, terminate recursion, and pass c so that the last visited node can be put into the target list.
- (iv) If graph decomposition yields context c and remaining graph g and a new index i is available – this is the “normal” recursion case –, pass c to let f extract the visited node, and continue recursion with decomposing g at i .

For readability we provide a pointwise definition of g_2 .

$$\begin{aligned} g_2 (\text{L } g, \text{L } ()) &= \text{LL } g & g_2 (\text{L } g, \text{R } (i, s)) &= \text{LR } ((i, g), s) \\ g_2 (\text{R } (c, g), \text{L } ()) &= \text{R } (c, \text{L } g) & g_2 (\text{R } (c, g), \text{R } (i, s)) &= \text{R } (c, \text{R } ((i, g), s)) \end{aligned}$$

Finally, the definition for f follows the structure of results yielded by g_2 : (i) on termination, a unit value is returned which is mapped by the target ADT into *Nil*. (ii) If no context is available, build a pair of $\text{L } ()$ and the recursively computed list of nodes. The “none” value $()$ will be inserted into this list and

can be eventually removed by applying a post-processing function keeping only the nodes. (iii,iv) Extract the visited node from the context and pair it with the empty list (on termination (iii)) or the recursively computed list (in case (iv)).

$$\begin{aligned} f(\text{L}g) &= \text{L}() & f(\text{L}l) &= \text{R}(\text{L}(), l) \\ f(\text{R}(c, \text{L}g)) &= \text{R}(\text{R}(\pi_2(c)), \text{Nil}) & f(\text{R}(c, \text{R}l)) &= \text{R}(\text{R}(\pi_2(c)), l) \end{aligned}$$

Now we can define dfs as follows:

$$\text{dfs} = \text{List} \xleftarrow{f} \text{Graph} \xrightleftharpoons[g_2]{g_1} \text{Stack}$$

It is also obvious how to express breadth-first search: we can just substitute a queue buffer for the stack buffer, and we obtain:

$$\text{bfs} = \text{List} \xleftarrow{f} \text{Graph} \xrightleftharpoons[g_2]{g_1} \text{Queue}$$

We can also express more complex algorithms, for example, Dijkstra's algorithm for finding shortest paths. This is shown in the long version of this paper [8].

We have only shown graph algorithms as examples for synchronomorphisms (further examples are: Prim's minimum spanning tree algorithm and Kruskal's minimum spanning tree algorithm), and in fact, expressing graph algorithms as instances of a fixed recursion scheme was the main motivation for developing synchronomorphisms. However, we believe that there are different application areas. For example, the plane-sweep paradigm for algorithms of computational geometry seems to fit the presented scheme: the buffer ADT can be used to implement the sweep-line status structure, and the IDT is a collection of geometric objects (which, however, is scanned in fixed order most of the time so that indexed access is not always needed).

8 Conclusions

We have demonstrated how to extend categorical abstract data types to indexed data types, and we have shown definitions of recursion operators operating on these data types. With these combinators we can now express algorithms that use data types in a random access manner.

The next step is to investigate the transformation of such algorithms into efficient programs. This can go along the same line as in [7] by introducing libraries of efficient ADT implementations and defining simple optimizing transformations that automatically select these implementations.

References

1. R. S. Bird. Lectures on Constructive Functional Programming. In M. Broy, editor, *Constructive Methods in Computer Science*, NATO ASI Series, Vol. 55, pages 151–216, 1989.

2. R. S. Bird and O. de Moor. *The Algebra of Programming*. Prentice-Hall International, 1997.
3. T.-R. Chuang. A Functional Perspective of Array Primitives. In *2nd Fuji Int. Workshop on Functional and Logic Programming*, pages 71–90, 1996.
4. M. Erwig. Graph Algorithms = Iteration + Data Structures? The Structure of Graph Algorithms and a Corresponding Style of Programming. In *18th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 657, pages 277–292, 1992.
5. M. Erwig. Functional Programming with Graphs. In *2nd ACM Int. Conf. on Functional Programming*, pages 52–65, 1997.
6. M. Erwig. Categorical Programming with Abstract Data Types. In *7th Int. Conf. on Algebraic Methodology and Software Technology*, LNCS 1548, pages 406–421, 1998.
7. M. Erwig. The Categorical Imperative – Or: How to Hide Your State Monads. In *10th Int. Workshop on Implementation of Functional Languages*, pages 1–25, 1998.
8. M. Erwig. Random Access to Abstract Data Types. Technical Report 266, Fern-Universität Hagen, 2000.
9. L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. In *23rd ACM Symp. on Principles of Programming Languages*, pages 284–294, 1996.
10. M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 1992.
11. M. M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.
12. J. Gibbons. An Initial Algebra Approach to Directed Acyclic Graphs. In *Mathematics of Program Construction*, LNCS 947, pages 282–303, 1995.
13. A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *Conf. on Functional Programming and Computer Architecture*, pages 223–232, 1993.
14. J. T. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, University of Utrecht, 1993.
15. G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, LNCS 375, pages 335–347, 1989.
16. L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. In *CWI Symp. on Mathematics and Computer Science*, pages 289–334, 1986.
17. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Conf. on Functional Programming and Computer Architecture*, pages 124–144, 1991.
18. E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. on Functional Programming and Computer Architecture*, pages 324–333, 1995.
19. T. Sheard and L. Fegaras. A Fold for all Seasons. In *Conf. on Functional Programming and Computer Architecture*, pages 233–242, 1993.
20. A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. In *Conf. on Functional Programming and Computer Architecture*, pages 306–313, 1995.
21. P. Wadler. Theorems for Free! In *Conf. on Functional Programming and Computer Architecture*, pages 347–359, 1989.

A Monad for Basic Java Semantics

Bart Jacobs and Erik Poll

Dept. of Computer Science, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{bart, erikpoll}@cs.kun.nl
<http://www.cs.kun.nl/~{bart, erikpoll}>

Abstract This paper describes the role of a computational monad in the denotational semantics of sequential Java and investigates some of its properties. This denotational semantics is an abstraction of the one used for the verification of (sequential) Java programs using proof tools, see [11,15].

1 Introduction

This paper describes the role of a computational monad in the denotational semantics of sequential Java, that has been developed as part of the LOOP project (Logic of Object-Oriented Programming) [11,15]. This Java semantics provides the basis of formal reasoning about Java programs using theorem provers. A compiler has been developed, called the LOOP tool, which, given a sequential Java program, generates its semantics in a form that can serve as input to a theorem prover. The theorem provers currently supported are PVS [13] and Isabelle [14], so the LOOP tool can generate the semantics of a Java program in several PVS or Isabelle theories. One of the aims of the LOOP project is to reason about a real programming language, warts and all; the Java semantics therefore covers all of sequential Java, including details such as exceptions, breaks, and nontermination. It has been used to reason about existing Java programs, for instance to prove an invariant property for the Vector class in the Java standard library [8].

We will not describe the denotational semantics of all of the Java constructs in this paper, but concentrate on the use of a monad to organise the semantics from a single perspective. A denotational semantics of Java is more complicated than the semantics typically considered in textbooks on denotational semantics. Not only does it involve the possibility of nontermination (using the familiar $\perp$), but it also involves different forms of abrupt termination of programs, such as exceptions and the different ways of “jumping” out of methods and repetitions via **break**, **return**, and **continue** statements.

We show that the computational monad approach [12] provides a useful level of abstraction and a good means for organizing all the complications that come with defining the semantics of a real programming language like Java. The paper also provides a *post hoc* justification of the Java semantics as used in the LOOP

project, by giving some of the central properties of the monadic structure and of the interpretation of some particular Java constructs.

Interestingly, the Java semantics used in the LOOP project has originally been developed from a coalgebraic perspective [16,9,10]. This perspective focuses on the state space as a black box, and leads to useful notions like class invariant and bisimilarity. The computational monad view is different. It keeps the state space fixed and focuses on the (functional) input-output behaviour. It took us some time to fully appreciate these differences, and to get the most out of both approaches in program semantics.

Not all aspects of the Java semantics are described in this paper; the representation of the global state space, essentially a mapping from references to values, is discussed [2], and the treatment of inheritance is discussed in [6].

This paper is organised as follows. It starts in Section 3 with a sketch of the Java semantics as used in the LOOP project, focusing on the different abnormalities in Java. In the monadic approach in Section 4 these abnormalities are simplified to a single set E . This leads to a monad J . Its Kleisli composition corresponds to Java composition, and its extension to Java extension. Furthermore, the homsets in its Kleisli category have a cpo structure. Next, in Section 5, while statements and recursive statements are studied in this framework. It is shown how an operational definition of while, taking abnormalities into account, can be described as a least fixed point—like in standard denotational semantics. The cpo structure of Kleisli homsets allows us to deal with recursive statements in the usual way. This whole computational monad approach has (also) been formalised in PVS. This is briefly discussed in the final section 6.

2 Preliminaries

We shall make frequent use of n -ary products $X_1 \times \cdots \times X_n$ of sets X_i , with projection functions $\pi_i: X_1 \times \cdots \times X_n \rightarrow X_i$. The empty product, when $n = 0$, describes a singleton set, which is written as $1 = \{*\}$. We also use n -ary coproducts (or disjoint unions) $X_1 + \cdots + X_n$ with coprojection (or injections) $\kappa_i: X_i \rightarrow X_1 + \cdots + X_n$. There is an associated “case” construction which is perhaps not so familiar: given n functions $f_i: X_i \rightarrow Y$, there is a unique function¹ $f: X_1 + \cdots + X_n \rightarrow Y$ with $f \circ \kappa_i = f_i$, for all $1 \leq i \leq n$. We shall write

$$f(z) = \text{CASES } z \text{ OF } \left\{ \begin{array}{l} \kappa_1(x_1) \mapsto f_1(x_1), \\ \vdots \\ \kappa_n(x_n) \mapsto f_n(x_n) \end{array} \right\}$$

This function matches if $z \in X_1 + \cdots + X_n$ is of the form $\kappa_i(x_i)$, and applies in that case f_i to x_i .

¹ which, in categorical notation, is written as cotuple $[f_1, \dots, f_n]$.

3 Java Semantics for Verification

This section explains the essentials of the semantics of (sequential) Java as used in the LOOP project. As such it exists in the form of PVS and Isabelle/HOL definitions in higher order logic, in so-called prelude files, which form the basis for every verification exercise. Here we shall use a more mathematical notation for the basic ingredients of this semantics. Later in this paper it will be reformulated (and simplified) using a monad.

Traditionally, in denotational semantics an imperative program s is interpreted as a partial function on some state space S , i.e.

$$S \xrightarrow{\llbracket s \rrbracket} 1 + S$$

The state space S is a global store giving the values of all program variables. We will not go in to the precise form of the store here; for more detail, see [2]. Above we have used the notation introduced in the previous section; the conventional notation for $1 + S$ is $S_{\perp}$. The $1 + \dots$ option in the result type signals nontermination (or “hanging”).

Similar to program statements, an expression e – possibly having side effects – is interpreted as a function

$$S \xrightarrow{\llbracket e \rrbracket} 1 + (S \times B)$$

Again, the first $+$ -option 1 in the result type signals non-termination. The second option is for normal termination, which for expressions (of type B) yields a state, needed for side-effects, and a result value in B .

In a real programming language like Java however, things are more complicated. Statements and expressions in Java can not just hang or terminate normally, they can also terminate abruptly. Expressions can only terminate abruptly because of an exception (*e.g.* through division by 0), but statements may also terminate abruptly because of a **return** (to exit from a method call), **break** (to exit from a block, repetition or switch-statement), or **continue** (to skip the remainder of a repetition). The last two options can occur with or without label. Consequently, the result types of statements and expressions will have to be more complicated than the $1 + S$ and $1 + (S \times B)$ above. The result types of statements and expressions are $\text{StatResult}(S)$ and $\text{ExprResult}(S, B)$, where:

$$\begin{aligned} \text{StatResult}(S) &= 1 + S + \text{StatAbn}(S) \\ \text{ExprResult}(S, B) &= 1 + (S \times B) + \text{ExprAbn}(S) \end{aligned}$$

Here $\text{StatAbn}(S)$ and $\text{ExprAbn}(S)$ are the types of statement and expression abnormalities, defined below.

Later in the monadic description we shall abstract away from the particular shapes of these abnormalities, but now we want to show what really happens in the semantics of Java (that is used for verification). Therefore, we describe all

these abnormality options in appropriate definitions, involving a state space S . First, abnormal termination for statements is captured via four options:

$$\text{StatAbn}(S) = (S \times \text{RefType}) + S + (S \times (1 + \text{String})) + (S \times (1 + \text{String}))$$

where RefType and String are constant sets used for references and strings. The first $+$ -option $S \times \text{RefType}$ describes an exception result, consisting of a state and a reference to an exception. The second $+$ -option is for a return result, the third one for a break result (possibly with a string as label), and the fourth one for a continue result (also possibly with a string as label).

Since exceptions are the only abnormalities that can result from expressions we have:

$$\text{ExprAbn}(S) = S \times \text{RefType}.$$

A void method `void m(A1 a1, ..., An an){ ... }` in Java is then translated as a state transformer function $S \times A1 \times \dots \times An \rightarrow \text{StatResult}(S)$. A non-void method `B m(A1 a1, ..., An an){ ... }` gets translated as a function $S \times A1 \times \dots \times An \rightarrow \text{ExprResult}(S, B)$. Notice that these state transformers can be described as coalgebras, namely of the functors $S \mapsto \text{StatResult}(S)^{A1 \times \dots \times An}$ and $S \mapsto \text{ExprResult}(S, B)^{A1 \times \dots \times An}$.

On the basis of this representation of statements and expressions all language constructs of (sequential) Java are translated into PVS and Isabelle [11,7,2,15]. For instance, the composition $s; t: S \rightarrow \text{StatResult}(S)$ of two statements $s, t: S \rightarrow \text{StatResult}(S)$ is defined as:

$$\begin{aligned} s; t = \lambda x \in S. \text{CASES } s(x) \text{ OF } \{ \\ \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x') \mapsto t(x'), \\ \kappa_3(w) \mapsto \kappa_3(w) \} \end{aligned} \quad (1)$$

This means that if $s(x)$ hangs or terminates abruptly, then $(s; t)(x) = s(x)$ so that t is not executed at all, and if $s(x)$ terminates normally resulting in a successor state x' , then $(s; t)(x) = t(x')$ and t is executed on this successor state. Notice how abnormalities are propagated.

The Java evaluation strategy prescribes that arguments should be evaluated first, from left to right (see [5, §§ 15.6.4]). But so far we have used values as arguments, and not expressions possibly having side-effects. Restricting to the case with one argument, this means that for a statement $t: S \times A \rightarrow \text{StatResult}(S)$ we still have to define an extension² t^* of t with type $t^*: (S \times (S \rightarrow \text{ExprResult}(S, A))) \rightarrow \text{StatResult}(S)$, namely as:

$$\begin{aligned} t^*(x, e) = \text{CASES } e(x) \text{ OF } \{ \\ \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x', a) \mapsto t(x', a), \\ \kappa_3(w) \mapsto \kappa_3(w) \} \end{aligned} \quad (2)$$

(and similarly for an expression instead of a statement t).

² In PVS and Isabelle we use overloading and also write t for t^* .

In the next section we shall see how composition and extension can be obtained from an underlying monad.

4 The Monad for Java Semantics and Its Properties

This section introduces an appropriate monad J for Java statements and expressions. Its (categorical) properties are investigated in some detail, with emphasis on extension and composition, and on the order on homsets of the Kleisli category $\mathbf{Kl}(J)$ of the monad J .

The first step is to simplify the situation from the previous section. This is done by ignoring the complicated structure of Java abnormalities, and using one fixed set E in place of both $\mathbf{StatAbn}$ and $\mathbf{ExprAbn}$. Then we can see a statement as a special form of expression, namely with result type 1. Thus, our general state transformer functions are of the form:

$$S \times A \longrightarrow 1 + (S \times B) + (S \times E)$$

Within the LOOP semantics they are regarded as coalgebras:

$$S \longrightarrow \left(1 + (S \times B) + (S \times E)\right)^A$$

But here we shall look at them as morphisms

$$A \longrightarrow \left(1 + (S \times B) + (S \times E)\right)^S$$

in the Kleisli category of a monad. Via currying these two representations are of course equivalent, but they give different perspectives. In the coalgebraic view the state space S plays a central role, in the monadic view state is just one of the ingredients of the monad, like partiality and exceptions.

Definition 1. Let $\mathbf{Sets}$ be the category of sets and functions. Fix two sets E for exceptions and S for states. A functor $J: \mathbf{Sets} \rightarrow \mathbf{Sets}$ is defined by

$$J(A) = \left(1 + (S \times A) + (S \times E)\right)^S \quad (3)$$

It forms a monad with unit and multiplication natural transformations:

$$A \xrightarrow{\eta_A} J(A) \qquad J^2(A) \xrightarrow{\mu_A} J(A)$$

given by

$$\eta_A(a) = \lambda x \in S. \kappa_2(x, a) \qquad \mu_A(f) = \lambda x \in S. \text{ CASES } f(x) \text{ OF } \left\{ \begin{array}{l} \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x', g) \mapsto g(x'), \\ \kappa_3(x', e) \mapsto \kappa_3(x', e) \end{array} \right\}.$$

It is not hard to check that the three monad equations $\mu_A \circ \eta_{J(A)} = \text{id}$, $\mu_A \circ J(\eta_A) = \text{id}$ and $\mu_A \circ J(\mu_A) = \mu_A \circ \mu_{J(A)}$ are satisfied. Notice that the monad J incorporates ingredients from three basic computational monads introduced in [12]: the partiality monad $A \mapsto 1 + A$, the exception monad $A \mapsto A + E$ and the side-effect monad $A \mapsto (S \times A)^S$. But J is not obtained via composition from these basic monads, so the modular approach to computational monads from *e.g.* [3] is not relevant here.

4.1 Extension and Composition

It is folklore knowledge that every functor $F: \mathbf{Sets} \rightarrow \mathbf{Sets}$ is strong, with strength natural transformation $\text{st}_{A,B}: A \times F(B) \rightarrow F(A \times B)$ given by $(a, z) \mapsto F(\lambda b \in B. (a, b))(z)$. This strength definition applies in particular to the above functor J . Explicitly,

$$\begin{aligned} \text{st}_{A,B}(a, f) = \lambda x \in S. \text{ CASES } f(x) \text{ OF } \{ \\ \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x', b) \mapsto \kappa_2(x', (a, b)), \\ \kappa_3(x', e) \mapsto \kappa_3(x', e) \}. \end{aligned} \quad (4)$$

In order to show that J is a *strong monad*, and not just a strong functor, we have to check that additionally the following diagrams commute.

$$\begin{array}{ccc} A \times B & \xrightarrow{\text{id} \times \eta_B} & A \times J(B) \\ & \searrow \eta_{A \times B} & \downarrow \text{st}_{A,B} \\ & & J(A \times B) \end{array} \quad \begin{array}{ccc} A \times J^2(B) & \xrightarrow{\text{id} \times \mu_B} & A \times J(B) \\ \text{st}_{A, J(B)} \downarrow & & \downarrow \text{st}_{A,B} \\ J(A \times J(B)) & & \\ J(\text{st}_{A,B}) \downarrow & & \\ J^2(A \times B) & \xrightarrow{\mu_{A \times B}} & J(A \times B) \end{array}$$

This is an easy exercise.

Using this strength map there is a standard way to turn functions $f: A \times B \rightarrow J(C)$ into functions $f^*: A \times J(B) \rightarrow J(C)$, namely as:

$$f^* = \mu_C \circ J(f) \circ \text{st}_{A,B} \quad (5)$$

Explicitly, this “Kleisli extension” can be described on $a \in A$ and $g \in J(B)$ as:

$$\begin{aligned} f^*(a, g) = \lambda x \in S. \text{ CASES } g(x) \text{ OF } \{ \\ \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x', b) \mapsto f(a, b)(x'), \\ \kappa_3(x', e) \mapsto \kappa_3(x', e) \} \end{aligned}$$

Thus, this Kleisli extension is the same as extension for Java described in (2).

Example 1. Recall that Java (like C) has two conjunctions, namely “and” $\&$ and “conditional and” $\&\&$, see [5, §§ 15.22]. The first one ($\&$) always evaluates both arguments, but the second one ($\&\&$) only does so if the first argument evaluates to true. The difference is relevant in the presence of side-effects, non-termination, or exceptions. We show how both these operations can be obtained via Kleisli extension, starting from the standard conjunction operations $\wedge: \text{bool} \times \text{bool} \rightarrow \text{bool}$.

First, the one-step extension $(\eta_{\text{bool}} \circ \wedge)^*: \text{bool} \times J(\text{bool}) \rightarrow J(\text{bool})$ only has to evaluate its second argument. Swapping the arguments appropriately—via the function swap with $\text{swap}(x, y) = (y, x)$ —and extending again yields:

$$J(\text{bool}) \times J(\text{bool}) \xrightarrow[\left((\eta_{\text{bool}} \circ \wedge)^* \circ \text{swap}\right)^* \circ \text{swap}]{\& \stackrel{\text{def}}{=}} J(\text{bool})$$

Explicitly,

$$\begin{aligned} f_1 \& f_2 = \lambda x \in S. \text{ CASES } f_1(x) \text{ OF } \{ \\ & \kappa_1(u_1) \mapsto \kappa_1(u_1), \\ & \kappa_2(x_1, b_1) \mapsto \text{ CASES } f_2(x_1) \text{ OF } \{ \\ & \quad \kappa_1(u_2) \mapsto \kappa_1(u_2), \\ & \quad \kappa_2(x_2, b_2) \mapsto \kappa_2(x_2, b_1 \wedge b_2), \\ & \quad \kappa_3(x_2, e_2) \mapsto \kappa_3(x_2, e_2) \}, \\ & \kappa_3(x_1, e_1) \mapsto \kappa_3(x_1, e_1) \} \end{aligned}$$

The conditional and can be obtained by once extending the auxiliary function $t: J(\text{bool}) \times \text{bool} \rightarrow J(\text{bool})$ given by $t(f, b) = \text{IF } b \text{ THEN } f \text{ ELSE } \eta_{\text{bool}}(\text{false})$.

$$J(\text{bool}) \times J(\text{bool}) \xrightarrow[t^* \circ \text{swap}]{\&\& \stackrel{\text{def}}{=}} J(\text{bool})$$

This concludes the example.

We turn to the Kleisli category $\mathbf{Kl}(J)$ of the monad J . Its objects are sets, and its morphisms $A \rightarrow B$ are functions $A \rightarrow J(B)$. The identity map $A \rightarrow J(A)$ in $\mathbf{Kl}(J)$ is the unit η_A at A , and the “Kleisli” composition $g \bullet f: A \rightarrow J(C)$ of two morphisms $f: A \rightarrow J(B)$ and $g: B \rightarrow J(C)$ in $\mathbf{Kl}(J)$ is standardly defined as:

$$g \bullet f = \mu_C \circ J(g) \circ f. \quad (6)$$

Unraveling yields for $a \in A$,

$$\begin{aligned} (g \bullet f)(a) = \lambda x \in S. \text{ CASES } f(x) \text{ OF } \{ \\ & \kappa_1(u) \mapsto \kappa_1(u), \\ & \kappa_2(x', b) \mapsto g(b)(x'), \\ & \kappa_3(x', e) \mapsto \kappa_3(x', e) \} \end{aligned}$$

Thus, Kleisli composition $\bullet$ is basically the same as Java composition ; from (1): if f does not terminate or terminates abruptly, so does $g \bullet f$, and if f terminates normally and produces a successor state, then g is executed on this state.

For future use we define how to iterate a function $s: A \rightarrow J(A)$ using Kleisli composition:

$$s^n = \begin{cases} \eta_A & \text{if } n = 0 \\ s \bullet s^{n-1} & \text{otherwise.} \end{cases} \quad (7)$$

4.2 Cpo Structure on Kleisli Homsets

The homsets of the Kleisli category of the monad J are the sets of morphisms $f: A \rightarrow B$ in the Kleisli category $\mathbf{Kl}(J)$, *i.e.* the sets of functions $f: A \rightarrow J(B)$. These can be ordered via the pointwise flat order: for $f_1, f_2: A \rightarrow J(B)$, $f_1 \sqsubseteq f_2$ if for each $a \in A$ and $x \in S$, $f_1(a)(x)$ hangs, or else is equal to $f_2(a)(x)$. More formally:

$$f_1 \sqsubseteq f_2 \iff (\forall a \in A. \forall x \in S. f_1(a)(x) = \kappa_1(*) \vee f_1(a)(x) = f_2(a)(x)) \quad (8)$$

It is not hard to see that $\sqsubseteq$ is a partial order. Also, there is a least element $\perp = \lambda a \in A. \lambda x \in S. \kappa_1(*)$. Notice that $f \bullet \perp = \perp$, but $\perp \bullet f$ may be different from $\perp$, namely when f throws an exception.

It is standard that the flat order is a complete partial order (cpo): an order in which each ascending sequence has a least upperbound. Hence the pointwise flat order $\sqsubseteq$ also makes the set of morphisms $A \rightarrow J(B)$ in the Kleisli category $\mathbf{Kl}(J)$ a cpo. Explicitly, for an ascending chain $(f_n: A \rightarrow J(B))_{n \in \mathbb{N}}$ there is a least upperbound $f = \bigsqcup_{n \in \mathbb{N}} f_n: A \rightarrow J(B)$ given by:

$$f(a)(x) = \begin{cases} \kappa_1(*) & \text{if } \forall n \in \mathbb{N}. f_n(a)(x) = \kappa_1(*) \\ f_\ell(a)(x) & \text{else, where } \ell \text{ is the least } n \text{ with } f_n(a)(x) \neq \kappa_1(*) \end{cases} \quad (9)$$

Kleisli composition $\bullet$ is continuous (*i.e.* preserves the order and least upperbounds) in both its arguments. This means that the Kleisli category $\mathbf{Kl}(J)$ is cpo-enriched, see [4].

We summarise what we have found so far.

Proposition 1. *The functor J from (3) describing the outputs of Java statements and expressions is a strong monad on the category of sets. Its Kleisli composition and extension correspond to composition and extension in Java. And its Kleisli category $\mathbf{Kl}(J)$ is cpo-enriched.* $\square$

The following result about extension and continuity is useful later in Subsection 5.2.

Lemma 1. *Consider a function $f: A \times B \rightarrow J(C)$ and its extension $f^*: A \times J(B) \rightarrow J(C)$ from (5).*

1. *For each $a \in A$, the function $f^*(a, -): J(B) \rightarrow J(C)$ is continuous.*
2. *If the set A carries an order in such a way that for each $b \in B$, the function $f(-, b): A \rightarrow J(C)$ is continuous, then for each $g \in J(B)$, $f^*(-, g): A \rightarrow J(C)$ is also continuous.* $\square$

5 While Statements and Recursive Statements

This section starts with the definition of the while construct that is used for the semantics of Java. Actually, here we present a simplified version in which exceptions cause a break out of the while statement. In Java one may have a `continue` statement inside a while loop, causing a skip of only the remainder of the current cycle. The full version that is used for the verification of Java programs is described in [7]. The second part of this section is devoted to handling of recursive statements, via least fixed points.

5.1 While Statement

For a Boolean computation $c \in J(\text{bool})$ we define a function $\text{while}(c)$ taking a statement $s: A \rightarrow J(A)$ in the Kleisli category $\mathbf{Kl}(J)$ to a new statement $\text{while}(c)(s): A \rightarrow J(A)$. The idea is of course to iterate s until c becomes false. But there are various subtleties involved:

1. The condition c may itself have a side-effect, which has to be taken into account. Therefore we iterate $s \bullet \widehat{c}$, where $\widehat{c}: A \rightarrow J(A)$ is the statement which executes c only for its side-effect and ignores its result:

$$\begin{aligned} \widehat{c}(a)(x) = \text{CASES } c(x) \text{ OF } \{ \\ \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x', b) \mapsto \kappa_2(x', a), \\ \kappa_3(x', e) \mapsto \kappa_3(x', e) \} \end{aligned}$$

(or, equivalently, $\widehat{c}(a) = J(\lambda b \in \text{bool}. a)(c)$).

2. During the iteration both c and s may throw exceptions. If this happens the while statement must throw the same exception.

In order to detect that the condition becomes false or an exception is thrown we define two partial functions $N(c, s), E(c, s) : A \rightarrow S \rightarrow \mathbb{N}$. The number $N(c, s)(a)(x)$, if defined, is the smallest number of iterations after which c becomes false without occurrence of exceptions. Similarly, $E(c, s)(a)(x)$, if defined, is the smallest number of iterations after which an exception is thrown. More formally, $N(c, s)(a)(x)$ is the smallest number n such that

$$(\widehat{c} \bullet (s \bullet \widehat{c})^n)(a)(x) = \kappa_2(x', \text{false})$$

for some x' , if such a number n exists, and $E(c, s)(a)(x)$ is the smallest number n such that

$$(s \bullet \widehat{c})^n(a)(x) = \kappa_3(x', e)$$

for some x' and e , if such a number n exists.

By the definition of Kleisli composition, if $f(a)(x) = \kappa_3(x', e)$ for some $f : A \rightarrow J(B)$ then $(g \bullet f)(a)(x) = f(a)(x)$ for all $g : B \rightarrow J(C)$. In other words, if $f(a)(x)$ throws an exception, then $(g \bullet f)(a)(x)$ also throws that exception. So, if $(s \bullet \widehat{c})^n(a)(x)$ throws an exception then $\widehat{c} \bullet (s \bullet \widehat{c})^m(a)(x)$ throws the same

exception for all $m \geq n$. This means that if both $N(c, s)(a)(x)$ and $E(c, s)(a)(x)$ are defined, then $N(c, s)(a)(x)$ is smaller than $E(c, s)(a)(x)$. This is used in the following definition of the while statement.

Definition 2. For a condition $c \in J(\text{bool})$ and a statement $s: A \rightarrow J(A)$ we define the statement $\text{while}(c)(s): A \rightarrow J(A)$ as:

$$\begin{aligned} \text{while}(c)(s) &= \lambda a \in A. \lambda x \in S. \\ &\begin{cases} (\widehat{c} \bullet (s \bullet \widehat{c})^n)(a)(x) & \text{if } N(c, s)(a)(x) = n \\ (s \bullet \widehat{c})^n(a)(x) & \text{if } E(c, s)(a)(x) = n \text{ and } N(c, s)(a)(x) \text{ is undefined} \\ \kappa_1(*) & \text{if both } N(c, s)(a)(x) \text{ and } E(c, s)(a)(x) \text{ are undefined} \end{cases} \end{aligned}$$

In the standard elementary semantics of programming languages using partial functions the while statement is characterised as least fixed point *e.g.* [17, Definition 9.18]. The next result shows that this also holds in the present setting (with side-effects in expressions and with exceptions).

Proposition 2. The above while statement $\text{while}(c)(s)$ is the least fixed point of the operator $F(c, s): (A \rightarrow J(A)) \rightarrow (A \rightarrow J(A))$ given by:

$$F(c, s) = \lambda t \in A \rightarrow J(A). \text{IfThenElse}(c, t \bullet s, \eta_A)$$

where the conditional statement is defined as:

$$\begin{aligned} \text{IfThenElse}(c, t_1, t_2) &= \lambda a \in A. \lambda x \in S. \text{CASES } c(x) \text{ OF } \{ \\ &\quad \kappa_1(u) \mapsto \kappa_1(u), \\ &\quad \kappa_2(x', b) \mapsto \text{IF } b \\ &\quad \quad \text{THEN } t_1(a)(x') \\ &\quad \quad \text{ELSE } t_2(a)(x'), \\ &\quad \kappa_3(x', e) \mapsto \kappa_3(x', e) \} \end{aligned}$$

This conditional statement may also be defined via extension, applied to the obvious conditional function $(A \rightarrow J(A))^2 \times \text{bool} \rightarrow (A \rightarrow J(A))$.

Proof. The proof that $\text{while}(c)(s)$ is a fixed point of $F(c, s)$ proceeds by distinguishing many cases and handling them one by one.

First we consider the following three cases: (1) the condition hangs; (2) the condition throws an exception; (3) the condition terminates normally and evaluates to **false**. In all these cases the statement is not executed at all, and the outcome of the while is easily established using the above functions N and E : it hangs in case of (1), it throws the same exception as the condition in (2) and it terminates normally in (3).

The statement does get executed in case: (4) the condition terminates normally and evaluates to **true**. This leads to the subcases: (4a) the statement hangs; (4b) the statement throws an exception; (4c) the statement terminates normally. In the last case we use the following auxiliary result. If $c(x) = \kappa_2(x', \text{true})$ and $s(a)(x') = \kappa_2(x'', a')$, then $\text{while}(c)(s)(a)(x) = \text{while}(c)(s)(a')(x'')$.

In order to show that $\text{while}(c)(s)$ is the least fixed point of $F(c, s)$, we assume a function $t: A \rightarrow J(A)$ with $F(c, s)(t) = t$. We then first show by induction on n that:

1. If $N(c, s)(a)(x) = n$, then $t(a)(x) = (\widehat{c} \bullet (s \bullet \widehat{c})^n)(a)(x)$.
2. If $E(c, s)(a)(x) = n$, then $t(a)(x) = (s \bullet \widehat{c})^n(a)(x)$.

The result $\text{while}(c)(s) \sqsubseteq t$ then follows by unpacking the definition of **while**. $\square$

Definition 2 and Proposition 2 give quite different characterisations of the meaning of **while**. Both correspond to an intuitive understanding of the semantics of **while**. Which of these characterisations is the more fundamental one – and should therefore be considered as the definition – is a matter of taste, but we should prove their equivalence.

5.2 Recursive Statements

A recursive method with input type A and result type B is interpreted as a function of type $A \rightarrow J(B)$, which is constructed from a mapping from statements to statements of type $(A \rightarrow J(B)) \rightarrow (A \rightarrow J(B))$. The semantics of such a recursive statement can be defined in the same way as the semantics of the **while** statement. For a mapping $F: (A \rightarrow J(B)) \rightarrow (A \rightarrow J(B))$ we define a partial function $U(F): A \rightarrow S \rightarrow \mathbb{N}$. The number $U(F)(a)(x)$, if defined, is the smallest n such that

$$F^n(\perp)(a)(x) \neq \kappa_1(*).$$

Definition 3. For a continuous mapping $F: (A \rightarrow J(B)) \rightarrow (A \rightarrow J(B))$ we define the (recursive) statement $\text{rec}(F): A \rightarrow J(B)$ as:

$$\text{rec}(F) = \lambda a \in A. \lambda x \in S. \begin{cases} F^n(\perp)(a)(x) & \text{if } U(F)(a)(x) = n \\ \kappa_1(*) & \text{if } U(F)(a)(x) \text{ is undefined} \end{cases}$$

Of course, like the semantics of **while** earlier, **rec** is a least fixed point:

Proposition 3. The above statement $\text{rec}(F)$ is the least fixed point of F for continuous F . $\square$

Continuity of a particular F is typically easy to show; by Lemma 1 all extensions – like $\&$ and $\&\&$ in Example 1 – are continuous, and it is a standard result that application, lambda abstraction, composition, and the taking of least fixed points all preserve continuity.

6 Meta-theory in PVS

The semantical basis for verifications of Java programs as sketched in Section 3 has (of course) been formalised in PVS and Isabelle/HOL, to allow actual verification of Java programs using these theorem provers. Sections 4 and 5 describe

a mathematical abstraction of this semantical basis in terms of a computational monad J . Also this abstract meta-theory has also been formalised in PVS. This formalisation is not the one that is used for actual verification of Java programs, but is used only here for avoiding errors. Indeed, the definitions and proofs in this paper involve many case distinctions, and it is easy to make mistakes there, *e.g.* by accidentally skipping an option.

For those readers familiar with PVS, we show three crucial definitions. First of all, the monad (3) is defined, in a PVS theory with parameter type A as:

```
J : TYPE = [S -> lift[union[[S, A], [S, E]]]]
```

It involves the predefined `lift` type constructor, corresponding to $1 + (-)$, and the coproduct constructor `union` for $+$. The Cartesian product in PVS is described via square brackets `[_,_]`, and the function space via `[->_]`. The multiplication natural transformation μ is then defined as:

```
mult : [J[J[A]] -> J[A]] =
  LAMBDA(z : J[J[A]]) :
    LAMBDA(x : S) :
      CASES z(x) OF
        bottom : bottom,
        up(w) : CASES w OF
          inl(u) : proj_2(u)(proj_1(u)),
          inr(v) : up(inr(v))
        ENDCASES
      ENDCASES
```

In such a way all the definitions, results and proofs in this paper have been formalised in PVS.

One of the trickiest parts in the formalisation in PVS concerns the $N(c, s)$ and $E(c, s)$ used in the definition of the while. We shall describe $N(c, s)(a)(x)$ as a set of numbers, see also [7]. It is either empty or a singleton.

```
N(c, s)(a)(x) : PRED[nat] = LAMBDA(n : nat) :
  CASES iterate(s o E2S(c), n)(a)(x) OF
    bottom : FALSE,
    up(w) : CASES w OF
      inl(u) : CASES c(proj_1(u)) OF
        bottom : FALSE,
        up(z) : CASES z OF
          inl(s) : NOT proj_2(s),
          inr(s) : FALSE
        ENDCASES
      ENDCASES,
    inr(v) : FALSE
  ENDCASES
ENDCASES
AND
```

```

FORALL(m : nat) : m < n IMPLIES
  CASES iterate(s o E2S(c), m)(a)(x) OF
    bottom : FALSE,
    up(w) : CASES w OF
      inl(u) : CASES c(proj_1(u)) OF
        bottom : FALSE,
        up(z) : CASES z OF
          inl(s) : proj_2(s),
          inr(t) : FALSE
        ENDCASES
      ENDCASES,
    inr(v) : FALSE
  ENDCASES
ENDCASES

```

The notation $E2S(c)$ is used for what we have written as $\hat{c}$ in the beginning of Subsection 5.1.

7 Conclusions

We have discussed the monad that is at the heart of the semantics of sequential Java used in the LOOP project as the basis of mechanically assisted verification of actual Java programs. This monad is a useful notion in organizing the Java semantics, particularly when it comes to the subtleties involved with handling abnormal termination.

It is interesting to compare our approach to the one taken in the denotational semantics of sequential Java described in [1], which uses continuations to deal with exceptions and breaks. In our approach the monad J makes explicit everything involved with the control flow of programs. In the approach of [1] some of this complexity is implicit in the shape of the global state S , which contains global variables that keep track of continuation information.

A disadvantage of the continuation approach is that in the definition of the semantics one has to be very careful to update this information in all the right places; forgetting to do this at some point will mean that a wrong continuation will be taken. Given the size and complexity of the semantics avoiding such mistakes is not trivial³. In our monadic approach the type information of the very basic type system discussed in Section 2 provides some safety: for the definitions to be well-typed we are essentially forced to consider all options, and simply forgetting a case would result in an ill-typed rather than an incorrect definition. This advantage is in particular relevant for the LOOP compiler, as it provides denotations as PVS or Isabelle code, which can indeed be mechanically typechecked.

³ For example, the definition of the semantics of the while statement in [1] does not appear to update the “continue” information when a repetition is entered.

The monadic approach is less ad-hoc, in that the same basic machinery is used to deal with nontermination and abrupt termination. The monadic approach is also more general. For example, extending the semantics to cope with threads could possibly be achieved by considering a more complicated monad than J , but such a semantics of concurrent Java is still left as future work.

One crucial ingredient of Java is throwing and catching of exceptions. Appropriate functions can be defined in the current setting: for $e \in E$, take

$$1 \xrightarrow{\text{throw}(e)} J(A)$$

as

$$\text{throw}(e)(*) = \lambda x \in S. \kappa_3(x, e)$$

and for $C \subseteq E$,

$$J(A) \times J(A)^E \xrightarrow{\text{try\_catch}(C)} J(A)$$

as

$$\begin{aligned} \text{try\_catch}(C)(s, t) = \lambda x \in S. \text{CASES } s(x) \text{ OF } \{ \\ \kappa_1(u) \mapsto \kappa_1(u), \\ \kappa_2(x', a) \mapsto \kappa_2(x', a), \\ \kappa_3(x', e) \mapsto \text{IF } e \in C \\ \text{THEN } t(e)(x') \\ \text{ELSE } \kappa_3(x', e) \} \end{aligned}$$

What is slightly unsatisfactory about these definitions is that they are *ad hoc* from a mathematical perspective: it is not clear in what sense they are canonical (like composition, extension or recursion) and what basic laws they (should) satisfy. This remains to be investigated.

References

1. J. Alves-Foss and F. S. Lam. Dynamic Denotational Semantics of Java. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes Comp. Sci.*, pages 201–240. Springer-Verlag, 1998.
2. J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.
3. P. Cenciarelli. An algebraic view of program composition. In A.M. Haeberer, editor, *Algebraic Methodology and Software Technology*, number 1548 in *Lect. Notes Comp. Sci.*, pages 325–340. Springer, Berlin, 1998.
4. M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Cambridge Univ. Press, 1996.
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
6. M. Huisman and B. Jacobs. Inheritance in Higher Order Logic: Modeling and Reasoning. Techn. Rep. CSI-R0004, Comput. Sci. Inst., Univ. of Nijmegen, 2000.

7. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci. Springer, Berlin, 2000.
8. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop*, pages 37–44. Techn. Rep. 251, Fernuniversität Hagen, 1999.
9. B. Jacobs. Inheritance and cofree constructions. In P. Cointe, editor, *European Conference on Object-Oriented Programming*, number 1098 in Lect. Notes Comp. Sci., pages 210–231. Springer, Berlin, 1996.
10. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
11. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.
12. E. Moggi. Notions of computation and monads. *Inf. & Comp.*, 93(1):55–92, 1991.
13. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
14. L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and computer science*, pages 361–386. Academic Press, London, 1990. The APIC series, vol. 31.
15. Loop Project. <http://www.cs.kun.nl/~bart/LOOP/>.
16. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
17. J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, MA, 1977.

A Global Semantics for Views

Christine Choppy¹, Pascal Poizat², and Jean-Claude Royer²

¹ LIPN, Institut Galilée - Université Paris XIII,
Avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

² IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, F-44322 Nantes cedex 3, France
(Pascal.Poizat,Jean-Claude.Royer@irin.univ-nantes.fr)
<http://www.sciences.univ-nantes.fr/info/perso/permanents/poizat/>

Abstract. We focus on the specification of mixed systems that contain static and dynamic aspects. Our approach aims at keeping advantage of the languages dedicated to both aspects (algebraic specifications for data types, and state transition diagrams for dynamic behaviour) while providing an underlying unifying framework accompanied by an appropriate semantic model. This underlying framework is based on our notion of *views*. In [4] we addressed the composition of any number of views in a composition. Here we address the strong links that exist between all aspects of a single component, and that altogether build its global semantics. After presenting the (state and transition) formulas and their semantics, we show how to glue the different aspects together, and we present the retrieval of the global view for components. We provide here a new set of rules for the STS part of our views, taking into account the composition of all aspects. We then give the formal (global) semantics for such global views. We illustrate our view model on a password management example.

Keywords: mixed specifications, view formalism, global semantics.

1 Introduction

We focus on the specification of mixed systems that contain static and dynamic aspects and feature a certain level of complexity that requires the definition of structuring mechanisms. We use two standard ways of achieving structuring/modularity that are, on the one hand, the extension (a simple form of inheritance), and on the other hand, the union (or composition). These two means come from the algebraic specification formalisms [18] and have been adapted to the object oriented analysis and design context.

In the last few years, the need for a separation of concerns with reference to static and dynamic aspects appeared. This issue was addressed in (non exhaustive list) LOTOS [7] (Abstract Data Types and CSP/CCS), SDL [5] (Abstract Data Types and State/Transition Diagrams), and also more recently in approaches combining Z and process algebras (CSP-OZ [16] for example). This is also reflected in object oriented analysis and design approaches such as OMT

or UML [17] where static and dynamic aspects are dealt with by different diagrams (class diagrams, interaction diagrams, statecharts). However, in both approaches, the (formal) links and consistency between the two aspects are not defined, which limits the possibilities of reasoning on the whole component.

Some approaches encompass both aspects within a single framework, like for example LTS [14], rewriting logic [10], TLA [8]. Our approach aims at keeping advantage of the languages dedicated to both aspects (*e.g.* algebraic specifications for data types, and state transition diagrams for dynamic behaviour) while providing an underlying unifying framework accompanied by an appropriate semantic model.

Our specification formalism (called KORRIGAN) is integrated into a framework for system specification (Fig. 1). In [12], we define a methodology for our specifications that makes use of conditions in order to build abstractions of systems and that is applied to generate LOTOS and SDL specifications and verify them. In [13], we explain how concurrent object oriented code may be generated from our specifications.

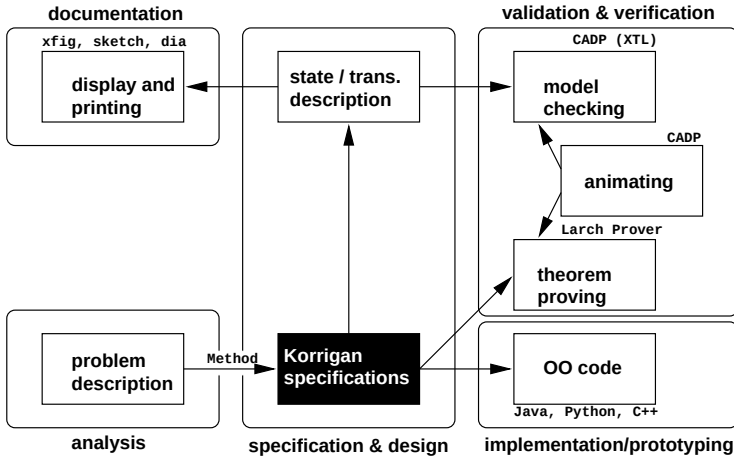


Fig. 1. The specification framework

We present first each of the view classes of our model and the corresponding specification language. Whereas in [4] we addressed the composition of any number of views, we address here the strong links that exist between all aspects of a single component, and that altogether build its global semantics. Therefore, after presenting the formulas (and their semantics) we show how to glue the different aspects together, and we present the retrieval of the global view for components. We provide here a new set of rules for the STS part of our views (see below Definition 4), taking into account the composition of all aspects. We then give the formal (global) semantics for such global views. We illustrate our view model on a password management example. The application is here sketched. The full version is to be found in [3].

2 The Password Management Example

The system is composed of several basic users, a privileged user (root) and the password manager. The basic users may change their password. The privileged user may act as a basic user (changing its password) but may also create passwords for new users and change passwords for actual users.

The creation of a password is done as follows: (i) the user id is prompted, then either (iia) it already exists and an error is signaled, or (iib) the password is prompted for. Thereafter (if no error occurred), the password is prompted for once again and either (iiia) it is different (and an error is signaled), or (iiib) it is the same and the password file is modified.

The change of a password is the same but for the difference that (iv) the user has to exist and that the (old) password is asked for (and verified) before the new password part (iib, iiia or iiib) takes place. More complex systems, with password restrictions, may exist but we do not take them into account here.

3 The View Model

A Specification Framework. We will here present the different views that compose our view model. As we deal with identified communicating components, we first present our communication choices and our notion of component identifier.

Communication Choices. Dealing with communicating components, one has first to make choices with respect to the communication arity and the various ways to express communication [3]. Since we are concerned both with object-orientation and readability, we will restrict ourselves to the 1→n communication type. Multiway communication may be more expressive but less easily understandable. Moreover, we will allow both explicit (with our process identifiers being some kind of URL) and implicit communication (through the use of the **sender** keyword, denoting the sender of the last message being received by the component).

Identifiers. A step to object orientation are (object) identifiers. Moreover, our communication mechanisms need component identifiers. Identifiers are simple terms of **PId** sorts. Each different component class has its associated **PId** sort. **PId** subsorts correspond to related (through inheritance) component classes.

We have two important relations on **PId**: *object equality* and *object structure*. Object equality expresses that two objects with the same identifier denote actually the same object: $\forall o, o' \in Obj, Id(o) = Id(o') \Leftrightarrow o = o'$.

Object structure is expressed by the fact that, whenever an object O is composed of several sub-objects, then the object identifier is a prefix for all sub-objects identifiers: $\forall o \in Subobjects(O), \exists i : PId . Id(o) = Id(O).i$.

This way, identifiers build a tree-ordered id-indexed structure over object composition. *Id-indexing* is widely used when dealing with components built over other (sub-)components.

3.1 The Different View Classes

We define an *Internal Structuring View* abstraction that expresses the fact that, in order to design an object, it is useful to be able to express it under its different aspects (here the static and dynamic aspects, with no exclusion of further aspects that may be identified later on).

We therefore have here a first structuring level (*internal structuring*) for objects. Another structuring level is the *external structuring*, expressed by the fact that an object may be composed of several other objects. An object may then be either a simple one (integrating different internal structuring views in an *integration view*), or a composite object (*composition view*).

Integration views follow an *encapsulation principle*: the static part (data type) may only be accessed through the dynamic part (behaviour) and its identifier (Id). The whole class diagram for the view model is given in Figure 2.

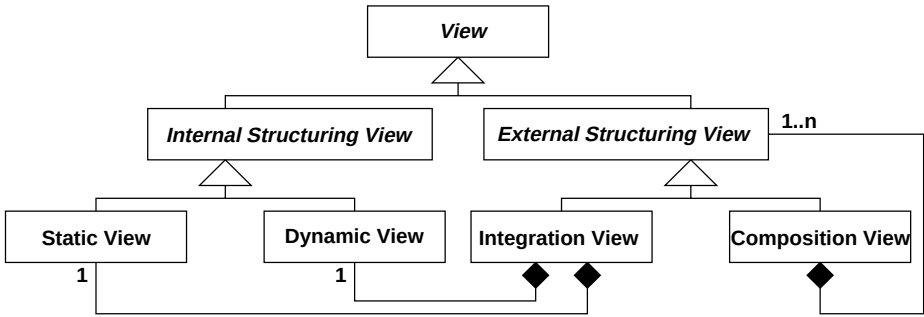


Fig. 2. The view model class diagram (using the UML notation)

Views. In a unifying approach, views are used to describe the different aspects of a component (internal and external structuring).

Definition 1 (View). A view V_T of a component T is a triple (A_T, α, D_T) where A_T is the data part, α is an abstraction of T and D_T a dynamic layer over A_T .

The data part addresses the functional issues for the component (definition of the operation values, in an algebraic way). The abstraction defines a point of view for the component, and the dynamic layer addresses the possible sequences of operations the component may perform.

Definition 2 (View data part). A data part A_T (in a view structure (A_T, α, D_T)) is an algebraic specification with the following components $(A', G, V, \Sigma_T, Ax_T, \bar{A})$ where:

- A' is a set of imported basic algebraic specifications (without views).
- G is a set of formal parameter algebraic specifications.
- V are parameter variables.
- Σ_T, Ax_T are the signature and axioms (first order formulas) for T .
- $\bar{A}$ are hidden (or non-exported) operations.

Dynamic signatures are signatures given in a LOTOS-like form [2], *i.e.* enriched with arguments concerning the explicit communication scheme. There is a simple correspondence between dynamic and usual (static) signatures.

reception of a message m from u on the gate g $g ?m:Msg \text{ from } u:Pid$	emission of a message m to u on the gate g $g !m:Msg \text{ to } u:Pid$
constructor $g : T, Msg, Pid \rightarrow T$	constructor $g\_c : T \rightarrow T$ observer $g\_o\_m : T \rightarrow Msg$ observer $g\_o\_id : T \rightarrow Pid$

The view dynamic layer describes the possible sequences of actions the component may perform. The main idea is to define operations in terms of their effect on conditions (C) that build the component abstraction ([12] explains how these conditions are identified). Hence, with each operation, its preconditions and postconditions will be associated.

Definition 3 (View dynamic layer). A dynamic layer D_T (in a view structure (A_T, α, D_T)) is a tuple $(Id, O_T, \overline{O_T})$ where:

- Id is an identifier of sort PId_T .
- O_T is a set of operations requirements (built over dynamic signatures, pre and postconditions), *i.e.* $O_T \subseteq Term_{\Sigma_T} \times P \times Q$ where P are preconditions and Q are postconditions.
- $\overline{O_T}$ (built on $\overline{A}$) are hidden members of O_T .

The preconditions are first order formulas expressing the operation requirements in terms of the values of the C conditions (P_C) and of the $Term_{\Sigma_T}$ arguments (denoted by P_{io}).

For example, given a specific buffer operation $put(self, x)$ that is only possible with even numbers, and when the buffer is not full, we have the following precondition $\neg full \wedge even(x)$ where $full$ belongs to C (hence to P_C), and $even(x)$ belongs to P_{io} .

The postconditions are a set of first order formulas $\{Q_{C'}\}$ expressing for each operation the values C' of the C conditions *after* the operation is applied in terms of the values of the C conditions (Q_C), of some new Cl “limit” conditions¹ (Q_{Cl}) and the $Term_{\Sigma_T}$ arguments (denoted by Q_{io}) *before* the operation is applied.

Taking the same put operation, and in the case where the buffer abstraction is built over $C = \{empty, full\}$, we have the postcondition $\{Q_{empty'}, Q_{full'}\}$ and $Q_{empty'} = false, Q_{full'} = nextFull$ (*i.e.* the buffer is *full* if, before the *put* operation took place, it would be *full* the next time). As one can see, we had to use a limit condition (*nextFull*) to express the postcondition over *full*.

α uses the condition predicates (C) to define equivalence classes for T . C being finite, there is a finite number of equivalence classes.

¹ Experience shows that these are required to express the C conditions values in postconditions.

Definition 4 (View abstraction). An abstraction α (in a view structure (A_T, α, D_T)) is a tuple (C, Cl, Φ, Φ_0) where:

- $C \subseteq \Sigma_T$ is a finite set of predicates.
- $Cl \subseteq \Sigma_T$ is a finite set of “limit” conditions used in postconditions of operations.
- $\Phi \subseteq Ax_T$ is a set of formulas over members of C (valid combinations).
- $\Phi_0 \subseteq Ax_T$ is a formula over members of C (initial values).

The syntax for views in KORRIGAN is given in Figure 3. The SPECIFICATION part corresponds to A_T , the ABSTRACTION part to α , and the OPERATIONS part to D_T . There is no need to state explicitly $\overline{O}$ as it may be retrieved using $\overline{A}$. There is no identifier either since identifiers shall only appear when the component is used in a composition (external structuring views).

VIEW T			
SPECIFICATION		ABSTRACTION	
imports A'	hides $\overline{A}$	conditions C	with Φ
generic on G	ops Σ	limit conditions Cl	initially Φ_0
variables V	axioms Ax	OPERATIONS	
		O_i	pre: P post: Q

Fig. 3. The KORRIGAN syntax (views)

As explained in [4], views may be defined using either the abstraction and operation parts or Symbolic Transition Systems (STS) [6,15]. A trivial abstraction and operation parts may then be derived from the STS. Conversely, a STS may be obtained in a unique way from the abstraction and operation parts. Note that STS parts of views are given using dynamic signatures. The corresponding KORRIGAN syntax is given in Figure 4, and an example in Figure 5 (see also [4]).

VIEW T		
SPECIFICATION		STS
imports A'	hides $\overline{A}$	$\cdot \rightarrow \text{initialState}$
generic on G	ops Σ	sourceState – transition $\rightarrow$ targetState
variables V	axioms Ax	...

Fig. 4. The KORRIGAN alternative syntax (views)

One interesting property is that, as said before, views use an abstraction (α) to describe equivalence classes for the components. The STS has exactly one state per equivalence class. The number of equivalence classes being finite, the STS is also finite.

In the following, we describe the static and the dynamic views. In our unifying approach, dynamic views are the same as static views but for some differences: (i) no full algebraic specification (only signatures matter) and, (ii) the **from/to** elements (using *Pid* types). This stems from the fact that dynamic views focus only on the communication abstraction.

Static Views. Static views (SV) describe the static aspect of a component. A *static view* SV_T of a component T is a view. The password manager static view is given in Figure 5.

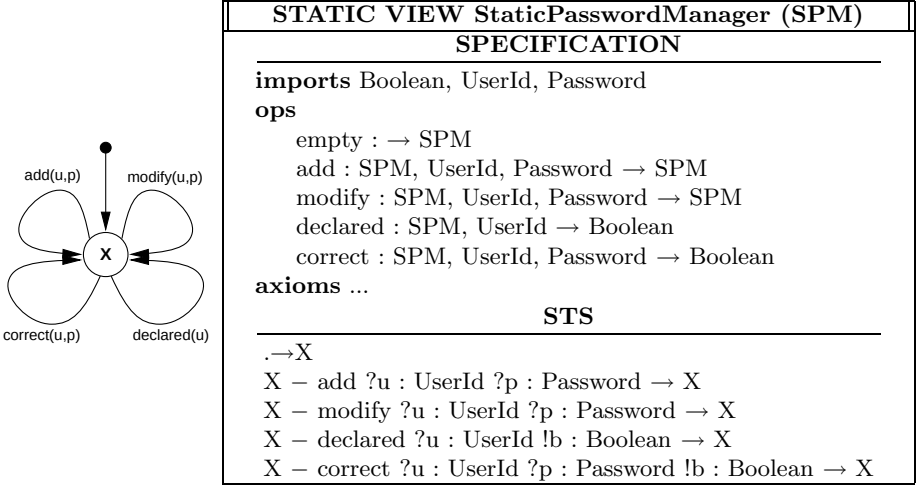


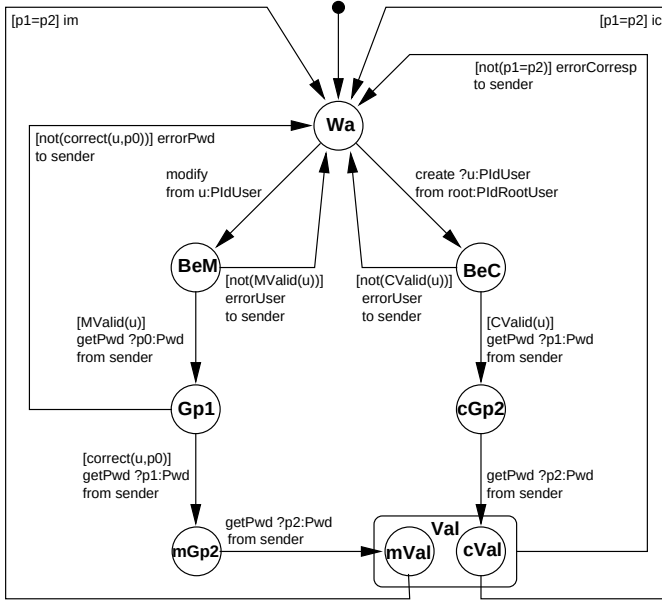
Fig. 5. Password Manager static view (SPM)

Dynamic Views. Dynamic views (DV) describe the behavioural aspect of a component, that is a process (or an active object). For dynamic views, we consider only communication protocols (emissions and receptions, in terms of senders and receivers) and not the definition of outputs in terms of inputs. In this sense, it is really a partial view (communication abstraction) of a component, without any global semantics.

A *dynamic view* DV_T of a component T is a view where the signatures (Σ_T) are dynamic and where the axiom part (Ax_T) is reduced to Φ and Φ_0 . The password manager dynamic view (graphical representation of the STS part) is given in Figure 6.

External Structuring Views. An external structuring view (ESV) may contain one or more global semantics components (*i.e.* integration views or composition views). External structuring views have a global semantics.

Definition 5 (External Structuring View). An external structuring view ESV_T of a component T is a triple (A_T, Θ, K_T) where A_T is an optional data part, Θ is the ESV “glue” part, and K_T the ESV composition layer.



state names: **Waiting**, **BeginModification**, **BeginCreation**, **Get password no.1**, **Get password no. 2** (**m** on modification and **c** on creation), **Validate** (**mVal** on modification and **cVal** on creation).

Fig. 6. Password Manager dynamic view STS (DPM)

New parameters and/or basic algebraic data types may be needed.

Definition 6 (ESV data part). An ESV data part A_T (in an ESV structure (A_T, Θ, K_T)) is made up of following components $(A', G, V, \bar{A})$ where:

- A' is a set of imported basic algebraic specifications (without views).
- G is a set of formal parameter algebraic specifications.
- V are parameter variables.
- $\bar{A}$ are hidden (or non-exported) operations.

Θ describes how the components of T are glued altogether.

Definition 7 (ESV glue part). An ESV glue part Θ (in an ESV structure (A_T, Θ, K_T)) is a tuple $(Ax_\Theta, \Phi, \Psi, \Phi_0, \delta)$ where:

- Ax_Θ are axioms relating guards and operations across different views (see the Password Manager Integration View in Figure 11).
- Φ is a (state) formula built over id-indexed C members of the components views².
- Ψ is a (transition) id-indexed formula relating (gluing) transitions from several components STS. This is a generalized form of synchronized product vector [1].

² This is used for example to express (mutual) exclusion between two conditions/states of two different components.

- Φ_0 is a (state) *id-indexed* formula expressing the initial states of the components (it should be coherent with the Φ_0 s of the composition components, that is $\vdash \Phi_0 \Rightarrow \bigwedge_{id} id.\Phi_0(Obj_{id})$).
- δ defines what to do with non-glued transitions (see Section 4).

Φ and Ψ are implicitly universally quantified over time (*i.e.* we have³ $AG\Phi$ and $AG\Psi$). Ψ is given as couples $\Psi = \{\psi_i = (\psi_{i_s}, \psi_{i_d})\}$, and means $AG \bigwedge_i \psi_{i_s} \Leftrightarrow \psi_{i_d}$. Θ uses environments, state and transition formulas.

Environments. An environment Γ binds variables to states or to transitions.
 $\Gamma : Var \rightarrow (STATE \cup TRANS)$

State and transition formulas are simple ones: there are no fixpoint or special modalities like in more expressive temporal logics such as CTL, μCRL or XTL[9]. The main point here is that this is expressive enough to express the glue between components. More complex logics may be defined afterwards to verify the models properties (following for example the [9] approach).

State Formulas. The state formulas Φ interpretation domain is the STS set of states. The deduction rules are given in Figure 7.

$$\Phi ::= c \mid x \mid \exists x.\Phi_1 \mid true \mid false \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_2 \mid -\Psi_1 \mid i.\Phi_1$$

$\frac{c(s)}{\Gamma, s \vdash c}$	$PROP_s$	
$\frac{\{x \mapsto s'\} \in \Gamma, s \sim s'}{\Gamma, s \vdash x}$	$IDENT_s$	$\frac{\Gamma \cup \{x \mapsto s\}, s \vdash \Phi_1}{\Gamma, s \vdash \exists x.\Phi_1} BIND_s$
$\overline{\Gamma, s \vdash true}$	$TRUE_s$	$\overline{\Gamma, s \not\vdash false} FALSE_s$
$\frac{\Gamma, s \not\vdash \Phi_1}{\Gamma, s \vdash \neg\Phi_1}$	NOT_s	$\frac{\Gamma, s \vdash \Phi_1 \wedge \Gamma, s \vdash \Phi_2}{\Gamma, s \vdash \Phi_1 \wedge \Phi_2} AND_s$
$\frac{\exists t = s \xrightarrow{l} s' \in T, \Gamma, t \vdash \Psi_1}{\Gamma, s \vdash -\Psi_1} TRANS$		$\frac{\exists i. s' \in s, \Gamma, s' \vdash \phi_1}{\Gamma, s \vdash i.\phi_1} INDEX_s$

Fig. 7. State formulas deduction rules

$PROP_s$: a state verifies a condition c (in the C set of conditions given in the view structure) if c is true in s (denoted by $c(s)$). $IDENT_s$: a state verifies x if x is bound to a state s' in the environment Γ , and if s and s' are similar ($s \sim s'$). Two states are *similar* if $\forall c \in C . c(s) \Leftrightarrow c(s')$. $BIND_s$: this formula is used to bind variables to states. A state s verifies $\exists x.\Phi_1$ if it verifies Φ_1 in an environment where x has been bound to s . $TRUE_s$: every state verifies *true*. $FALSE_s$: no state verifies *false*. NOT_s : a state verifies $\neg\Phi_1$ if it does not verify Φ_1 . AND_s : a state verifies $\Phi_1 \wedge \Phi_2$ if it verifies both Φ_1 and Φ_2 . $TRANS$: this state formula links state and transition formulas. A state s verifies $-\Psi_1$ if there is a transition t outgoing from s that verifies Ψ_1 . $INDEX_s$: a state verifies $i.\Phi_1$ if it has a subcomponent i that verifies Φ_1 . This rule enables us to deal with the system structuring (indexing).

³ Where AG denotes “always globally”.

Transition Formulas. The transition formulas Ψ interpretation domain is the STS set of transitions. The deduction rules are given in Figure 8.

$$\Psi ::= [g]l[g'] \mid x \mid \exists x.\Psi_1 \mid true \mid false \mid \neg\Psi_1 \mid \Psi_1 \wedge \Psi_2 \mid > \Phi_1 \mid i.\Psi_1$$

$\frac{t=s \quad \frac{[g_t] \quad l_t \quad [g'_t]}{t \xrightarrow{s} s', l \sim l_t, g_t \Rightarrow g, g'_t \Rightarrow g'} \quad \Gamma, t \vdash [g]l[g']}{\Gamma, t \vdash true} PROP_t$		
$\frac{\frac{\{x \mapsto t'\} \in \Gamma, t \sim t'}{\Gamma, t \vdash x}}{\Gamma, t \vdash x} IDENT_t$		$\frac{\Gamma \cup \{x \mapsto t\}, t \vdash \Psi_1}{\Gamma, t \vdash \exists x.\Psi_1} BIND_t$
$\overline{\Gamma, t \vdash true} TRUE_t$		$\overline{\Gamma, t \not\vdash false} FALSE_t$
$\frac{\Gamma, t \not\vdash \Psi_1}{\Gamma, t \vdash \neg\Psi_1} NOT_t$		$\frac{\Gamma, t \vdash \Psi_1 \wedge \Gamma, t \vdash \Psi_2}{\Gamma, t \vdash \Psi_1 \wedge \Psi_2} AND_t$
$\frac{t=s \xrightarrow{l} s', \Gamma, s' \vdash \Phi_1}{\Gamma, t \vdash >\Phi_1} TARGET$		$\frac{\exists i. t' \in t, \Gamma, t' \vdash \Psi_1}{\Gamma, t \vdash i.\Psi_1} INDEX_t$

Fig. 8. Transition formulas deduction rules

$PROP_t$: a t transition verifies $s \xrightarrow{[g_t] \quad l_t \quad [g'_t]} s'$ if it has similar offers, and if t guards imply the formula ones. Two offers $l = po_i$ and $l_t = p_t o_{t_j}$ are similar (denoted by $l \sim l_t$) if the communication gates have the same name ($p = p_t$), if the numbers of offer parameters are the same ($i = j$), and if these offer parameters correspond ($\forall i. o_i = o_{t_i}$). $IDENT_t$: a transition verifies x if x is bound to a transition t' in the Γ environment, and if t and t' are similar in the Γ environment ($t \sim_\Gamma t'$), that is if $\Gamma, t \vdash t'$ and $\Gamma, t' \vdash t$. $BIND_t$: this formula is used to bind variables to transitions. A transition t verifies $\exists x.\Psi_1$ if it verifies Ψ_1 in an environment where x has been bound to t . $TRUE_t$: every transition verifies $true$. $FALSE_t$: no transition verifies $false$. NOT_t : a transition verifies $\neg\Psi_1$ if it does not verify Ψ_1 . AND_t : a transition verifies $\Psi_1 \wedge \Psi_2$ if it verifies both Ψ_1 and Ψ_2 . $TARGET$: this transition formula links transition and state formulas. A transition t verifies $> \Phi_1$ if it outgoes from a state s to a state s' such that s' verifies Φ_1 . $INDEX_t$: a transition verifies $i.\Psi_1$ if it has a subcomponent i that verifies Ψ_1 . This rule enables us to deal with the system structuring (indexing).

As usual, $\models$ may be defined for states as $s \models \Phi$ iff $\{s\}, s \vdash \Phi$, and for transitions as $t \models \Psi$ iff $\{t\}, t \vdash \Psi$.

Definition 8 (ESV composition layer). An ESV composition layer K_T (in a composition view structure (A_T, Θ, K_T)) is a tuple (Id, Obj_{id}, I_{id}) where:

- Id is an identifier of sort PId_T .
- Obj_{id} is a id -indexed family of objects (integration views or composition views). Identifiers follow the equality and structuring principles.
- I_{id} are id -indexed sets of terms instantiating the components parameters.

The syntax for external structuring views in KORRIGAN is given in Figure 9.

Integration Views. Integration views (IV) are used to describe objects that have a global semantics (*i.e.* the integration of all the object aspects). Integration views are a special kind of external structuring views (same structure) with the

EXTERNAL STRUCTURING VIEW T			
SPECIFICATION		COMPOSITION δ	
imports A'	variables V	is	axioms Ax_{Θ}
generic on G	hides $\bar{A}$	$id_i : Obj_i[I_i]$	with Φ, Ψ
			initially Φ_0

Fig. 9. The KORRIGAN syntax (external structuring views)

constraint that it is composed of exactly one view per aspect of the component (for the moment, one static and one dynamic view, identified respectively by s and d). The syntax for integration views in KORRIGAN is given in Figure 10, and its application to the case study in Figure 11.

INTEGRATION VIEW T			
SPECIFICATION		COMPOSITION δ	
imports A'	variables V	is	axioms Ax_{Θ}
generic on G	hides $\bar{A}$	static $s : Obj_s[I_s]$	with Φ, Ψ
		dynamic $d : Obj_d[I_d]$	initially Φ_0

Fig. 10. The KORRIGAN syntax (integration views)

INTEGRATION VIEW Password Manager	
COMPOSITION ALONE	
is static s : SPM dynamic d : DPM axioms $d.MValid(s,u) == s.declared(s,u)$ $d.CValid(s,u) == \text{not}(s.declared(s,u))$ $d.correct(s,u,p) == s.correct(s,u,p)$	with true, { ($d.([true] \text{ im}), s.(\text{modify}(d.u,d.p1)))$), ($d.([true] \text{ ic}), s.(\text{add}(d.u,d.p1)))$ } initially true

Fig. 11. Password Manager integration view

Composition Views. A composition view (CV) is an external structuring view that may contain one or more global semantics components (*i.e.* integration views or composition views). A composition corresponds to the union of subcomponents. Its structure and the corresponding KORRIGAN syntax is the same as for external structuring views.

4 A Global Semantics for Views

External Structuring Views (ESV) have a global semantics in the sense that they specify objects with all aspects (internal structuring) well defined. First of all, we have to explain how we may obtain a view structure (such a view is global for the same reasons) for ESV. Then, we shall give its semantics. Another way could have been to give directly a semantics from the ESV structure.

ESV View Structures Retrieval. ESV view structures are a special kind of view where states (resp. transitions) are built as an indexed compound of states (transitions). This indexing enables us to keep the system internal structure while “gluing” components together. This gluing is used in order to specify how the states and transitions of the static and dynamic aspects are related.

We may work through the obtaining of a view structure for an ESV either in its (A_T, α, D_T) form, or its (A_T, STS) form (with $STS = (S, S_0, T)$). We shall work on the retrieval of a (A_T, STS) structure from the ESV structure (A_T, Θ, K_T) , as working on STS is easier.

Θ (in ESV structures, Definition 7) gives glue rules for both states and transitions. We should also define what to do with any transition that has not to be explicitly glued (*stand-alone transition*). This is the role of the δ component of Θ . We propose three different possibilities:

- **KEEP** : any stand-alone transition may happen alone (glued with some ϵ transition) or glued with any other stand-alone transition.
- **ALONE** : as in LOTOS, any stand-alone transition has to happen alone.
- **LOOSE** : stand-alone transitions are discarded (default).

The retrieval rules (Figures 12 and 13) have been implemented in [11].

Initial states are built from couples of indexed initial states of the dynamic and the static views (denoted by $\langle s.s_0/d.s_0 \rangle$) that verify the Φ and Φ_0 components of the ESV structure. The set of states is built from the initial states ($STATE_0$ rule) and the target states of the transitions ($STATE_1$ rule).

$\frac{s_0_s \in S_0(Obj_s), s_0_d \in S_0(Obj_d), \langle s.s_0/d.s_0 \rangle \models \Phi \wedge \Phi_0}{\langle s.s_0/d.s_0 \rangle \in S_0} \quad INIT$
$\frac{s \in S_0}{s \in S} \quad STATE_0 \qquad \frac{s \in S, s \xrightarrow{l} s' \in T}{s' \in S} \quad STATE_1$

Fig. 12. State related retrieval rules

A couple of transitions (one from the static view, the other from the dynamic view) that satisfies some couple of transition formulas from the Ψ component of the ESV structure are glued into a composed transition (denoted by $\langle s.s_s/d.s_d \rangle \xrightarrow{[\langle s.g_s/d.g_d \rangle] \ [\langle s.l_s/d.l_d \rangle] \ [\langle s.gio_s/d.gio_d \rangle]} \langle s.s'_s/d.s'_d \rangle$) if the corresponding composed (glued from the two transition source states) source state ($\langle s.s_s/d.s_d \rangle$) is a valid state and if the corresponding composed target state ($\langle s.s'_s/d.s'_d \rangle$) satisfies the Φ component of the ESV structure. The composition of states and transitions uses indexing in order to keep the system structure.

The rules for the $A_T = (A', G, V, \Sigma_T, Ax_T, \bar{A})$ part (from [4]) use indexing to build a product (static) type for A_T :

- $A' = A'(ESV) \cup I_{G_i} \bigcup_i A'(Obj_i(ESV))$, I_{G_i} being instantiations I_i of $G(Obj_i(ESV))$
- $G = \bigcup_i i.G(Obj_i(ESV))$, removing instantiated I_i of $G(Obj_i(ESV))$

$$\begin{array}{c}
 t_s = s_s \xrightarrow{[g_s] \quad l_s \quad [gio_s]} s'_s \in T(Obj_s), \\
 t_d = s_d \xrightarrow{[g_d] \quad l_d \quad [gio_d]} s'_d \in T(Obj_d), \\
 \exists(\psi_{i_s}, \psi_{i_d}) \in \psi \mid ts \models \psi_{i_s}, td \models \psi_{i_d}, \\
 s = \langle s.s_s/d.s_d \rangle, s' = \langle s.s'_s/d.s'_d \rangle, \\
 g = \langle s.g_s/d.g_d \rangle, gio = \langle s.gio_s/d.gio_d \rangle, l = \langle s.l_s/d.l_d \rangle, \\
 s \in S, s' \models \phi \\
 \hline
 s \xrightarrow{[g] \quad t \quad [gio]} s' \in T
 \end{array}
 \quad \begin{array}{l}
 \text{TRANSACTION} \\
 \text{(LOOSE)}
 \end{array}$$

Fig. 13. Transition related retrieval rule⁴

- $V = \bigcup_i i.V(Obj_i(ESV))$, removing instantiated I_i of $V(Obj_i(ESV))$
- $\Sigma_T = \bigcup_i i.\Sigma(Obj_i(ESV))$
- $Ax_T = Ax_{\Theta} \bigcup_i i.Ax(Obj_i(ESV))$. Some renaming and indexing has to be done on the axioms (see [4]).
- $\overline{A} = \overline{A} \bigcup_i \overline{A}(Obj_i(ESV))$

ESV Semantics. The semantics is given in an operational and rewriting way. The semantic function profile is: $Term \times \Gamma \times STATE \times EVENT \rightarrow Term \times \Gamma \times STATE$ where $EVENT$ is a communication event (LOTOS-like offer). It is written: $(t, \Gamma, s) \xrightarrow{e} (t', \Gamma', s')$. Here environments (Γ) bind terms to variables.

Components are initially instantiated. This is denoted by the application of some $*$ generator to $\vec{t}$ terms. Here $\vec{v}$ denotes a vector v of terms and $\overset{Ax, \Gamma}{\downarrow} (t)$ the normal form of term t , using the Ax axioms in the Γ environment.

$$\text{if } \langle s_s/s_d \rangle \in S_0 \text{ then } \xrightarrow{*(\vec{t})} \left(\overset{Ax_T}{\downarrow} (\vec{t}), \{\}, \langle s_s/s_d \rangle \right)$$

Then, transitions may happen. $\triangleleft$ denotes the overriding union of environments and $unify$ is a function that unifies two terms and returns an environment in which both terms are equal (substitution). Finally, $\overline{c_d}$ denotes the event that corresponds (through communication) to c_d . We use suffixes to denote inputs (i) and outputs (o) and indices to denote static ($.s$) and dynamic ($.d$) parts.

$$\begin{array}{c}
 \text{if} \\
 \langle s_s/s_d \rangle \xrightarrow{[(g_s/g_d)] \quad \langle c_s \xrightarrow{i_s} o_s/c_d \xrightarrow{i_d} o_d \rangle \quad [(gio_s/gio_d)]} \langle s'_s/s'_d \rangle \in T \\
 \sigma_{event} = unify(c_d \xrightarrow{i_d} o_d, \overline{c_d} \xrightarrow{ai} ao) \wedge \sigma_{event} \neq \emptyset \\
 \sigma_{s/d} = unify(c_s \xrightarrow{i_s} o_s, c_d \xrightarrow{i_d} o_d) \wedge \sigma_{s/d} \neq \emptyset \\
 \Gamma' = \Gamma \triangleleft \sigma_{event} \triangleleft \sigma_{s/d} \\
 \overset{Ax_T, \Gamma'}{\downarrow} (g_s) \wedge \overset{Ax_T, \Gamma'}{\downarrow} (g_d) \wedge \overset{A', \Gamma'}{\downarrow} (gio_s) \wedge \overset{A', \Gamma'}{\downarrow} (gio_d)
 \end{array}$$

⁴ Two other rules [3] take into account the specific **KEEP** and **ALONE** semantics.

$$\begin{array}{c}
 \text{then} \\
 \boxed{t, \Gamma, \langle s_s/s_d \rangle} \xrightarrow{\overline{c_d} \ \overrightarrow{a_i} \ \overrightarrow{a_o}} \boxed{\begin{array}{c} Ax_T, \Gamma' \\ \downarrow \\ (c_s(t, \overrightarrow{i_s}, \overrightarrow{o_s})) , \Gamma' , \langle s'_s/s'_d \rangle \end{array}}
 \end{array}$$

This rule expresses that, whenever a transition that has the current state as its source state may be applied then it is applied, obtaining new (current) term and environment. The definition of applicability is that (i) there is an event that unifies with the transition, and (ii) after unifying static and dynamic transitions, all guards are true.

5 Conclusion

We presented here the different views that compose our model. In our unifying approach, views are used to describe the different aspects of a component. Hence, there are static and dynamic views. This corresponds to a first structuring level that we call internal structuring view. A more complex structuring level is the external structuring view, dealing with collections of communicating components. We formally define all these concepts.

The goal of this paper is to define a global semantics for external structuring views. An external structuring view may contain one or more global semantics components (*i.e.* integration views or composition views). An ESV has a data part, a glue part and a composition layer. The glue part is a triple of formulas: a state formula, a transition formula and a state initial formula. Therefore, we define logical rules for a state or a transition to verify a formula. The ESV composition layer is an indexed family of objects views.

The definition of a global semantics for such an ESV is done in two steps. First, we explain how we may obtain a view structure from an ESV. ESV view structures are a special kind of view where states (resp. transitions) are built as an indexed compound of states (transitions). This indexing enables us to keep the system internal structure while “gluing” components together. This gluing is used in order to specify how the states and transitions of the static and dynamic aspects are related. Then, we give the semantics in an operational style, based on rewriting over the state and the dynamic parts. The main rule states how, from a given system state (a state, a term and an environment), an event rewrites the system state. Another way could have been to give directly a semantics from the ESV structure.

Future work will deal with a proof that the semantics diagram commutes and the definition of an adequate model-checking procedure for our view structures.

References

1. André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Etudes et recherches en informatique. Masson, 1992.
2. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

3. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. A Global Semantics for Views. Research Report 189, IRIN, 1999. /papers/rr189.ps.gz in Poizat's web page.
4. Christine Choppy, Pascal Poizat, and Jean-Claude Royer. Control and Datatypes using the View Formalism. Research Report 188, IRIN, 1999. /papers/rr188.ps.gz in Poizat's web page.
5. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
6. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
7. ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.
8. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
9. Radu Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. PhD thesis, Institut National Polytechnique de Grenoble, 1998.
10. José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR'96 : Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372, Pisa, Italy, 1996.
11. Gaël Nedelec, Marielle Papillon, Christelle Piednoirs, and Gwen Salaün. CLAP: a Class Library for Automata in Python. Master's thesis, 1999. CLAP.html in Poizat's web page.
12. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. Concurrency and Data Types: A Specification Method. An Example with LOTOS. In J. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop on Algebraic Development Techniques WADT'98*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291, Lisbon, Portugal, 1999. Springer-Verlag.
13. Pascal Poizat, Christine Choppy, and Jean-Claude Royer. From Informal Requirements to COOP: a Concurrent Automata Approach. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag.
14. Gianna Reggio and Mauro Larosa. A graphic notation for formal specifications of dynamic systems. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61. Springer-Verlag, September 1997. ISBN 3-540-63533-5.
15. Carron Shankland, Muffy Thomas, and Ed Brinksma. Symbolic Bisimulation for Full LOTOS. In *Algebraic Methodology and Software Technology AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag.
16. Graeme Smith. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
17. Rational Software. Unified Modeling Language, Version 1.1. Technical report, Rational Software Corp, <http://www.rational.com/uml>, September 1997.

18. Martin Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990. Jan Van Leeuwen, Editor.

Analysis of Downward Closed Properties of Logic Programs^{*}

Patricia M. Hill¹ and Fausto Spoto^{2**}

¹ School of Computer Studies, University of Leeds, Leeds, LS2 9JT, UK
hill@scs.leeds.ac.uk

² Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56100 Pisa, Italy
Ph.: +39-(0)50887248, Fax: +39-(0)50887226
spoto@di.unipi.it

Abstract. We study the analysis of downward closed properties of logic programs, which are a very abstract presentation of types. We generalise to a very large class of downward closed properties the construction of the traditional domains for groundness analysis in such a way that the results enjoy the good properties of that domain. Namely, we obtain abstract domains with a clear representation made of logical formulas and with optimal and well-known abstract operations. Moreover, they can be built using the linear refinement technique, and, therefore, are provably *optimal* and enjoy the condensing property, which is very important for a goal-independent analysis.

Keywords: *Abstract interpretation, domain theory, linear refinement, type theory, type analysis, logic programming.*

1 Introduction

A downward (instantiation) closed set of terms represents a property which is maintained during the computation. Therefore, it is a very abstract definition of a type. The analysis of downward closed properties is therefore a general way of doing type analysis, i.e., the upward approximation of the success set of a program through types. Type analysis of logic programs is important for verification as well as optimisation of unification. Note that other interesting properties of logic programs, like freeness and sharing, are not types since they are not downward closed.

An important and simple type, which distinguishes whether a term contains variables or not, is groundness [1,5,6]. The usual domain for groundness analysis, *Pos*, features some desirable properties: simplicity, human readability, effectivity, usefulness. Moreover, it has been shown [13,15] that *Pos* is condensing and is the most precise domain for groundness analysis that does not consider the name

^{*} Part of this work was supported by EPSRC grant GR/M05645.

^{**} Part of this work was done while Fausto Spoto was visiting the School of Computer Studies of the University of Leeds, UK.

of the functors in a term. All these good properties of the *Pos* domain should have encouraged a generalisation of *Pos* to a general type domain. Instead, type domains have been developed, up to now, in a way totally independent from the domain *Pos* for groundness analysis. In [16] the authors define generic type domains with dependencies but, again, this is not a real generalisation of *Pos*, though it can be used for abstract compilation. The only paper which generalises the definition of the domain *Pos* to generic types is [3]. However, in this paper it is assumed that the usual properties of *Pos* hold for all the type domains. For instance, logical conjunction between formulas is used as conjunction operator and Schröder elimination as cylindrification operator. It is not obvious at all that these operators, which are optimal in the case of groundness analysis [6], are even correct in the general case of type analysis and no proof is given. Although in [4] a domain with properties similar to those of *Pos* is built, the way it is constructed is not a generalisation of that used for *Pos* [15]. Indeed, the domain defined in [4] is not made of logical formulas and cannot be obtained by linear refinement of a basic domain for the type properties.

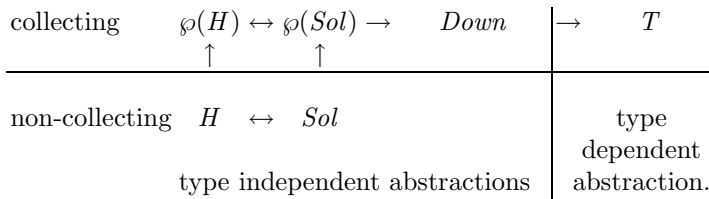
In [12] it is shown that a hierarchy of domains for type analysis can be defined in the same way as it has been done in the case of groundness analysis [15]. Transfinite formulas, which are needed to deal with polymorphism, are used to represent the type domain and generalise the finite formulas of *Pos*. It is shown how a finite representation of these formulas can be achieved through the use of type variables. Two operators, logical conjunction and Schröder elimination, are used as abstract operators over transfinite formulas and are the generalisation of the same operators used for the *Pos* domain for groundness analysis.

In this paper we use the abstract interpretation framework [7,8] to generalise the correctness result of [6] and [12] to a much larger class of type domains. Namely, we show that the abstract operators defined in these two papers are correct for every type domain, including then groundness and non-freeness, which were not covered in [12]. Moreover, we present a very weak condition, sufficient to entail the optimality of the abstract operators for a large class of type systems, including groundness, non-freeness and the type systems considered in [12] when speaking of correctness. Note that no optimality result were given in [12], while [6] proves optimality for the case of groundness only. The optimality result is important since it shows that, when our sufficient condition is satisfied, the use of transfinite formulas as a representation of type domains does not introduce any loss of precision for the computation of the abstract operators.

The correctness and optimality result proved in [6] for the two operations over the formulas of *Pos* cannot easily be lifted to a generic type domain. Given a different type domain, every proof should be rewritten. This is because a direct (Galois) connection between the concrete domain of existential Herbrand constraints and the abstract domain for groundness analysis is defined, and this connection is not parametrised with respect to a generic type. Instead, we use a two-step generic abstraction from existential Herbrand constraints into a generic type domain. In the first step we show that the collecting domain of existential Herbrand constraints can be abstracted into the set of downward closed sets of

substitutions, which we call *Down*. A downward closed set of substitutions represents the union of the solutions of a set of existential Herbrand constraints. We provide the optimal abstract operators on *Down* induced by the corresponding operators on existential Herbrand constraints. Note that this first step is independent from the type analysis at hand. We consider the particular analysis only in the second step when we abstract *Down* into a type domain of transfinite formulas. Since the distance between type domains and *Down* is very much shorter than the distance between type domains and existential Herbrand constraints, the definition of a Galois connection between *Down* and the type domain is rather obvious. Moreover, it is quite easy to provide the optimal operators on the type domains induced by the optimal operators on *Down*.

The paper is organised as follows. Section 3 introduces the domain *Down*, whose constraints are downward closed sets of substitutions and shows a subset *Sol* of *Down* which is isomorphic to the set *H* of existential Herbrand constraints. Section 4 defines the domain $\wp(H)$ used for a collecting version of the semantics on *H* and its isomorphic counterpart $\wp(\text{Sol})$, and shows that it can be abstracted into our *concrete* domain *Down*. This time, the whole domain *Down* will be used, rather than its subset *Sol*. From now on, the abstraction will be type-specific. Namely, Section 5 defines a generic type system and its induced type domain *T* formed by transfinite formulas. Moreover, it provides correct operators on *T* corresponding to the operators on *Down*, and shows sufficient conditions for their optimality. Section 6 shows that our work generalises some old results and provides logical domains for type analysis of logic programs. Section 7 shows the application of the linear refinement technique for developing abstract domains able to model conjunction in a very precise way. Some concluding remarks are discussed in Section 8. The picture below synthesises the various domains considered in this paper, and their relationships as abstraction (represented by horizontal arrows) or lifting to the powerset (represented by vertical arrows).



For reasons of space, proofs are omitted.

2 Preliminaries

2.1 Terms and Substitutions

The *powerset* of a set *S* is $\wp(S) = \{S' \mid S' \subseteq S\}$. We denote by $\wp_f(S)$ the set of all finite subsets of *S*. If *S* is partially ordered w.r.t. $\leq$, we denote by $\wp\downarrow(S)$ the set of all the downward closed sets of *S*, i.e., $\wp\downarrow(S) = \{s \in \wp(S) \mid \text{if } x \in$

s and $y \leq x$ then $y \in s$. A sequence is an ordered collection of elements. If $\tilde{x}$ is a sequence of elements of S , we write $\tilde{x} \subseteq S$.

Given a set of variables V and a set of function symbols Σ with associated arity, we define $\text{terms}(\Sigma, V)$ as the set of terms built from V and Σ . We denote by $\text{vars}(t)$ the set of variables which occur in a term t . When $\text{vars}(t) = \emptyset$ we say that the term t is ground. Given a set of variables V and a variable x , $V \cup x$ means $V \cup \{x\}$ and $V \setminus x$ means $V \setminus \{x\}$.

We define $\Theta_{V,W}^Z$, with $Z \subseteq V \cap W$, as the set of substitutions θ such that $\text{dom}(\theta) \subseteq V$, $\theta(x) \in \text{terms}(\Sigma, W)$ for every $x \in V$ and $\text{dom}(\theta) \cap \text{rng}(\theta) \subseteq Z$. If $Z = \emptyset$ we omit the superscript. The elements of $\Theta_{V,W}$ are called idempotent substitutions. We write Θ_V for $\Theta_{V,V}$ and Θ_V^Z for $\Theta_{V,V}^Z$. Given θ and a set of variables R , we define $\theta|_R(x) = \theta(x)$ if $x \in R$ and $\theta|_R(x) = x$ otherwise. Given a term $t \in \text{terms}(\Sigma, V)$ and $\theta \in \Theta_{V,W}^Z$, $t\theta \in \text{terms}(\Sigma, W)$ is the term obtained with parallel substitution of every variable x in t with $\theta(x)$. Composition of substitutions $\theta \in \Theta_{V,W}$ and $\sigma \in \Theta_{W,Z}$ is defined as $(\theta\sigma)(x) = \theta(x)\sigma$ for every $x \in V \cup W$. If $f : \wp(\text{terms}(\Sigma, V)) \rightarrow \wp(\text{terms}(\Sigma, V))$ then $\mu t.f(t)$ denotes the least fixpoint of f .

For every set of variables V , a preorder is defined on Θ_V as $\theta' \leq_V \theta$ if there exists a substitution $\sigma \in \Theta_V^V$ such that $\theta' = \theta\sigma$. When V is clear from the context, we write $\leq$ instead of $\leq_V$. A preorder, called subsumption, is defined on $\text{terms}(\Sigma, V)$ as $t_1 \leq_V t_2$ (t_1 is an instance of t_2) if $t_1 = t_2\theta$ for a suitable $\theta \in \Theta_V^V$. As usual, the subscript is omitted in $\leq_V$ when it is clear from the context. By $\equiv$ we denote the associated equivalence relation (*variance*).

2.2 Existential Herbrand Constraints

Let Σ be a set of function symbols with associated arity and V a finite set of variables. We define the set of finite sets of Herbrand equations C_V as

$$C_V = \wp_f(\{t^1 = t^2 \mid t^1, t^2 \in \text{terms}(\Sigma, V)\}) .$$

Suppose $c \in C_V$. Then we say that $c\theta$ is *true* if $t^1\theta$ is syntactically equal to $t^2\theta$ for every $(t^1 = t^2) \in c$. We know [14] that if there exists $\theta \in \Theta_V$ such that $c\theta$ is true, then c can be put in *normal form* $\text{mgu}(c) = \{t_j^1 = t_j^2 \mid j \in J\}$, where $t_j^1 \in V$ are distinct variables which do not occur in t_k^2 for every $j, k \in J$ and the variables in $\text{mgu}(c)$ are all contained in the variables of c . Moreover, we have $c\theta$ is true if and only if $\text{mgu}(c)\theta$ is true. If no θ exists such that $c\theta$ is true, then $\text{mgu}(c)$ and hence the normal form of c are undefined. If c is in normal form, c can be seen as a substitution, and we use the notation $c(v)$ meaning the term t on the right of an equality $v = t \in c$ if such an equality exists, v itself otherwise. Moreover, every substitution can be seen as an existential Herbrand constraint without existential variable. Namely, we define $\text{Eq}(\theta) = \exists_\emptyset \{v = \theta(v) \mid v \in \text{dom}(\theta)\}$.

Let $\mathcal{V}$ and $\mathcal{W}$ be disjoint infinite sets of variables. For each $V \in \wp_f(\mathcal{V})$, we have a set of constraints, called existential Herbrand constraints, given by

$$H_V = \left\{ \exists_W c \left| \begin{array}{l} W \in \wp_f(\mathcal{W}), \ c \in C_{V \cup W} \\ \text{there exists } \theta \in \Theta_{V \cup W} \text{ such that } \text{rng}(\theta) \subseteq V \text{ and } c\theta \text{ is true} \end{array} \right. \right\} .$$

Here, $\mathcal{V}$ are called the *program variables* and $\mathcal{W}$ the *existential variables*. The set of solutions of an existential Herbrand constraint is defined as

$$\text{sol}_V(\exists_W c) = \{\theta|_V \mid \theta \in \Theta_{V \cup W}, \text{rng}(\theta) \subseteq V \text{ and } c\theta \text{ is true}\}.$$

Hence $\text{sol}_V(\exists_W c) = \text{sol}_V(\exists_W \text{mgu}(c))$. A constraint $\exists_W c$ is said to be in *normal form* if c is in normal form. Note that $\text{sol}_V(h)$ is a downward closed set.

A preorder is defined on H_V as $h_1 \leq h_2$ if and only if $\text{sol}_V(h_1) \subseteq \text{sol}_V(h_2)$. This preorder becomes a partial order if we consider equivalence classes of constraints, where $h_1, h_2 \in H_V$ are called *equivalent* if and only if $\text{sol}_V(h_1) = \text{sol}_V(h_2)$. In the following, a constraint will stand for its equivalence class. Since, as shown above, every constraint can be put in an equivalent normal form, in the following we will consider only normal constraints.

In order for a domain to be used for program analysis, a conjunction and a restriction operation must be provided, together with *diagonal elements*, used for variable renaming.

Definition 1. Let $\{W_1, W_2\} \subseteq \mathcal{W}$ with $W_1 \cap W_2 = \emptyset$ ¹ and $x \in V$. We define

$$(\exists_{W_1} c_1) \star^{H_V} (\exists_{W_2} c_2) = \begin{cases} \exists_{W_1 \cup W_2} \text{mgu}(c_1 \cup c_2) & \text{if } \text{mgu}(c_1 \cup c_2) \text{ exists} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\exists_x^{H_V} (\exists_W c) = \exists_{W \cup N} c[N/x] \quad \text{where } N \in \mathcal{W} \text{ is fresh.}$$

Finally, given two disjoint sequences of variables $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$ in V , we define $\delta_{\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle}^{H_V} = \{x_i = y_i \mid i = 1, \dots, n\}$.

2.3 Abstract Interpretation

Abstract interpretation [8] is a theory developed to reason about the abstraction relation between two different domains (the *concrete* and the *abstract* domain).

Definition 2. Let $(C, \leq)$ and $(A, \preceq)$ be two complete lattices (the *concrete* and the *abstract* domain). A Galois connection $\langle \alpha, \gamma \rangle : (C, \leq) \rightleftharpoons (A, \preceq)$ is a pair of maps $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ such that α and γ are monotonic, for each $x \in C$ we have $x \leq (\gamma \circ \alpha)(x)$, and for each $y \in A$ we have $(\alpha \circ \gamma)(y) \preceq y$. Moreover, a Galois insertion (of $(C, \leq)$ into $(A, \preceq)$) $\langle \alpha, \gamma \rangle : (C, \leq) \rightleftharpoons (A, \preceq)$ is a Galois connection where $\alpha \circ \gamma = \text{Id}_A$.

The map α (γ) is called the *abstraction* (*concretisation*). The composition of Galois insertions is a Galois insertion.

In a Galois insertion, γ is one-to-one if and only if α is onto if and only if $\alpha \circ \gamma = \text{Id}_A$, α is additive and γ is co-additive, the abstraction map uniquely

¹ Note that this is not restrictive because the names of existential variables are irrelevant: given an existential Herbrand constraint $\exists_W c$, the constraint $\exists_{W'} c[W'/W]$ is equivalent to it. Hence we can always assume existential Herbrand constraints to be renamed apart with respect to existential variables.

determines the concretisation map and vice versa. Namely, $\gamma = \lambda y. \sqcup_C \{x \in C \mid \alpha(x) \preceq y\}$ and $\alpha = \lambda x. \sqcap_A \{y \in A \mid x \leq \gamma(y)\}$. Conversely, if C and A are complete lattices and $\alpha : C \rightarrow A$ is additive or $\gamma : A \rightarrow C$ is co-additive, then $\langle \alpha, \gamma \rangle$ is a Galois connection from C to A .

When, in a Galois connection $\langle \alpha, \gamma \rangle$, γ is not injective, several distinct elements of the abstract domain $(A, \preceq)$ have the same meaning (by γ). This is usually considered useless.

A map $\rho : L \rightarrow L$ is a *closure operator* if it is monotonic, extensive and idempotent. The equivalence between Galois insertions and closure operators is well-known [2]. In view of this equivalence, abstract domains can be studied independently from the representation chosen for their abstract objects. Assume L to be a complete lattice playing the role of the concrete domain.

Theorem 1 ([7]). *Let $A \subseteq L$ be a complete lattice and $\rho : L \rightarrow L$. We have that $\langle \rho, id \rangle : (L, \leq_L) \rightleftharpoons (A, \leq_L)$ is a Galois insertion if and only if ρ is an upper closure operator such that $\rho(L) = A$.*

Any closure operator $\rho : L \rightarrow L$ is identified by the set of its fixpoints, which is its image $\rho(L)$. Recall that $\rho(L)$ is a *Moore family* of L , i.e., a complete meet-sublattice of L (i.e., $\top_L \in \rho(L)$ and for any non-empty $Y \subseteq \rho(L)$ we have $\sqcap_L Y \in \rho(L)$). Moreover, any Moore family of L is the image of some upper closure operator on L . For any $X \subseteq L$, we denote by $\bigwedge(X) = \{\sqcap_L I \mid I \subseteq X\}$ the *Moore closure* of X , i.e., the least Moore family of L containing X .

Let $f : C^n \rightarrow C$ be a concrete operator and assume that $\tilde{f} : A^n \rightarrow A$. Then $\tilde{f}$ is correct with respect to f if and only if for all $y_1, \dots, y_n \in A$ we have $\alpha(f(\gamma(y_1), \dots, \gamma(y_n))) \preceq \tilde{f}(y_1, \dots, y_n)$. For each operator f , there exists an optimal (most precise) correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \dots, y_n) = \alpha(f(\gamma(y_1), \dots, \gamma(y_n)))$.

Given an abstract domain A , a domain refinement operator R yields an abstract domain $R(A)$ which is more precise than A , i.e., which contains more points than A . Classical domain refinement operators are *reduced product* and *disjunctive completion* [10]. The reduced product $A \sqcap B$ of two domains A and B , both contained in C , is isomorphic to the Cartesian product of A and B , modulo the equivalence relation $\langle a_1, b_1 \rangle \equiv \langle a_2, b_2 \rangle$ if and only if $a_1 \sqcap b_1 = a_2 \sqcap b_2$. This means that pairs having the same *meaning* are identified.

Linear refinement, proposed in [11], allows to include in a domain the information relative to propagation of the abstract property of interest before and after the application of a concrete operator $\boxtimes$ (in our case, $\boxtimes$ is the point-wise extension of unification).

Let L and C be complete lattices such that $L \subseteq C$, and let $\boxtimes$ be a commutative partial map $\boxtimes : L \times L \rightarrow L$. For any $a, b \in L$, we define $a \rightarrow^{\boxtimes} b = \bigsqcup_C \{c \in C \mid \text{if } a \boxtimes c \text{ is defined then } a \boxtimes c \leq_C b\}$. The linear refinement domain $L \rightarrow^{\boxtimes} L$ is $L \rightarrow^{\boxtimes} L = L \sqcap \bigwedge \{a \rightarrow^{\boxtimes} b \mid a, b \in L\}$, which can be simplified into

$$L \rightarrow^{\boxtimes} L = \bigwedge \{a \rightarrow^{\boxtimes} b \mid a, b \in L\} \quad (1)$$

if the elements of L can be obtained as greatest lower bounds of arrows.

3 The Domains *Down* and *Sol*

In this section we define a domain of downward closed sets of substitutions. Given a substitution θ , its downward closure represents the set of substitutions which are *consistent* with θ , i.e., which can be derived from θ with computation. For instance, if in a program point we have $y = f(x)$, then it is possible that, as computation proceeds, we have $y = f(g(w))$ and $x = g(w)$. It is not possible, however, that $y = g(w)$. With this interpretation in mind, we can say that a downward closed set of substitutions contains exactly all substitutions which are consistent with the prosecution of computation. For instance, if S_1 is the (downward closed) set of substitutions which are consistent with a procedure call p_1 and S_2 is the (downward closed) set of substitutions which are consistent with a procedure call p_2 , then $S_1 \cap S_2$ is the (downward closed) set of substitutions which are consistent with the calls p_1, p_2 and p_2, p_1 .

Definition 3 (The *Down* domain). We define the family of sets $Down = \{Down_V\}_{V \in \wp_f(\mathcal{V})}$ where $Down_V = \wp \downarrow (\Theta_V)$. Let $\{S, S_1, S_2\} \subseteq Down_V$ and $x \in V$. We define

$$S_1 \star^{Down_V} S_2 = S_1 \cap S_2$$

$$\exists_x^{Down_V} S = \left\{ \theta' \in \Theta_V \mid \begin{array}{l} \text{there exist } \sigma \in S[n/x], \theta \in \Theta_{V \cup n, V \cup n} \\ \text{such that } \theta|_V = \theta' \text{ and } \theta \leq_{V \cup n} \sigma \end{array} \right\},$$

where $n \in \mathcal{V} \setminus V$ is fresh, $S[n/x] = \{\sigma[n/x] \mid \sigma \in S\}$ and $\sigma[n/x] \in \Theta_{V \cup n, V \cup n}$ is defined as $\sigma[n/x](x) = x$, $\sigma[n/x](n) = \sigma(x)[n/x]$ and $\sigma[n/x](y) = \sigma(y)[n/x]$ if $y \neq x$ and $y \neq n$.

Given two disjoint sequences $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$ in V , we define

$$\delta_{\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle}^{Down_V} = \{\theta \in \Theta_V \mid \theta(x_i) = \theta(y_i) \text{ for all } i = 1, \dots, n\}.$$

While the definition of conjunction is the classical one for the case of downward closed sets of substitution (see, for instance, [15]), and is justified by the above considerations, it turns out that an *explicit* definition of restriction on downward closed sets of substitutions was never given. The conjunction operator is closed on $Down_V$, as it can be checked easily. The same holds for restriction.

Proposition 1. Let $S, S_1, S_2 \subseteq \Theta_V$.

- i) If $S = \downarrow S$ then $\exists_x^{Down_V} S = \downarrow(\exists_x^{Down_V} S)$ (restriction is closed on the set of downward closed sets of substitutions).
- ii) If $S_1 \subseteq S_2$, then $\exists_x^{Down_V} S_1 \subseteq \exists_x^{Down_V} S_2$ (restriction is monotonic).
- iii) $S \subseteq \exists_x^{Down_V} (S)$ (restriction is extensive).

An existential Herbrand constraint $h \in H_V$ can be mapped into a downward closed set of substitutions through the map sol_V which yields the set of its solutions. However, this map is not onto.

Proposition 2. If Σ contains at least a constant and a functor symbol, then $\{\text{sol}_V(h) \mid h \in H_V\} \subset \wp \downarrow (\Theta_V)$.

In spite of this result, the following proposition shows that $\star^{Down_V}$ and $\exists^{Down_V}$ are closed on the set $\text{sol}_V(H_V)$.

Proposition 3. *Let $\{h, h_1, h_2\} \subseteq H_V$ and $\tilde{x}, \tilde{y} \subseteq V$ disjoint and of the same length.*

- i) $\text{sol}_V(h_1) \star^{Down_V} \text{sol}_V(h_2) = \text{sol}_V(h_1 \star^{H_V} h_2),$
- ii) $\exists_x^{Down_V} \text{sol}_V(h) = \text{sol}_V(\exists_x^{H_V} h)$ for every $x \in V,$
- iii) $\delta_{\tilde{x}, \tilde{y}}^{Down_V} = \text{sol}_V(\delta_{\tilde{x}, \tilde{y}}^{H_V}).$

Therefore, we can introduce the following definition.

Definition 4 (The Sol domain). *We define $Sol = \{\text{Sol}_V\}_{V \in \wp_f(V)}$, where $\text{Sol}_V = \{\text{sol}_V(h) \mid h \in H_V\}$. The $\star^{Sol_V}$ and $\exists^{Sol_V}$ operators on Sol_V are the restriction of the corresponding operators of $Down_V$ to Sol_V . The diagonal elements of Sol_V are the diagonal elements of $Down_V$.*

Since sol_V is one-to-one and onto from H_V into Sol_V , and for every $\{h_1, h_2\} \subseteq H_V$ we have $h_1 \leq h_2$ if and only if $\text{sol}_V(h_1) \subseteq \text{sol}_V(h_2)$, we conclude that sol_V , endowed with the $\subseteq$ partial ordering, is isomorphic to H_V , and we know that the corresponding operations coincide (Proposition 3). The usefulness of Sol is that of presenting an existential Herbrand constraint as the set of its solutions. This will be very important in the next section.

4 The Collecting Semantics

Now we know that existential Herbrand constraints are essentially the same as their sets of solutions. This isomorphism can be lifted to an isomorphism between $\wp(H)$ and $\wp(Sol)$, the domains used for the collecting semantics. The domain $\wp(Sol)$ is not exactly what we are looking for. Indeed, we want to be able to represent every downward closed set of substitutions rather every set of sets of solutions. This means that we want to use $Down_V$ to represent our abstract properties. We prove here that there is a Galois insertion from $\wp(Sol_V)$ into $Down_V$ for every $V \in \wp_f(V)$.

We use $\cup$ as abstraction map from $\wp(Sol_V)$ into $Down_V$. The map $\cup$ is onto from $\wp(Sol_V)$ into $Down_V$. This is because every $d \in Down_V$ can be written as the infinite union $d = \cup\{\downarrow\{\theta\} \mid \theta \in d\}$ and every single $\downarrow\{\theta\}$ belongs to Sol_V , as shown by the following proposition.

Proposition 4. *Given $\theta \in \Theta_V$ we have $\text{sol}_V(\text{Eq}(\theta)) = \downarrow\{\theta\}$.*

It can be shown that $\cup$ is additive from $\wp(Sol_V)$ into $Down_V$.

Proposition 5. *Given $\{p_i\}_{i \in I} \subseteq \wp(Sol_V)$ with $I \subseteq \mathbb{N}$, we have $\cup\left(\bigvee_{i \in I}^{\wp(Sol_V)} p_i\right) = \bigvee_{i \in I}^{Down_V} (\cup p_i).$*

The above proposition and the fact that $\cup$ is onto entail that $\cup$ induces a Galois insertion from $\wp(Sol_V)$ into $Down_V$. Moreover, it can be shown that the operators and the diagonal elements on $Down_V$ are the best abstraction through $\cup$ of the corresponding operators and diagonal elements on $\wp(Sol_V)$.

Proposition 6. *Let $\{p_1, p_2, p\} \subseteq \wp(\text{Sol}_V)$ and $\tilde{x}$ and $\tilde{y}$ be two disjoint sequences in V of the same length.*

- i) $\cup (p_1 \star^{\wp(\text{Sol}_V)} p_2) = (\cup p_1) \star^{\text{Down}_V} (\cup p_2).$
- ii) $\cup (\exists_x^{\wp(\text{Sol}_V)} p) = \exists_x^{\text{Down}_V} (\cup p).$
- iii) $\cup (\delta_{\tilde{x}, \tilde{y}}^{\wp(\text{Sol}_V)}) = \delta_{\tilde{x}, \tilde{y}}^{\text{Down}_V}.$

5 Type Systems and Type Domains

We have shown that the collecting semantics over existential Herbrand constraints can be abstracted into a semantics over the domain Down . This domain allows to represent *every* downward closed property of logic programs. In general, the full power of Down is not needed. For instance, we might be interested in *some* downward closed property, like a set of types and their dependencies. This means that we want to abstract Down into a domain for a specific *type system*, where a type system specifies which downward closed properties we are interested in. From now on the abstraction becomes type-dependent.

Definition 5. *A type system is $\langle \Delta, \Sigma, I \rangle$ where Δ and Σ , called type signature and term signature, respectively, are sets of function symbols with associated arities and $I(\mathbf{f}^n)$ is a total map from $(\wp \downarrow (\text{terms}(\Sigma, \mathcal{V})))^n$ to $\wp \downarrow (\text{terms}(\Sigma, \mathcal{V}))$ for every $\mathbf{f}^n \in \Delta$. We define $\llbracket \mathbf{f}^0 \rrbracket I = I(\mathbf{f}^0)$ and $\llbracket \mathbf{f}^n(t_1, \dots, t_n) \rrbracket I = I(\mathbf{f}^n)(\llbracket t_1 \rrbracket I, \dots, \llbracket t_n \rrbracket I).$*

Example 1 (Type system for groundness). Let $\mathbf{G} = \langle \{\mathbf{g}^0\}, \Sigma, I \rangle$ and let $I(\mathbf{g}) = \text{terms}(\Sigma, \emptyset)$. The functor symbol $\mathbf{g}$ is interpreted as the set of terms which do not contain variables.

Example 2 (Type system for non-freeness). Let $\mathbf{NF} = \langle \{\mathbf{nf}^0\}, \Sigma, I \rangle$ where $I(\mathbf{nf}) = \text{terms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$. The functor symbol $\mathbf{nf}$ is interpreted as the set of terms which are not in $\mathcal{V}$, i.e., the set of non-free terms.

Example 3 (Type system for integers, lists and trees). Let $\mathbf{ILT} = \langle \{\mathbf{int}^0, \mathbf{list}^1, \mathbf{tree}^1\}, \Sigma, I \rangle$, where Σ contains $\{0^0, \mathbf{s}^1, []^0, [[]]^2, \mathbf{void}^0, \mathbf{tree}^3\}$ and

$$\begin{aligned} I(\mathbf{int}) &= \mu i. \{0\} \cup \{\mathbf{s}(n) \mid n \in i\} & I(\mathbf{list}) &= \lambda \tau. \mu l. \{[]\} \cup \{[h|t] \mid h \in \tau, t \in l\} \\ I(\mathbf{tree}) &= \lambda \tau. \mu t. \{\mathbf{void}\} \cup \{\mathbf{tree}(n, l, r) \mid n \in \tau, l \in t, r \in t\}. \end{aligned}$$

Since $\mathbf{list}$ and $\mathbf{tree}$ are unary, we can write $\mathbf{list}(\mathbf{int})$ and $\mathbf{tree}(\mathbf{list}(\mathbf{int}))$.

Given a type system, we can define a type domain. Type domains are formed by transfinite propositional formulas, defined below.

Definition 6. Given a type signature Δ and a set of variables V , the set $\Phi_{\Delta,V}$ of transfinite formulas over Δ and V is defined as the least set such that $x \in t$ is a transfinite formula, where $x \in V$ and $t \in \text{terms}(\Delta, \emptyset)$, if S is a set of transfinite formulas then $\bigwedge S$ and $\bigvee S$ are transfinite formulas, if ϕ_1 and ϕ_2 are transfinite formulas then $\phi_1 \Rightarrow \phi_2$ is a transfinite formula and if ϕ is a transfinite formula then $\neg\phi$ is a transfinite formula. We write $\phi_1 \wedge \phi_2$ for $\bigwedge\{\phi_1, \phi_2\}$, true for $\bigwedge\emptyset$ and false for $\bigvee\emptyset$.

Definition 7 (Type domain). Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system. A type domain for T is a family $T = \{T_V\}_{V \in \wp_f(V)}$ where $T_V \subseteq \Phi_{\Delta,V}$ contains the diagonal elements $\delta_{\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle}^{T_V} = \bigwedge_{d \in \text{terms}(\Delta, \emptyset), i=1, \dots, n} \{(x_i \in d \Rightarrow y_i \in d) \wedge (y_i \in d \Rightarrow x_i \in d)\}$ for every $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$ in V disjoint, and is closed w.r.t.

$$\begin{aligned} \phi_1 \star^{T_V} \phi_2 &= \phi_1 \wedge \phi_2 \\ \exists_x^{T_V} \phi &= \bigvee \left\{ \phi[P/x] \mid \begin{array}{l} \text{there exists } t \in \text{terms}(\Sigma, V) \text{ such} \\ \text{that } P = \{d \in \text{terms}(\Delta, \emptyset) \mid t \in [d]I\} \end{array} \right\}, \end{aligned}$$

where, for any $P \in \wp(\text{terms}(\Delta, \emptyset))$,

$$(x \in t)[P/x] = \begin{cases} \text{true} & \text{if } \cap_{p \in P} [p]I \subseteq [t]I \\ \text{false} & \text{otherwise} \end{cases} \quad (y \in t)[P/x] = (y \in t)$$

and the other cases are similar.

Note that if $\text{terms}(\Delta, \emptyset)$ is finite then the set $\Phi_{\Delta,V}$ is finite too, and the resulting type domains are finite. Moreover, the conjunction and the restriction operations become effective, while they are not computable in the general case. Finally, note that, for the type domains of Examples 1 and 2, restriction becomes the classical Schröder elimination [1], since $\text{terms}(\Delta, \emptyset)$ contains only a constant symbol whose evaluation is not empty.

Definition 8. Given a type system $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$, we define the map $\llbracket \cdot \rrbracket_{\mathsf{T}} : \Phi_{\Delta,V} \rightarrow (\Theta_V \rightarrow \{0, 1\})$ as

$$\llbracket x \in t \rrbracket_{\mathsf{T}} \sigma = \begin{cases} 1 & \text{if } \sigma(x) \in [t]I \\ 0 & \text{otherwise} \end{cases} \quad \llbracket \bigwedge S \rrbracket_{\mathsf{T}} \sigma = \begin{cases} 1 & \text{if } \llbracket \phi \rrbracket_{\mathsf{T}} \sigma = 1 \text{ for any } \phi \in S \\ 0 & \text{otherwise.} \end{cases}$$

The other cases are defined similarly. When it is clear from the context, we write $\llbracket \cdot \rrbracket$ for $\llbracket \cdot \rrbracket_{\mathsf{T}}$. Given $V \in \wp_f(V)$ and $\phi_1, \phi_2 \in \Phi_{\Delta,V}$, we define $\phi_1 \leq_{\mathsf{T},V} \phi_2$ if and only if for every $\theta \in \Theta_V$ we have that $\llbracket \phi_1 \rrbracket_{\mathsf{T}} \theta = 1$ entails $\llbracket \phi_2 \rrbracket_{\mathsf{T}} = 1$. When it is clear from the context, we write $\leq$ for $\leq_{\mathsf{T},V}$. We define $\phi_1 \equiv_{\mathsf{T},V} \phi_2$ if and only if $\phi_1 \leq_{\mathsf{T},V} \phi_2$ and $\phi_2 \leq_{\mathsf{T},V} \phi_1$. This equivalence is called (T, V) -equivalence. Again, we drop the subscripts T and V when it is clear from the context.

From now on, if $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ is a type domain and $V \in \wp_f(\mathcal{V})$, every transfinite formula in T_V will stand for its $(\mathbb{T}, V)$ -equivalence class. Note that, for every $V \in \wp_f(\mathcal{V})$, T_V is a complete lattice w.r.t. $\leq_{\mathbb{T}, V}$, since by Definition 7 it is completely $\wedge$ -closed and topped (since $true = \wedge \emptyset$). We must check that the operations $\star^{T_V}$ and $\exists^{T_V}$ of Definition 7 are independent from the representants chosen for the $(\mathbb{T}, V)$ -equivalence classes. This is obvious for conjunction. For restriction, it is true if the following property holds.

P1: Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type domain for $\langle \Delta, \Sigma, I \rangle$. For every $V \in \wp_f(\mathcal{V})$ and $\{\phi_1, \phi_2\} \subseteq T_V$, if $\phi_1 \equiv \phi_2$ then $\phi_1[P/x] \equiv \phi_2[P/x]$ for any $x \in V$ and $P = \{d \in \text{terms}(\Delta, \emptyset) \mid t \in \llbracket d \rrbracket I\}$ with $t \in \text{terms}(\Sigma, V)$.

For any $V \in \wp_f(\mathcal{V})$, a Moore family of $Down_V$ (i.e., an abstract interpretation of $Down_V$) can be defined once a type domain is given.

Definition 9. Given a type domain $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\langle \Delta, \Sigma, I \rangle$ and $\phi \in T_V$, we define $\gamma_{T_V}(\phi) = \{\theta \in \Theta_V \mid \text{for all } \sigma \in \Theta_V \text{ s.t. } \sigma \leq \theta \text{ we have } \llbracket \phi \rrbracket_{\mathbb{T}} \sigma = 1\}$. Note that $\gamma_{T_V}(\phi) \in Down_V$ for any $\phi \in T_V$. Given $\phi_1, \phi_2 \in T_V$, we define $\phi_1 \equiv_{\gamma_{T_V}} \phi_2$ if and only if $\gamma_{T_V}(\phi_1) = \gamma_{T_V}(\phi_2)$. This equivalence relation will be called γ_{T_V} -equivalence. When it is clear from the context, we drop the subscript T_V from γ_{T_V} .

It is easy to realise that $\equiv$ entails $\equiv_{\gamma}$, though the converse does not hold.

Example 4. Consider the type system $\mathbf{G}$ of Example 1 and the type domain $\{\Phi_{\Delta, V}\}_{V \in \wp_f(\mathcal{V})}$ for $\mathbf{G}$. Consider V such that $\{x, y\} \subseteq V$. Let $\phi_1 = false$ and $\phi_2 = x \in \mathbf{g} \Rightarrow false$. Note that there is no θ such that $\llbracket \phi_1 \rrbracket_{\mathbf{G}} \theta = 1$. But any substitution θ such that $\theta(x)$ is not ground satisfies $\llbracket \phi_2 \rrbracket_{\mathbf{G}} \theta = 1$. Hence ϕ_1 and ϕ_2 are not $(\mathbf{G}, V)$ -equivalent. However, $\gamma(\phi_1) = \emptyset$ and $\gamma(\phi_2) = \emptyset$. This is obvious for ϕ_1 . For ϕ_2 , assume $\theta \in \gamma(\phi_2)$. Any instance of θ must belong to $\gamma(\phi_2)$. But this is a contradiction as soon as we consider an instance θ' of θ which makes x ground, because in such a case it would be $\llbracket false \rrbracket_{\mathbf{G}} \theta' = 1$.

Proposition 7. Given a type domain $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for $\mathbb{T}$, γ_{T_V} is co-additive and $\gamma_{T_V}(T_V)$ is a Moore family of $Down_V$ for any $V \in \wp_f(\mathcal{V})$.

Since γ is co-additive and $Down_V$ and T_V are complete lattices, we conclude that γ is the concretisation map of a Galois connection from $Down_V$ into T_V . Let α (i.e., α_{T_V}) be the corresponding abstraction map. Since, in general, γ is not one-to-one (Example 4), we are not guaranteed to have a Galois insertion instead of just a Galois connection. This is what the property below requires.

P2: Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a domain for $\mathbb{T}$. For any $V \in \wp_f(\mathcal{V})$ and $\phi_1, \phi_2 \in T_V$, if $\phi_1 \equiv_{\gamma} \phi_2$ then $\phi_1 \equiv \phi_2$.

Definition 10. Given a type system $\mathbb{T} = \langle \{c^0\}, \Sigma, I \rangle$ and $V = \{x_1, \dots, x_n\} \in \wp_f(\mathcal{V})$, $\phi \in \Phi_{\Delta, V}$ is called positive if and only if for any ground term $t \in \text{terms}(\Sigma, \emptyset)$ such that $t \in \llbracket c \rrbracket I$, we have $\llbracket \phi \rrbracket \{x_1 \mapsto t, \dots, x_n \mapsto t\} = 1$.

Definition 11. A type domain $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\langle \Delta, \Sigma, I \rangle$ is positive if Δ is formed by just one constant, i.e., $\Delta = \{c^0\}$ for some c , and for every $V \in \wp_f(\mathcal{V})$ and every $\phi \in T_V$, ϕ is positive (Definition 10).

A type domain $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ for the type system $\mathbf{T} = \langle \Delta, \Sigma, I \rangle$ is structural if and only if $\mathbf{T}$ is structural, i.e., if and only if for every $V \in \wp_f(\mathcal{V})$ and $t_1, \dots, t_n \in \text{terms}(\Sigma, V)$ there exists $\sigma \in \Theta_V^V$ such that, for every $i = 1, \dots, n$, the term $t_i \sigma$ is ground and for every $d \in \text{terms}(\Delta, \emptyset)$ we have $t_i \in \llbracket d \rrbracket I$ if and only if $t_i \sigma \in \llbracket d \rrbracket I$.

In a structural type domain, every finite set of terms can be instantiated into a finite set of ground terms with the same type properties. The following proposition shows the importance of positive or structural type domains.

Proposition 8. Every positive or structural type domain satisfies both properties P1 and P2.

Proposition 9. Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type domain for the type system $\langle \Delta, \Sigma, I \rangle$, satisfying property P1.

- i) The operator $\star^{T_V}$ is always correct w.r.t. $\star^{Down_V}$, and it is its best possible approximation if property P2 holds for T .
- ii) The operator $\exists^{T_V}$ is always correct w.r.t. $\exists^{Down_V}$, and it is its best possible approximation if T is positive or structural.
- iii) The element $\delta_{x,y}^{T_V}$ is a correct approximation of $\delta_{x,y}^{Down_V}$, and it is its best possible approximation if T is positive or structural.

In conclusion, a positive or structural type domain enjoys interesting properties. Namely, restriction is well-defined (property P1), a Galois insertion can be established between $Down_V$ and $Type_V$ (property P2), the operators over $Type_V$ are the best possible approximations of the corresponding operators over $Down_V$ and the diagonal elements of $Type_V$ are exactly the abstraction of the diagonal elements of $Down_V$ (Proposition 9).

It is not easy to apply the definition of structural type domain (Definition 11). Therefore, we provide a sufficient condition which entails that a type domain is structural, and shows how large the class of structural type domains is.

Proposition 10. Let $T = \{T_V\}_{V \in \wp_f(\mathcal{V})}$ be a type domain for the type system $\langle \Delta, \Sigma, I \rangle$. Assume there exists a ground term $t \in \text{terms}(\Sigma, \emptyset)$ such that for every $d \in \text{terms}(\Delta, \emptyset)$ and for all $t' \in \llbracket d \rrbracket I$, every term obtained from t' by substituting some occurrences of t with variables of $\mathcal{V}$ is still in $\llbracket d \rrbracket I$. Then T is structural.

6 Applications

Example 5. Consider the type system of Example 1 and the type domain $Pos_{\{g\}} = \{Pos_{\{g\}, V}\}_{V \in \wp_f(\mathcal{V})}$ where $Pos_{\{g\}, V}$ is formed by the set of positive transfinite formulas in $\Phi_{\{g\}, V}$. Note that this set is finite. In [1] it is shown that $Pos_{\{g\}, V}$

is closed under conjunction and restriction and contains the diagonal elements. This type domain is well-known [1,5,6].

A formula is called definite if the set of its propositional models is closed under instantiation. Consider the type domain $Def_{\{g\}} = \{Def_{\{g\},V}\}_{V \in \wp_f(V)}$ where $Def_{\{g\},V}$ is the set of (positive) definite formulas in $\Phi_{\{g\},V}$. Even this type domain is well-known [1]. In [1] it is shown that the set of definite formulas is closed under conjunction and restriction and contains the diagonal elements.

Since $Pos_{\{g\}}$ and $Def_{\{g\}}$ are positive, we conclude that they are related to *Down* through a Galois insertion and that the operations and diagonal elements of Definition 7 are the best possible approximations of the corresponding operations and diagonal elements of *Down* (Proposition 9).

Example 6. The same construction done above for groundness can be applied to non-freeness (Example 2). The resulting type domains will be called $Pos_{\{nf\}}$ and $Def_{\{nf\}}$. Note that they are positive type domains.

Since Proposition 9 considers every positive type domain, we have generalised the result contained in [6] for the case of groundness. Therefore, we can use many *incarnations* of positive type analyses, combining them with a reduced product operation and using the operations and diagonal elements of Definition 7 as done in [3], where, however, the authors did not provide any justification of the correctness of this approach.

The following example shows that we can apply our theory to more complicated cases than those considered above, dealing with many types, dependencies among different types and polymorphism.

Example 7. Given the type system of Example 3, consider the type domain $ILT = \{ILT_V\}_{V \in \wp_f(V)}$ where $ILT_V = \Phi_{\Delta,V}$. Since Proposition 10 holds with $t = \mathbf{s}(\square)$, we conclude that *ILT* is structural, is related to *Down* through a Galois insertion and the operators and diagonal elements of Definition 7 are optimal w.r.t. the corresponding operators and diagonal elements of *Down*.

For groundness and non-freeness we considered only positive formulas (Examples 5 and 6), while we considered the whole set of formulas for *ILT*. Indeed, in the case of groundness or non-freeness, a variable can always eventually belong to the type, and a formula like $\neg(x \in g)$ has an empty concretisation. Instead, the formula $\neg(x \in \mathbf{int})$ has a clear meaning. Namely, it says that x is not and can never become an integer. For instance, it might be bound to a list.

Note that ILT_V is not a finite set, although V is finite, since we can write arbitrarily complex types, and the definition of the restriction operator (Definition 7) is not effective. In [12] it is shown how to overcome these problems using a domain of type dependencies with type variables which is an abstract interpretation of our type domains of transfinite formulas.

7 Type Systems and Linear Refinement

We have shown that every type domain induces an abstract interpretation of *Down* (Proposition 7). In this section, we want to show how a hierarchy of abstract interpretations of *Down* can be defined starting from a basic one modelling just the type properties of interest. Every domain in this hierarchy will be the linear refinement of the previous one w.r.t. concrete conjunction. This means that our hierarchy will be a chain of domains which induce an approximation of concrete conjunction which is more and more precise as long as we refine. This generalises an analogous result about groundness analysis shown in [15].

Definition 12. Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a type system and $V \in \wp_f(\mathcal{V})$. We define $\mathbf{v}_t = \{\theta \in \Theta_V \mid \theta(v) \in \llbracket t \rrbracket I\}$ with $v \in V$ and $t \in \mathbf{terms}(\Delta, \emptyset)$, $\mathsf{Basic}_{\mathsf{T},V} = \bigwedge \{\mathbf{v}_t \mid v \in V \text{ and } t \in \mathbf{terms}(\Delta, \emptyset)\}$, $\mathsf{Basic}_{\mathsf{T},V}^{i+1} = \mathsf{Basic}_{\mathsf{T},V}^i \rightarrow^{\star \mathsf{Down}_V} \mathsf{Basic}_{\mathsf{T},V}^i$ for $i \geq 0$.

We show now what is $\rightarrow^{\mathsf{Down}_V}$ for downward closed properties and that $\mathsf{Basic}_{\mathsf{T},V}^{i+1}$ contains $\mathsf{Basic}_{\mathsf{T},V}^i$, which is why we used Equation (1) in Definition 12.

Proposition 11. Let $V \in \wp_f(\mathcal{V})$.

- i) If $\{d_1, d_2\} \subseteq \mathsf{Down}_V$, $d_1 \rightarrow^{\star \mathsf{Down}_V} d_2 = \{\theta \in \Theta_V \mid \text{for all } \sigma \leq \theta \text{ if } \sigma \in d_1 \text{ then } \sigma \in d_2\}$.
- ii) Given a type system T , $\mathsf{Basic}_{\mathsf{T},V}^i \subseteq \mathsf{Basic}_{\mathsf{T},V}^{i+1}$ for any $i \geq 0$.

For a large class of type systems, a *best* domain exists.

Proposition 12. Let $\mathsf{T} = \langle \Delta, \Sigma, I \rangle$ be a structural type system. For any $V \in \wp_f(\mathcal{V})$, $\mathsf{Basic}_{\mathsf{T},V}$ is condensing, $\mathsf{Basic}_{\mathsf{T},V}^2 = \mathsf{Basic}_{\mathsf{T},V}^i$ for $i \geq 2$ and $\mathsf{Basic}_{\mathsf{T},V}^2 = (\mathsf{Basic}_{\mathsf{T},V} \rightarrow^{\mathsf{Down}_V} \mathsf{Basic}_{\mathsf{T},V}) \rightarrow^{\mathsf{Down}_V} \mathsf{Basic}_{\mathsf{T},V}$.

Note that we do not know if the analogous of Proposition 12 holds for the type system of a positive type domain (Definition 11). However, in [15] it is shown that Proposition 12 holds for the type system for groundness of Example 1, and that $\mathsf{Basic}_{\mathsf{G},V}$ is isomorphic to $\mathsf{Pos}_{\{\mathbf{g}\},V}$ (Example 5). The proofs of [15] hold for the type system for non-freeness too (Example 2). However, they cannot be generalised to every type system of a positive type domain.

The domains $\mathsf{Basic}_{\mathsf{T},V}$, $\mathsf{Basic}_{\mathsf{T},V}^1$ and $\mathsf{Basic}_{\mathsf{T},V}^2$ can be represented easily.

Definition 13. Let $\langle \Delta, \Sigma, I \rangle$ be a type system. Let $V \in \wp_f(\mathcal{V})$. We define $\mathsf{And}_{\Delta,V} = \{\wedge S \mid S \subseteq \{v \in t \mid v \in V \text{ and } t \in \mathbf{terms}(\Delta, \emptyset)\}\}$, $\mathsf{Or}_{\Delta,V} = \{\vee S \mid S \subseteq \{v \in t \mid v \in V \text{ and } t \in \mathbf{terms}(\Delta, \emptyset)\} \text{ and } S \neq \emptyset\}$, $\mathsf{Def}_{\Delta,V} = \{A_1 \Rightarrow A_2 \mid \{A_1, A_2\} \subseteq \mathsf{And}_{\Delta,V}\}$ and $\mathsf{Pos}_{\Delta,V} = \{A \Rightarrow O \mid A \in \mathsf{And}_{\Delta,V} \text{ and } O \in \mathsf{Or}_{\Delta,V}\}$.

Proposition 13. Given a positive or structural type domain T for $\langle \Delta, \Sigma, I \rangle$, $\mathsf{Def}_{\Delta,V}$ is isomorphic to $\mathsf{Basic}_{\Delta,V}^1$ and $\mathsf{Pos}_{\Delta,V}$ is isomorphic to $\mathsf{Basic}_{\Delta,V}^2$.

Therefore, for structural type domains $\mathsf{Pos}_{\Delta,V}$ allows to approximate the concrete conjunction in the best possible way, among the abstract domains which do not consider the name of the functors (Proposition 12).

8 Conclusions

We have defined a large class of type domains that enjoy the same desirable properties of the well-known domain for groundness analysis [1,5,6]. This leads to the use of transfinite formulas and operators on transfinite formulas for the type analysis of logic programs. In [12] the authors tackle the problem of the finiteness of the analysis. Finiteness is achieved through the use of *type variables*, which allow one to represent infinite conjunctions by using a finite object. The resulting domains are made of logic programs, and the abstract operations are operations over logic programs.

It would be interesting to know if the condition of being positive or structural, which entails all the desirable properties of a type domain, is necessary or, at least, can be further weakened.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. G. Birkhoff. Lattice Theory. In *AMS Colloquium Publication*, third ed., 1967.
3. M. Codish and B. Demoen. Deriving Polymorphic Type Dependencies for Logic Programs Using Multiple Incarnations of Prop. In *Proc. of the first International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 281–296. Springer-Verlag, 1994.
4. M. Codish and V. Lagoon. Type Dependencies for Logic Programs Using ACI-Unification. In *Proceedings of the 1996 Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, June 1996. Extended version to appear in *Theoretical Computer Science*.
5. A. Cortesi, G. Filè, and W. Winsborough. Prop Revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.
6. A. Cortesi, G. Filè, and W. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *Journal of Logic Programming*, 27(2):137–167, 1996.
7. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. 6th ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
8. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
9. N.J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
10. G. Filè, R. Giacobazzi, and F. Ranzato. A Unifying View on Abstract Domain Design. *ACM Computing Surveys*, 28(2):333–336, 1996.
11. R. Giacobazzi, F. Ranzato, and F. Scozzari. Building Complete Abstract Interpretations in a Linear Logic-based Setting. In G. Levi, editor, *Static Analysis, Proceedings of the 5th International Static Analysis Symposium SAS 98*, volume 1503 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 1998.
12. Giorgio Levi and Fausto Spoto. An Experiment in Domain Refinement: Type Domains and Type Representations for Logic Programs. In Catuscia Palamidessi,

- Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 152–169, Pisa, Italy, September 1998. ©Springer-Verlag.
13. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.
14. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
15. F. Scozzari. Logical Optimality of Groundness Analysis. In P. Van Hentenryck, editor, *Proceedings of the 4th International Static Analysis Symposium SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1997.
16. Jan-Georg Smaus, Patricia Hill, and Andy King. Mode Analysis for Typed Logic Programs. In *Proc. of the LOPSTR'99 Workshop*, pages 163–170, Venice, Italy, September 1999.

Invited Talk: ASM Formalware in the Software Engineering Cycle

Yuri Gurevich

Microsoft and University of Michigan

Abstract. Software is becoming more and more complex. As you move down the product cycle – from market requirements to product definition, specification, design, implementation, documentation, testing, and support — the cost of error correction escalates dramatically. We aim to improve on the various stages by means of high-level yet executable models. This is based on the abstract state machine approach <http://www.eecs.umich.edu/gasm/>.

Process Calculi for Coordination: From Linda to JavaSpaces*

Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
`busi,gorrieri,zavattar@cs.unibo.it`

Abstract. We present a collection of process calculi featuring coordination primitives for the shared dataspace coordination model (inspired by Linda, JavaSpaces and TSpaces), some of which have never been formally defined before. The operational semantics of the calculi is used to clarify possible ambiguities of the informal definitions of these languages, to discuss possible implementation choices, to compare the expressive power of the new primitives and, finally, to support formal reasoning about programs written with these primitives.

1 Introduction

Coordination middlewares are emerging as suitable architectures for making easier the programming of distributed applications. JavaSpaces [12] and TSpaces [14], produced by Sun and IBM respectively, are the most prominent proposals; they are both based on the shared dataspace coordination architecture, originally proposed in Linda (see, e.g., [6]). The basic idea behind Linda is the so-called *generative communication*; its main features are the following:

- *Asynchronous communication*: it is realized by means of a (conceptually shared) communication medium (called tuple space) that is the actual place where all messages/tuples are delivered and extracted by the processes. A sender may proceed just after performing the insertion of the tuple in the tuple space, while the receiver can remove the tuple at any time after that tuple is in the tuple space. Hence, the asynchronous communication between the sender and the receiver is realized by means of two synchronous operations with the tuple space.
- *Read operation*: a process can check the presence of some tuple, without removing it from the space.
- *Conditional input/read predicates*: these are non-blocking variants of the remove and read operations; if the required message is absent, the process is not blocked and continues with a different alternative.

More recent extensions of the Linda paradigm (e.g., JavaSpaces and TSpaces) include some new primitives useful for coordination of complex applications in open, large, distributed systems. In this paper we will focus on some of them:

* Work partially supported by Esprit working group n.24512 “Coordina”

- *Event notification.* Besides the data-driven coordination typical of Linda, it may be very useful to include in a language an event-driven mechanism of process activation. A process can register its interest in future arrivals of some objects and then receive communication of each occurrence of this event.
- *Blocking operations with timeouts.* The operations of removal or reading of an object can be weakened by expressing the deadline beyond which the operation fails.
- *Guaranteed duration of service.* An object inserted in the dataspace as well as the interest in an event notification need not to hold forever; in many cases it is useful that the language has the capability to declare time bounds on these operations, and even better to re-negotiate such bounds if needed.

Despite of the clear relevance of such primitives for coordination middlewares, very little has been done to define formally their semantics. One may think that formalizing the intended semantics of these primitives is superfluous, as their semantics could appear obvious. Unfortunately, this is not the case. In places, the informal definition of these primitives in the available documentation is quite ambiguous (e.g., variants of the same primitives exist in different languages and even in the same language); this may have the effect of giving too much freedom in the implementation choices, hence producing semantically different implementations. Moreover, awareness of the expressiveness capabilities of the various primitives is often lacking, as well as methods for reasoning about programs written with these primitives.

The standard approach to solve the problems above is to give a formal semantics to the coordination language of interest. Such a semantics would fix the actual interpretation of the primitives, can be a precise guide for the implementor as well as a tool for reasoning about language expressiveness and program properties.

In this paper we initiate an investigation about the semantics of languages like JavaSpaces and TSpaces, by abstracting the coordination primitives away from the concrete language. To this aim, we use the mathematical machinery developed by the process algebra community (see, e.g., [10,8]) that seems flexible enough to be useful for our aims. We start our investigation by presenting a kernel process calculus, comprising the coordination primitives of Linda, inspired by a previous proposal studied in [1]. We then enrich it incrementally by adding in isolation each of the primitives sketched above: event notification, timeouts on blocking operations, leasing for timing service requests. For each primitive we discuss possible interpretations that may offer rather different expressive power to the language or may be suitable for different implementations. Some examples are reported to illustrate the flexibility of the newly introduced primitives. Future research regarding formal analysis of programs are addressed in the conclusive section.

(1) $\langle a \rangle \xrightarrow{\bar{a}} \mathbf{0}$	(2) $write(a).P \xrightarrow{\tau} \langle a \rangle P$
(3) $take(a).P \xrightarrow{a} P$	(4) $read(a).P \xrightarrow{a} P$
(5) $take\exists(a)?P\_Q \xrightarrow{a} P$	(6) $take\exists(a)?P\_Q \xrightarrow{\neg a} Q$
(7) $read\exists(a)?P\_Q \xrightarrow{a} P$	(8) $read\exists(a)?P\_Q \xrightarrow{\neg a} Q$
(9) $\frac{P \xrightarrow{\bar{a}} P' \quad Q \xrightarrow{a} Q'}{P Q \xrightarrow{\tau} P' Q'}$	(10) $\frac{P \xrightarrow{\bar{a}} P' \quad Q \xrightarrow{a} Q'}{P Q \xrightarrow{\tau} P' Q'}$
(11) $\frac{P \xrightarrow{\alpha} P'}{P Q \xrightarrow{\alpha} P' Q} \quad \alpha \neq \neg a, \bar{a}, \dot{a}, \check{a}$	(12) $\frac{P \xrightarrow{\neg a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P Q \xrightarrow{\neg a} P' Q}$
(13) $\frac{P \xrightarrow{\alpha} P' \quad K = P}{K \xrightarrow{\alpha} P'}$	

Table 1. Operational semantics (symmetric rules omitted).

2 The Kernel Language

In this section we introduce the syntax and the operational semantics of a calculus comprising the basic Linda-like coordination primitives. It is a small variant of a previous calculus formerly presented in [1].

Let *Name* be a denumerable set of object types ranged over by $a, b, \dots$, and *Const* be a set of program constants ranged over by $K, K', \dots$

Let *Conf* be the set of the possible configurations defined by the following grammar:

$$\begin{aligned}
 P &::= \langle a \rangle \mid C \mid P|P \\
 C &::= \mathbf{0} \mid \mu.C \mid \eta?C\_C \mid C|C \mid K
 \end{aligned}$$

where:

$$\begin{aligned}
 \mu &::= write(a) \mid read(a) \mid take(a) \\
 \eta &::= read\exists(a) \mid take\exists(a)
 \end{aligned}$$

Programs are represented by terms C containing the coordination primitives; the dataspace is modeled by representing each of its objects a with a term $\langle a \rangle$. A configuration is composed of some programs and some available objects composed in parallel using the composition operator $|$. A program can be a terminated program $\mathbf{0}$ (which is usually omitted for the sake of simplicity), a prefix form $\mu.P$, an *if-then-else* form $\eta?P\_Q$, the parallel composition of subprograms, or a program constant K .

A prefix μ can be one of the primitives $write(a)$, which introduces a new object $\langle a \rangle$ inside the data repository, $read(a)$, which tests for the presence of an instance of object $\langle a \rangle$, and $take(a)$, which consumes an instance of object $\langle a \rangle$.

The last two primitives are blocking, in the sense that a program performing one of them cannot proceed until the operation is successfully completed.

The guards of the if-then-else forms are $read\exists(a)$ and $take\exists(a)$, which are the non-blocking variants of $read(a)$ and $take(a)$, respectively. The notation is inspired by the similar operations *readIfExists* and *takeIfExists* of JavaSpaces [12]. Their behaviour is represented by using terms having two possible continuations, e.g., $read\exists(a)?P\_Q$. The first continuation P is chosen if the requested object is available in the data repository; in this case the non-blocking operations behave exactly as the corresponding blocking ones. On the other hand, if no instance of the requested object is actually available, the second continuation Q is chosen and the data repository is left unchanged.

Constants are used to permit the definition of programs with infinite behaviours. We assume that each constant K is equipped with one and only one definition $K = C$; as usual we assume also that only guarded recursion is used [10].

The semantics of the language is described via a labeled transition system $(Conf, Label, \longrightarrow)$ where $Label = \{\tau\} \cup \{a, \underline{a}, \bar{a}, \neg a \mid a \in Name\}$ (ranged over by $\alpha, \beta, \dots$) is the set of the possible labels. The labeled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1. For the sake of simplicity we have omitted the symmetric rules of (9) – (12).

Axiom (1) indicates that $\langle a \rangle$ is able to give its contents to the environment by performing an action labeled with $\bar{a}$. Axiom (2) describes the output operation: in one step a new object is produced. Axiom (3) associates to the action performed by the prefix $in(a)$ a label a , the complementary of $\bar{a}$, while axiom (4) associates to a $read(a)$ prefix a label $\underline{a}$.

Axioms (5) and (6) describe the semantics of $take\exists(a)?P\_Q$: if the required $\langle a \rangle$ is present it can be consumed (axiom (5)), otherwise, in the case $\langle a \rangle$ is not available, its absence is guessed by performing an action labeled with $\neg a$ (axiom (6)). Axioms (7) and (8) are the corresponding axioms for the $read\exists(a)$ operator; the unique difference is that the label $\underline{a}$ is used instead of a .

Rule (9) is the usual synchronization rule; while rule (10) deals with the new label $\underline{a}$ representing a non-consuming operation: in this case the term performing the output operation (labeled with $\bar{a}$) is left unchanged as the $read$ operation does not consume the tested object.

Rule (11) is the usual local rule, which is valid only for labels different from $\neg a$; indeed, an action of this kind can be performed only if no object $\langle a \rangle$ is available in the data repository, i.e., no actions labeled with $\bar{a}$ can be performed by the terms in the environment (rule (12)). The side condition of rule (11) considers also labels that will be defined in the following sections.

The last rule (13) allows a program constant K defined by $K = C$ to perform the same actions permitted to C .

Note that rule (12) uses a negative premise; however, the operational semantics is well defined, because the transition system specification is strictly stratifiable [7], condition that ensures (as proved in [7]) the existence of a unique transition system agreeing with it.

Alternative Semantics. The semantics we have defined assumes that the programs are synchronous with the dataspace, in the sense that the emitted object is surely already available inside the data repository at the moment the *write* operation terminates (see rule (2)). In a previous paper on the semantics of Linda [2], this kind of semantics is called *ordered* and alternative interpretations are presented, among which the *unordered* one which assumes that programs are asynchronous with the dataspace: the emitted object becomes available after an unpredictable delay. The paper [3] shows the existence of an expressiveness gap between these two alternatives. We have chosen the ordered interpretation as it is the semantics adopted by the actual JavaSpaces specifications, as indicated in the sections 2.3 and 2.8 of [12], and also confirmed us by personal communications with John McClain of Sun Microsystems Inc. [9].

3 Event Notification

In this section we extend the previous calculus with an event notification mechanism inspired by the *notify* primitive of JavaSpaces [12].

The syntax of the kernel language is simply extended by permitting a new prefix:

$$\mu ::= \dots \mid \text{notify}(a, C)$$

A program $\text{notify}(a, C).P$ can register its interest in the future incoming arrivals of the data of kind a , and then receive communication of each occurrence of this event. This behaviour can be modeled by introducing a new term $\text{on}(a, C)$ (that we add to the syntax as an auxiliary term) which is a listener that spawns an instance of program C every time a new object $\langle a \rangle$ is introduced in the dataspace. This is modeled by extending the possible labels with the set $\{\dot{a}, \vec{a} \mid a \in \text{Name}\}$,

$$\begin{array}{ll}
 (2') & \text{write}(a).P \xrightarrow{\vec{a}} \langle a \rangle | P \\
 (14) & \text{notify}(a, Q).P \xrightarrow{\tau} \text{on}(a, Q) | P \quad (15) \quad \text{on}(a, P) \xrightarrow{\dot{a}} P | \text{on}(a, P) \\
 (16) & \frac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P | Q \xrightarrow{\dot{a}} P' | Q'} \quad (17) \quad \frac{P \xrightarrow{\dot{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P | Q \xrightarrow{\dot{a}} P' | Q} \\
 (18) & \frac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\vec{a}} Q'}{P | Q \xrightarrow{\vec{a}} P' | Q'} \quad (19) \quad \frac{P \xrightarrow{\vec{a}} P' \quad Q \not\xrightarrow{\vec{a}}}{P | Q \xrightarrow{\vec{a}} P' | Q}
 \end{array}$$

Table 2. Operational semantics of *notify* (symmetric rules omitted).

by adding the rules in Table 2 (we omit the symmetric rules of (17) – (19)) to the ones in Table 1, and by substituting the rule (2') for the rule (2). Negative

premises are used in the new rules (see rules (17) and (19)), but the transition system specification is still stratifiable; thus, the operational semantics is well defined.

The new labels $\bar{a}$ and $\dot{a}$ represent the occurrence and the observation of the event “creation of a new object of kind a ”, respectively. This event happens when an output operation is executed; for this reason we change the label associated to the prefix $write(a)$ from τ to $\bar{a}$ (see the new rule (2')). Axiom (14) indicates that the $notify(a, P)$ prefix produces a new instance of the term $on(a, P)$. This term has the ability to spawn a new instance of P every time a new $\langle a \rangle$ is produced; this behaviour is described in axiom (15) where the label $\dot{a}$ is used to describe this kind of computation step.

Rules (16) and (17) consider actions labelled with $\dot{a}$ indicating the interest in the incoming instances of $\langle a \rangle$. If one program able to perform this kind of action is composed in parallel with another one registered for the same event, then their local actions are combined in a global one (rule (16)); otherwise, the program performs its own action locally (rule (17)). Rules (18) and (19) deal with two different cases regarding the label $\bar{a}$ indicating the arrival of a new instance of $\langle a \rangle$: if there are terms waiting for the notification of this event are present in the environment, then they are woken-up (rule (18)); otherwise, the environment is left unchanged (rule (19)).

Example of a Client/Server System. As an example of use of the *notify* primitive, we consider a simple client/server system; clients ask for services producing objects $\langle request \rangle$, and the server consumes these objects and produces a reply $\langle service \rangle$. The complete system is defined as follows:

$$\begin{aligned} Clients \& Server &= notify(request, Server).(Client_1 \mid \dots \mid Client_n) \\ Client_i &= write(request).take(service) \\ Server &= take(request).write(service) \end{aligned}$$

As its first operation, the server is produced by means of the execution of a $notify(request, Server)$ operation. Afterward, several clients are spawn. Each of these clients simply produces a request and then waits for a service. Each time a $\langle request \rangle$ is produced, this activates a new *Server* exploiting the event notification mechanism. The server consumes the object $\langle request \rangle$ and produces a corresponding object $\langle service \rangle$.

Observe that the notification mechanism allows the concurrent execution of more than one server; this happens if a request is produced before the previous one is served. According to this observation, we can say that we model a multithreaded server.

Alternative Semantics. Observe that the synchronization between the action $\bar{a}$ and $\dot{a}$ is modeled as a synchronous multicast; when an action $\bar{a}$ occurs, it synchronizes with all the possible actions $\dot{a}$ that the environment may perform. The fact that all the interested listener are involved in the synchronization is

ensured by the negative premises of the rules (17) and (19) and by the side condition $\alpha \neq \dot{a}$ of the locality rule (11). In this way, we model a reliable service. On the other hand, the event notification mechanism is sometimes a best-effort service such as in JavaSpaces [12], where some listeners may be not notified of the occurrence of an event in which they have registered their interest.

We can model a best-effort event notification service simply by removing the negative premise of the rules (17) and (19).

Expressiveness of Notify. Regarding the expressiveness of the *notify* we first observe that it introduces an implicit form of recursion, then we address the problem of implementing the new operations using only primitives of the kernel language.

The *notify* primitive permits us to describe processes with an infinite behaviour without being forced to use recursive definitions. This because the listener $on(a, P)$, which is the result of a $notify(a, P)$ operation, is able to repeatedly produce a new program P every time the corresponding event occurs. For example, this mechanism can be exploited to simulate the behaviour of a term which recursively renames all objects $\langle a \rangle$ in $\langle b \rangle$ using the following non-recursive definition:

$$\begin{aligned} Trans &= notify(rec, Rename).write(rec) \\ Rename &= take(rec).take(a).write(b).write(rec) \end{aligned}$$

where the name *rec* is used to repeatedly activate an instance of a one-shot renaming program. In the kernel calculus introduced in the previous section, this kind of behaviour can be defined only by exploiting a recursive definition.

Another interesting question on the expressiveness of the *notify* primitive regards the possibility to encode it in terms of the coordination primitives of the kernel language, only. For example, one could think to simulate the above multithreaded server without using the event notification mechanism. The answer to this question has been given in [5], where it is shown how to encode the *notify* primitive in a language similar to the kernel calculus of the previous section. Thus, we can conclude that a similar client/server system can be obtained also without using the event notification mechanism. However, the proposed encoding is rather complex; every time a new object is created, the sender must perform a protocol composed of three phases: first it is necessary to count the number of registered listener, then to communicate to each of them the happening of an interesting event, and finally to wait for the total amount of acknowledgements. Moreover, this protocol must be executed in mutual exclusion. For this reason, we can conclude that even if the *notify* primitive does not strictly increase the expressive power of the language, it is very useful for an easier, more direct, and more natural definition of interesting coordination patterns, such as the one adopted by the above multithreaded server. The encoding of *notify* presented in [5], strictly requires the test for absence incorporated in the $take\exists$ (or $read\exists$). This allows to conclude that *notify* strictly increases the expressive power of a language including only the *write*, *read*, and *take* basic operations.

In a related paper [4] we have investigated the interplay between the ordered and unordered interpretations of *write* (that we have described in the previous section) with the *notify* primitive. An interesting result is that *notify* permits us to encode the ordered approach on top of the unordered one.

4 Timeout

In this section we consider the problem of modeling operations equipped with timeouts, which are used as extra parameters for the blocking operations *read* and *take* in order to indicate a maximum amount of time during which the presence of the requested object is investigated. If no instance is found before the end of this period, the operation fails and terminates.

$$\begin{array}{ll}
 (3') \quad take(a, t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q & (4') \quad read(a, t)?P\_Q \xrightarrow{\tau} read(a)_t?P\_Q \\
 (20) \quad take(a)_t?P\_Q \xrightarrow{a} P & (21) \quad read(a)_t?P\_Q \xrightarrow{a} P \\
 (22) \quad \eta_t?P\_Q \xrightarrow{\checkmark} \eta_{t-1}?P\_Q \quad \text{if } t \neq 0 & (23) \quad \eta_0?P\_Q \xrightarrow{\checkmark} Q \\
 (24) \quad \frac{P \xrightarrow{\checkmark} P' \quad Q \xrightarrow{\checkmark} Q'}{P|Q \xrightarrow{\checkmark} P'|Q'} & (25) \quad \frac{P \xrightarrow{\checkmark} P' \quad Q \not\xrightarrow{\checkmark}}{P|Q \xrightarrow{\checkmark} P'|Q}
 \end{array}$$

Table 3. Operational semantics for timeouts (symmetric rule of (25) omitted).

In order to model this kind of behaviour, we change the primitives *read*(*a*) and *take*(*a*) with the new *read*(*a*, *t*) and *take*(*a*, *t*) operations (where *t* defines the timeout period). These operations are no more used as prefixes of terms of the kind $\mu.P$, but as guards of if-then-else forms $\eta?P\_Q$. In this way, we can describe both the program *P*, chosen if the operation succeeds, and the program *Q*, activated if, on the contrary, the operation fails.

The way we represent time is inspired by JavaSpaces, where the current time is represented by an integer which is incremented each millisecond. If the execution of an operation with timeout period *t* is scheduled when the current time is *c*, the operation fails at the end of the interval with current time $c + t$.

In our process calculus we do not use any value to represent the current time, but we only consider the passing of time, which is considered as divided into basic discrete intervals. Operationally, we model the instant in which an interval finishes, and the subsequent start, by means of a transition labelled with $\checkmark$. More precisely, we use $P \xrightarrow{\checkmark} P'$ to state that the term *P* becomes *P'* due to the fact that the current interval has finished, and the subsequent just started.

In our process calculus we represent two kinds of terms: those sensible to the passing of time and those which are not. In the first case, the terms have outgoing transitions labelled with $\checkmark$, while in the second they have not.

As an example, the term $\langle a \rangle$ representing an object of kind a inside the dataspace, has no outgoing transition labelled with $\checkmark$; on the other hand, the program $take(a)_t?P\_Q$, representing the execution of a *take* operation with a remaining timeout period of t , we define the transition $take(a)_t?P\_Q \xrightarrow{\checkmark} take(a)_{t-1}?P\_Q$.

Terms of the kind $take(a)_t?P\_Q$ cannot appear as initial programs of a configuration; they only arise as the result of the scheduling of a *take* operation. The term $take(a, t)?P\_Q$ is not sensible to the passing of time until the execution of the *take* operation is scheduled; this instant is represented by the transition $take(a, t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$. The index t means that t complete intervals should pass before the operation fails; the failure is modeled by the transition $take(a)_0?P\_Q \xrightarrow{\checkmark} Q$ (in this way the failure happens after that $t - 1$ transitions labelled with $\checkmark$ are executed).

We introduce the notation $\eta_t?P\_Q$ to denote either the term $take(a)_t?P\_Q$ or $read(a)_t?P\_Q$. The duration t is an integer number or a special symbol ∞ which denotes an infinite duration (we assume $\infty - 1 = \infty$).

The new syntax is obtained by removing the $read(a)$ and $take(a)$ prefixes and by extending the possible guards of the if-then-else forms:

$$\eta ::= \dots \mid take(a, t) \mid read(a, t) \mid take(a)_t \mid read(a)_t$$

The new semantics informally described above is modeled by introducing the new label $\checkmark$ and the axioms and rules reported in Table 3 (the axioms (3') and (4') are substituted for (3) and (4), respectively).

Axioms (3') and (4') model the starting of the timeout periods. Axioms (20) and (21) represent a successful execution of these operations, while axioms (22) and (23) represent the passing of time; the subscript t in $\eta_t?P\_Q$ is decremented if it is not 0 (axiom (22)), otherwise the timeout period finishes and the second continuation is chosen (axiom (23)). The rules (24) and (25) describe how the structured term $P|Q$ behaves according to the passing of time. If both the processes have an outgoing transition labeled with $\checkmark$, they synchronize on the execution of this operation; on the other hand, one of the two processes can perform its own transition $\checkmark$ locally, only if the other one is not sensible to the passing of time, i.e., it has no outgoing transitions labeled with $\checkmark$. A negative premise is used in the rule (25), but the labelled transition system is still well defined.

Observe that a timeout period may terminate even if the sought object is available. Thus, when an operation fails we cannot conclude anything about the presence or absence of the requested object. We have adopted this weak semantics as it seems the intended interpretation of JavaSpaces; see [12], where in Section 2.5 we read “A **read** request ... will wait until a matching entry is found ... up to the timeout period”. For example, in the configuration:

$$\langle a \rangle \mid read(a, t)?\mathbf{0}.write(b)$$

object $\langle b \rangle$ may be produced if $\langle a \rangle$ is not found before the end of the timeout t .

Using Timeout. The operations with timeout are particularly useful in a scenario where the test for absence, incorporated in the nonblocking operations $take\exists$ and $read\exists$, is difficult to implement. For example, in distributed implementations of the shared dataspace, a test for absence requires the snapshot of the whole distributed data repository. In order to avoid this, we may obtain nonblocking input operation by using timeouts. As an example, consider a simple program which has to consume one object which could be either $\langle a \rangle$ or $\langle b \rangle$ before terminating. In the kernel language this program could be defined as follows:

$$Consume = take\exists(a)?\mathbf{0}.(take\exists(b)?\mathbf{0}.Consume)$$

If the first $take\exists(a)$ operation requires a long time to be performed, the whole program could be negatively affected by this. For this reason, the following alternative program could be more efficient:

$$NewConsume = take(a, t)?\mathbf{0}.(take(b, t)?\mathbf{0}.NewConsume)$$

provided that the timeout t is defined appropriately.

The use of timeouts seems particularly useful also when the dataspace service is not reliable. Consider, as an example, a server implementing the dataspace which fails during the execution of a blocking $read$ or $take$ operation; the corresponding client is blocked until the server recovers. On the other hand, in the presence of timeout, the intended semantics allows the client to restart at least its local computation at the moment the indicated timeout period finishes.

Synchronous v.s. Asynchronous Interpretation of Time Passing. We have modeled configurations in which the passing of time is global, i.e., it is the same for all the components. According to this approach, the time passes synchronously. This is ensured by the side condition $\alpha \neq \surd$ of the locality rule (11), according to which a processes cannot perform locally its own transitions $\surd$. If we remove this side condition we obtain configurations in which the time passes asynchronously, as their components may or may not synchronize on the execution of their transitions labelled with $\surd$. In the following we use $P \longrightarrow_s^* P'$ to denote that a configuration P' can be reached from P in the transition system under the synchronous interpretation, while we use $P \longrightarrow_a^* P''$ to denote that P'' can be reached under the asynchronous one.

The synchronous approach is used to model centralized systems with a global clock, while the asynchronous approach corresponds to distributed systems where the global clock is not available. Distributed and centralized systems usually present strongly different behaviours; however, we prove that for the calculus presented in this section this discrimination does not hold. On the other hand, we will see in the next section that the introduction of leasing in the calculus

will permit to observe differences between the synchronous and the asynchronous interpretations.

The equivalence between the two approaches is a consequence of the following theorem, stating that, given an initial configuration P , the configurations that can be reached from P are exactly the same under the synchronous and the asynchronous approaches.

Theorem 1. *Let P be a configuration not including terms of the kind $\eta_t?P.Q$. We have that $P \longrightarrow_s^* P'$ if and only if $P \longrightarrow_a^* P'$.*

The proof of the theorem (that we omit due to space limits) is based on two observations: first, a computation under the synchronous approach is trivially valid also under the asynchronous one; second, the order of $\surd$ transitions in a computation under the asynchronous approach can be changed in order to obtain another valid computation leading to the same configuration such that the new computation can be mimicked under the synchronous approach. Intuitively, this can be done because there is no way to observe the instant in which a timeout period starts, instant which is modeled by transitions labelled with τ (see rules (3') and (4'))

5 Leasing

Leasing represents an emerging style of programming for distributed systems and applications. According to this style, a service offered by one object to another one is based on a notion of “granting the service for a certain period of time”.

$$\begin{array}{ll}
 (2'') \quad (\nu l) \text{write}(a, t).P \xrightarrow{\tau} \langle a \rangle_{l':t} | P\{l'/l\} & \text{with } l' \text{ fresh} \\
 (14') \quad (\nu l) \text{notify}(a, Q, t).P \xrightarrow{\tau} \text{on}(a, Q)_{l':t} | P\{l'/l\} & \text{with } l' \text{ fresh} \\
 (1') \quad \langle a \rangle_{l:t} \xrightarrow{\bar{a}} \mathbf{0} & (15') \quad \text{on}(a, P)_{l:t} \xrightarrow{\dot{a}} P | \text{on}(a, P)_{l:t} \\
 (26) \quad \text{cancel}(l)?P.Q \xrightarrow{\dagger l} P & (27) \quad \text{cancel}(l)?P.Q \xrightarrow{\neg \dagger l} Q \\
 (28) \quad \text{renew}(l, t)?P.Q \xrightarrow{l:t} P & (29) \quad \text{renew}(l, t)?P.Q \xrightarrow{\neg l:t} Q \\
 (30) \quad P_{l:t} \xrightarrow{\bar{l}} \mathbf{0} & (31) \quad P_{l:t'} \xrightarrow{\bar{l:t}} P_{l:t} \\
 (32) \quad P_{l:t} \xrightarrow{\surd} P_{l:t-1} \quad \text{if } t \neq 0 & (33) \quad P_{l:0} \xrightarrow{\surd} \mathbf{0} \\
 (34) \quad \frac{P \xrightarrow{\neg \lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P | Q \xrightarrow{\neg \lambda} P' | Q'} \quad \lambda = \dagger l \quad \text{or} \quad \lambda = l : t
 \end{array}$$

Table 4. Operational semantics for leasing (symmetric rules omitted).

In this way, objects which ask for services declare also the corresponding period of interest in that service. These are usually called leased services.

As it happens in JavaSpaces, we could introduce the notion of leasing for the *write* and *notify* operations. In our calculus we obtain this by adding to these operations a parameter which represents the duration of the interval for which the emitted datum should be maintained inside the data repository (for *write*) or the amount of time after which the listener for the event should be removed (for *notify*).

Inspired by JavaSpaces, we permit to cancel or renew previously leased services. In order to obtain this, each leased service is provided with a *unique identifier* which is used by two new operations called *cancel* and *renew* in order to indicate the service to be cancelled or renewed, respectively. Let *Lease*, ranged over by $l, l', \dots$, be the set of lease identifiers.

The syntax of the calculus redefines the possible prefixes μ and extends the guards η :

$$\begin{aligned}\mu &::= (\nu l)write(a, t) \mid (\nu l)notify(a, C, t).P \\ \eta &::= \dots \mid cancel(l) \mid renew(l, t)\end{aligned}$$

The notation and the mechanism we use to model the lease identifiers has been inspired by the π -calculus [11], where $(\nu l)P$ binds the name l in P . Similarly, we use $(\nu l)write(a, t).P$ and $(\nu l)notify(a, C, t).P$ to bind l in P . This reflects the fact that a leased object can be accessed (via the *renew* and *cancel* operations) only in the continuation of the *write* or *notify* operation which produced it. Operationally, we avoid name clashes by dynamically creating a fresh name (not previously used), to be substituted for the name l in the process P , whenever the *write* or *notify* operation is executed. For example, when a $(\nu l)write(a, t).P$ process performs its *write* operation, a new leased object is produced with a life length t and a fresh leasing identifier l' ; this fresh name is substituted for the name l in P (see axiom (2'')). The same mechanism is used also in axiom (14') for the new version of the *notify* primitive.

In order to associate to the leased resources the indication of the corresponding leasing identifier l and the remaining time t , we use a subscript notation $l : t$. Namely, we represent objects and listeners with $\langle a \rangle_{l:t}$ and $on(a, P)_{l:t}$, respectively. As a shorthand notation we use $P_{l:t}$ to denote a term which is either $\langle a \rangle_{l:t}$ or $on(a, P)_{l:t}$. Axioms (1') and (15') adapt the corresponding (1) and (15) to deal with leased resources.

The *cancel* and *renew* operations are used as guards of if-then-else forms because the operations could either succeed or fail, according to the presence or absence of the required leased resource. The execution of a *cancel* on the leased object l is denoted by the new label $\dagger l$, while its failure by $\neg \dagger l$ (see axioms (26) and (27)). On the other hand, the redefinition to t of the time granted to the leased object l is denoted by the label $l : t$, while its failure by $\neg l : t$ (see axioms (26) and (27)). The axioms (30) and (31) model the complementary operations of $\dagger l$ and $l : t$, respectively, which may be performed at any moment by the leased object l .

In order to execute a transition labelled with $\neg \dagger l$ (resp. $\neg l : t$) it is required that the environment does not contain any leased object with identifier l , formally, that it cannot perform any transition labeled with $\dagger l$ (resp. $l : t$). This is described by the rule (34).

The operational semantics is redefined by adding the new labels in $\{\dagger l, l : t, \overline{\dagger l}, \overline{l : t}, \neg \dagger l, \neg l : t \mid l \in Lease, t \in \mathbb{N} \cup \{\infty\}\}$ and the new axioms in Table 4.

Example of a Seat Reservation Service. As an example of use of leased resources we consider a seat reservation system for trains. The idea is to associate to each train a process responsible for seat booking. The process should expire at the leaving of the train.

$$\begin{aligned} ResTrain &= (\nu l) notify(req, Reserve, time). CheckTrain_l \\ Reserve &= take(req). write(seat, \infty) \end{aligned}$$

The process $CheckTrain_l$ is responsible to manage variations of the train scheduling. For example, if it is cancelled, the reserving process should be removed, while if the departure is delayed, the remaining lifetime should be changed. Thus, $CheckTrain_l$ (parametric in the name of the leasing identifier) can be defined as follows:

$$CheckTrain_l = take(delay). renew(l, newtime)? CheckTrain_l\_0 \mid take(cancel). cancel(l)? 0\_0$$

Alternative Semantics. Also in the language extended with leasing, we could think to adopt either the synchronous or asynchronous interpretations of time passing described in the previous section. It is interesting to observe that the introduction of leased resources permits us to distinguish between them, formally, the Theorem 4.1 does not hold any more. As a counter example consider the following program:

$$write(a, t). read(b, t + 1)? 0\_ take(a)$$

After the execution of the write operation the following term is obtained:

$$\langle a \rangle_{l:t} \mid read(b, t + 1)? 0\_ take(a)$$

The right hand process requires the presence of object $\langle b \rangle$ in order to continue its execution. As this object will never be produced, its behaviour consists of waiting for a $t + 1$ long period, and then becoming process $take(a)$. The object on the left has a lifetime shorter than $t + 1$. According to the synchronous interpretation, $\langle a \rangle_{l:t}$ disappears before $take(a)$ can be performed, while this is not true under the asynchronous one. Thus, the $take(a)$ operation may succeed only under the asynchronous approach.

Another alternative interpretation concerns the granting policy. We model the leasing mechanism referring to an idel situation in which the leased services

are granted exactly for the requested amount of time. This is not always possible in situations in which resources are limited. For example, in the Jini Distributed Leasing Specifications [13], the service provider may decide to grant the resource for a shorter period. We can easily adapt our semantics in order to take into account this feature by replacing rules (2'') and (14') with the following:

$$\begin{aligned}
 (2'') \quad & (\nu l) \text{write}(a, t).P \xrightarrow{\tau} \langle a \rangle_{l':t'} | P\{l'/l\} && \text{with } l' \text{ fresh and } t' \leq t \\
 (14') \quad & (\nu l) \text{notify}(a, Q, t).P \xrightarrow{\tau} \text{on}(a, Q)_{l':t'} | P\{l'/l\} && \text{with } l' \text{ fresh and } t' \leq t
 \end{aligned}$$

The following example shows the differences between our approach and this alternative interpretation:

$$\text{write}(a, 10).\text{write}(b, 20).\text{take}\exists(b)?\mathbf{0} \_ (\text{take}(a).\text{write}(c))$$

According to the ideal approach, we are sure that object $\langle b \rangle$ cannot expire before $\langle a \rangle$; thus object $\langle c \rangle$ will never be produced. On the other hand, if we move to the Jini-like interpretation, the object $\langle b \rangle$ may be granted for a shorter period; thus, $\langle c \rangle$ could be produced.

6 Conclusion

We have proposed a formal semantics for primitives typically adopted by Linda-like coordination middlewares, with a particular emphasis on the innovations introduced in JavaSpaces specifications.

As a first outcome, this analysis enlightens some interesting unaddressed choices in the design of these languages which may influence the overall behaviour. For example, in Section 2 we discuss two alternative semantics for the *write* operation, and in Section 4 we present two possible variants for the representation of time.

We plan to use the formal framework developed here for the analysis and verifications of properties of systems modeled according to the coordination paradigm of JavaSpaces. This is particularly useful in this context as, when dealing with concurrent systems, it is quite difficult to intuitively gain a clear view of all possible computations, thus leaving undetected the presence of undesired behaviours. The adoption of a process algebraic approach permits us to address this kind of problems by adapting the large amount of formal techniques developed in this area.

For example, consider a master/worker system where the master produces some jobs that will be performed by the workers; only when all jobs have been processed, the whole system terminate. As a first attempt we may represent the system as follows:

$$\begin{aligned}
 \text{System} &= \text{Master} \mid \text{Worker} \mid \dots \mid \text{Worker} \\
 \text{Master} &= \text{write}(\text{job}).\text{write}(\text{job}).\dots.\text{write}(\text{job}) \\
 \text{Worker} &= \text{take}\exists(\text{job})?(\text{write}(\text{result}).\text{Worker})\_ \mathbf{0}
 \end{aligned}$$

At a deeper analysis, we can find an undesired behaviour: if the workers perform their first test before the master starts producing the jobs, the final configuration will contain unprocessed jobs.

The problem addressed above can be fixed by blocking the workers until all the jobs are produced. The correct specifications is obtained by redefining the master and the worker as follows:

$$\begin{aligned} \text{Master} &= \text{write}(\text{job}).\text{write}(\text{job}).\dots.\text{write}(\text{job}).\text{write}(\text{start}) \\ \text{Worker} &= \text{read}(\text{start}).\text{take}\exists(\text{job})?(\text{write}(\text{result}).\text{Worker})\_0 \end{aligned}$$

The correctness of this specification can be formally proved by showing that each computation terminates in a state consisting of the proper number of objects $\langle \text{result} \rangle$ (i.e. equal to the number of $\text{write}(\text{job})$ operations performed by the master process).

References

1. N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.
2. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, to appear, 2000. Extended abstract appeared in Proc. of *Coordination'97*.
3. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, to appear, 2000. Extended abstract appeared in Proc. of *Express'97*.
4. N. Busi and G. Zavattaro. Event Notification in Data-driven Coordination Languages: Comparing the Ordered and Unordered Interpretations. In *Proc. of SAC2000*. ACM Press, 2000.
5. N. Busi and G. Zavattaro. On the Expressiveness of Event Notification in Data-driven Coordination Languages. In *Proc. of ESOP2000*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.
6. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.
7. J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
9. J. McClain. Personal communications. March 1999.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
11. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.
12. Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.
13. Sun Microsystem, Inc. *Jini Distributed Leasing Specifications*, 1998.
14. P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.

The Algebra of Multi-tasking

Colin J. Fidge

Software Verification Research Centre, The University of Queensland,
Queensland 4072, Australia
cjf@it.uq.edu.au

Abstract. Modelling multi-tasking behaviour is an important phase of real-time system design. It is shown how task scheduling principles can be captured in a CCS-based process algebra via extensions for both asymmetric interleaving, to model intraprocessor scheduling decisions, and for asynchronous communication, to model interprocessor precedence constraints. Examples are given of task preemption, blocking on shared resources, and multi-task transactions.

1 Introduction

Real-time scheduling theory has advanced considerably in recent years, thanks to a simplified, abstract computational model of multi-tasking behaviour [2,6]. Here we exploit this model to show how task scheduling can be captured in a process algebraic formalism.

We begin with an existing timed, causal version of the Calculus of Communicating Systems (CCS). This formalism provides both a notion of real time, by allowing each atomic action to be stamped with its time of occurrence, and true concurrency semantics, by allowing each parallel agent to evolve in its own independent local context [1,10]. We then extend this calculus further by introducing prioritised choices between eligible actions, via an asymmetric interleaving operator, and by introducing precedence constraints between parallel agents, via an asynchronous message-passing mechanism. It is then shown how the result can be used to model both multi-tasking on a single processor, and multi-task transactions occurring across several processors.

2 Previous Work

Multi-tasking systems have been modelled in a variety of formalisms. For instance, Yuhua and Chaochen used the Duration Calculus to prove properties of a dynamic-priority scheduling policy for non-communicating tasks [19]. Corbett showed how to translate tasks written in the Ada programming language into an analysable timed automata model [7]. Jacky used operations in the Z specification language to model the actions of a multi-tasking run-time environment [13]. Dong et al. presented an abstract specification of multi-processor multi-tasking in the Object-Z notation [8]. Liu et al. used the Temporal Logic of Actions to

show how applying a scheduling policy to a set of tasks can be represented as a formal program transformation [14]. Bornot et al. used the Timed Automata with Deadlines formalism, plus a notion of ‘urgency’, to model non-preemptive task scheduling [4].

More relevant to this paper are algebraic approaches. It has long been recognised that standard algebraic formalisms are ill-equipped to model multi-tasking systems, so many attempts have been made to extend them. Jackson showed how a set of tasks and a non-preemptive scheduler can be represented in the CSP process algebra, using a special ‘tick’ event to simulate the passage of time. He then showed that the resulting model can be analysed using the FDR model checker (although he was forced to modify the checker itself to recognise event priorities) [12]. Van Glabbeek and Rittgen noted the inadequacy of (some) process algebras for modelling timing and delays, and went on to devise an entirely new algebra for representing scheduling of atomic actions of varying durations [17]. Hsiung et al. used timed automata to represent tasks as ‘clients’ and a timer-driven scheduler as their ‘server’ and applied model checking techniques to the result [11]. Breveglieri et al. noted the central role played by task queues in making scheduling decisions and used a syntactic approach to model operating-system scheduling policies by extending a formal grammar with queues. Each preemptible task was represented as a sequence of distinct atomic actions, each taking one quantum [5]. Finally, and closest of all to our approach, Ben-Abdallah et al. used the CCS-based timed process algebra ACSR-VP to specify and reason about a multi-tasking system. In their formalism each action incorporates a priority and resource usage requirements. To define the asymmetric choices characteristic of priority-driven scheduling, they introduced a special ‘preemption relation’ between actions and then defined a prioritised transition relation [3].

All of the above-cited work is relevant to the current paper. However, the limitations and complexity of these models inspired us to instead seek a straightforward extension of the already well-known CCS notation. In particular, our approach differs from all of the above work by providing distinct operators for both intra and inter-processor task composition, and by introducing asynchronous communication to model precedence constraints between tasks.

3 Definitions

In this section we define the notations, some laws, and the operational semantics of our multi-tasking algebra. It is based on Milner’s CCS language [15], extended with timestamped actions, local agent contexts, a new scheduling operator, and asynchronous communication.

3.1 Notations

As usual, we assume a set of names representing the atomic actions that CCS agents can perform [15, §2.4]. However, we also distinguish a set of names for use as *asynchronous communication* actions and, since asynchronous *sends* and *receives* behave differently, we sometimes further distinguish these actions. Let

- $a, b, \dots$ be standard, synchronous CCS actions (with conames $\bar{a}, \bar{b}, \dots$ denoting synchronous outputs) [15, p. 43],
- $i, j, \dots$ be asynchronous communication actions (with conames $\bar{i}, \bar{j}, \dots$ denoting asynchronous sends),
- $x, y, \dots$ be either synchronous or asynchronous actions,
- $m, n, \dots$ be synchronous actions or asynchronous send actions,
- $r, s, \dots$ be synchronous actions or asynchronous receive actions,
- $f, g, \dots$ be renaming functions on (synchronous or asynchronous) action names [15, p. 43], and
- $L, M, \dots$ be sets of (synchronous or asynchronous) action names [15, p. 43].

Since we are constructing a real-time model, we require each occurrence of an action x to be *timestamped* with a particular absolute time t , denoted ' $x@t$ ' [1,10]. The time domain is assumed to start at 0 and may be either discrete (the natural numbers) or dense (the non-negative reals). Let

- $t, u, \dots$ be (absolute) times,
- $x@t, y@u, \dots$ be timestamped events (action occurrences), and
- $I, J, \dots$ be sets of timestamped, asynchronous events.

To support true concurrency, parallel agents must carry their own local *contexts*. For an agent expression E , this usually consists of the time t at which E 's causal-predecessor finished, since this defines the earliest time at which E can begin [1,10]. Here we extend this mechanism to implement asynchronous communication. Let the context also contain a set I of timestamped events representing messages currently in transit to E . We denote agent expression E , starting no earlier than time t , and with a set I of incoming messages waiting to be received, as ' ${}_{(t,I)}E$ '. Let

- $P, Q, \dots$ be agents [15, p. 44] in our extended CCS,
- $A, B, \dots$ be named agent constants [15, p. 44], and
- ${}_{(t,I)}E, {}_{(u,J)}F, \dots$ be agent expressions [15, p. 43] with explicit contexts.

As usual, agent expressions can be constructed from the standard CCS operators for prefixing ' $\cdot$ ', summation '+', restriction '\', and relabelling '[$\cdot$]' [15, §2.4]. In our true concurrency model the CCS "composition" operator ' $|$ ' actually represents interprocessor *parallelism*, and we refer to it as such here. Our multi-tasking model also introduces an entirely new compositional operator for intraprocessor *scheduling*, denoted ' $\uparrow$ '.

To allow some activity a that consumes q quanta of computation time to be preempted, we model it as a sequence of atomic ' a ' actions, each consuming one quantum [5]. To avoid prefixing q copies of primitive action a , we allow the following syntactic shorthand [3].

$$a^q \cdot E \stackrel{\text{def}}{=} \overbrace{a \cdot (a \cdot \dots (a \cdot E))}^{q \text{ times}}$$

In the examples below we model static-priority scheduling by associating a priority with each action name. Let the relation ' $\prec$ ' define a partial order of action priorities. Expression ' $a \prec b$ ' tells us that action b has higher priority than action a , and ' $a \asymp b$ ' states that a and b have equal priority.

3.2 Equivalence Laws

Since each agent expression now has an associated time and incoming-message context, we introduce the following laws to show how these contexts distribute through the various agent constructors.

- (1) $(t, I)(P + Q) = (t, I)P + (t, I)Q$
- (2) $(t, I)(P \mid Q) = (t, I)P \mid (t, I)Q$
- (3) $(t, I)(P \upharpoonright Q) = (t, I)P \upharpoonright (t, I)Q$
- (4) $(t, I)(P \setminus L) = ((t, I)P) \setminus L$
- (5) $(t, I)(P[f]) = ((t, I)P)[f]$
- (6) $(t, I)\mathbf{0} = \mathbf{0}$
- (7) $(t, I)((u, J)(x \cdot P)) = (\max(t, u), I \cup J)(x \cdot P)$
- (8) $(t, I \cup \{i @ t\})P = (t, I)P \quad \text{if } i \text{ can never appear in } P$

Laws (1) to (5) tell us that contexts distribute in the obvious way through the main agent constructors [1]. Law (6) tells us that contexts are redundant for the nil agent $\mathbf{0}$ since it cannot perform any actions in any context. Law (7) relates to the prefix operator, which is the only one that directly performs actions in CCS, and shows how nested contexts are merged. If agent ' $x \cdot P$ ' has two earliest starting times, t and u , then its earliest starting time is the later of the two. If the agent has two sets of incoming messages, I and J , then its complete set of incoming messages is the union of the two. Lastly, Law (8) states that if agent P is incapable of ever receiving some incoming asynchronous message i , then we can remove i from P 's context. (For this to be applicable, i must not appear in P , any possible relabelling of P , or any agent into which P can evolve.)

So that it is not always necessary to explicitly provide a complete context for every agent expression, we allow the following syntactic equivalences.

- (9) $E \stackrel{\text{def}}{=}_{(0, \emptyset)} E$
- (10) ${}_t E \stackrel{\text{def}}{=}_{(t, \emptyset)} E$
- (11) ${}_I E \stackrel{\text{def}}{=}_{(0, I)} E$

Law (9) tells us that an agent expression E with no explicit context is equivalent to the agent in a context where the earliest starting time is 0 and there are no messages in transit. (Expression E may contain one or more contexts 'internally'.) Recall from Law (7) that adding context ' $(0, \emptyset)$ ' to an agent has no effect. Law (9) allows us to write agent expressions without explicit contexts below. In the same way, Law (10) allows us to provide just a starting time t , and Law (11) allows us to provide just an incoming-message set I , when desired.

We conclude this section with two frequently-used equational laws [15, Ch. 3] for our parallel and scheduling operators.

- (12) $P \mid \mathbf{0} = \mathbf{0} \mid P = P$
- (13) $P \upharpoonright \mathbf{0} = \mathbf{0} \upharpoonright P = P$

3.3 Operational Semantics

As usual, we define the operational semantics of our algebra via a labelled transition system [15, §2.5]. Let transition $E \xrightarrow{x@t} E'$ indicate that agent expression E is capable of performing action x at time t and then evolving into agent E' . Operator semantics are then defined by giving their complete set of transition rules. Each rule has a name, optional premises defining situations under which the rule may apply, a conclusion defining the transition that can be performed, and optional side conditions [15, p. 45].

$$\text{name} \frac{\text{premises}}{\text{conclusion}} (\text{conditions})$$

To define the prefixing operator in our formalism, we must consider the effect of performing an action on the passage of time, and must treat an asynchronous receive action as a special case.

$$\mathbf{Act}_1 \frac{}{(t, I)(m. E) \xrightarrow{m@t} (t+1, I) E}$$

$$\mathbf{Act}_2 \frac{}{(t, I)(i. E) \xrightarrow{i@v} (v+1, I \setminus \{i@u\}) E} \quad (i@u \in I \text{ and } v = \max(t, u))$$

Rule $\mathbf{Act}_1$ tells us that an agent prefixed by action m , in context (t, I) , can perform action m at time t , and thereby evolve into an agent with context $(t+1, I)$. Incrementing the time in the context reflects our assumption that each atomic action consumes one quantum.

Rule $\mathbf{Act}_2$ covers the special case of asynchronous receives. It states that an agent in context (t, I) , prefixed by an asynchronous receive action i , can perform action i at time v , and evolve into an agent with context $(v+1, I \setminus \{i@u\})$. The side condition requires that the initial incoming-message set I must contain a timestamped event $i@u$, indicating that the corresponding send action $\bar{i}$ was completed at time u . (The way asynchronous messages are added to contexts is defined by rules $\mathbf{Par}_4$ and $\mathbf{Par}_5$, below.) Furthermore, the side condition states that the time of receipt v is the later of the local time t and the time u at which the message became available. The updated context subtracts event $i@u$ from set I , since this message has now been consumed.

The derivation rules for the summation, restriction, relabelling and agent constant definition operators are identical to standard CCS [15, p. 46], except that each action x is stamped with its time of occurrence t [1, 10].

$$\mathbf{Sum}_1 \frac{E \xrightarrow{x@t} E'}{E + F \xrightarrow{x@t} E'} \quad \mathbf{Sum}_2 \frac{F \xrightarrow{x@t} F'}{E + F \xrightarrow{x@t} F'}$$

$$\mathbf{Res} \frac{E \xrightarrow{x@t} E'}{E \setminus L \xrightarrow{x@t} E' \setminus L} \quad (x, \bar{x} \notin L) \quad \mathbf{Rel} \frac{E \xrightarrow{x@t} E'}{E[f] \xrightarrow{f(x)@t} E'[f]}$$

$$\mathbf{Con} \frac{P \xrightarrow{x@t} P'}{A \xrightarrow{x@t} P'} (A \stackrel{\text{def}}{=} P)$$

Similarly, the rules for parallel composition of independent actions and synchronous message-passing correspond to the untimed case [15, p. 46].

$$\begin{aligned} \mathbf{Par}_1 & \frac{E \xrightarrow{r@t} E'}{E \mid F \xrightarrow{r@t} E' \mid F} & \mathbf{Par}_2 & \frac{F \xrightarrow{r@t} F'}{E \mid F \xrightarrow{r@t} E \mid F'} \\ \mathbf{Par}_3 & \frac{E \xrightarrow{a@t} E' \quad F \xrightarrow{\bar{a}@t} F'}{E \mid F \xrightarrow{\tau@t} E' \mid F'} \end{aligned}$$

Rule **Par**₃ allows a synchronous communication event only when both agents agree on its time of occurrence t . Rules **Par**₁ and **Par**₂ allow (truly) parallel agents to evolve independently, each with their own contexts [1,10].

However, an asynchronous message send $\bar{i}$ is treated specially to allow for the addition of the message in transit to the context of the other agent.

$$\begin{aligned} \mathbf{Par}_4 & \frac{E \xrightarrow{\bar{i}@t} E'}{E \mid F \xrightarrow{\bar{i}@t} E' \mid \{i@(t+1)\}F} & \mathbf{Par}_5 & \frac{F \xrightarrow{\bar{i}@t} F'}{E \mid F \xrightarrow{\bar{i}@t} \{i@(t+1)\}E \mid F'} \end{aligned}$$

In rule **Par**₄, for instance, the occurrence in agent E of an asynchronous send $\bar{i}$ at time t , results in the timestamped event $i@(t+1)$ being added to the context of agent F . This indicates that the corresponding receive action i may now occur in F at time $t+1$ or later. (If agent F was itself composed of parallel agents, the effect is to *broadcast* message i to all of these agents—we assume that only one parallel operand contains a corresponding receive action and others will ignore the message. Furthermore, we did not add i to the context of the sending agent E here on the assumption that agents do not send messages to themselves, i.e., there is no intraprocessor asynchronous communication.)

The final set of rules define the behaviour of our new ‘intraprocessor scheduling’ operator. The first two tell us that when two agents are each ready to perform an action, then the scheduling operator favours the earlier possible action. In other words, the imaginary ‘processor’ will not stand idle while an action is ready.

$$\begin{aligned} \mathbf{Sch}_1 & \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@u} F'}{E \mid F \xrightarrow{x@t} E' \mid_{(t+1)} F} (t < u) & \mathbf{Sch}_2 & \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@u} F'}{E \mid F \xrightarrow{y@u} F' \mid_{(t+1)} E} (u < t) \end{aligned}$$

In rule **Sch**₂, the order of the ‘ $\mid$ ’ operands is reversed after agent F performs an action. The rules always place the agent that has just performed an action on the left. In this way the structure of the agent expression remembers the order in which scheduled agents have executed. This ensures that preempted agents resume execution in the proper sequence. Also, when one agent performs an action, the time in the context of the other agent is increased. Unlike parallel

composition, ‘scheduled’ agents share a common time-frame, so the passage of time in one affects both.

The next two rules tell us that when two agents are ready to perform actions at the same time, the scheduling operator favours higher-priority actions. Thus we model a priority-driven scheduling policy.

$$\text{Sch}_3 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@t} F'}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} (y \prec x) \quad \text{Sch}_4 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@t} F'}{E \upharpoonright F \xrightarrow{y@t} F' \upharpoonright_{(t+1)} E} (x \prec y)$$

The next rule defines what happens when both agents can perform actions of the same priority at the same time. In this case the scheduling operator always favours its left-hand operand. Since this will usually be the agent that last performed an action, this models a ‘no unnecessary preemption’ scheduling policy in which tasks that have started executing are allowed to continue unless preempted by a strictly higher priority task.

$$\text{Sch}_5 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@t} F'}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} (x \asymp y)$$

All the scheduling rules above applied to cases where both operands were ready to perform an action. Lastly, we must therefore explicitly say what happens when only one operand is ready. Let ‘ $E \nrightarrow$ ’ mean that agent E cannot perform any action.

$$\text{Sch}_6 \frac{E \xrightarrow{x@t} E' \quad F \nrightarrow}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} \quad \text{Sch}_7 \frac{E \nrightarrow \quad F \xrightarrow{x@t} F'}{E \upharpoonright F \xrightarrow{x@t} F' \upharpoonright_{(t+1)} E}$$

Rules with negative premises are not normally needed in CCS because an agent that cannot perform any action is equivalent to the nil agent $\mathbf{0}$ and can be eliminated. This is not, however, the case with asynchronous message passing. An agent that is awaiting a message may not be able to perform an action in its current context, but may later be able to perform a receive action in an updated context. Agents waiting for asynchronous messages must therefore be distinguished from the nil agent which can never perform any action, in any context. (Negative premises must be introduced with care since they can result in ill-defined, ‘circular’ transition system definitions [18].)

4 Examples

This section presents a series of examples illustrating some of the capabilities of the language defined above.

4.1 Preemption and Offsets

The asymmetry of our scheduling operator allows it to be used to model task preemption, and the time contexts can be used to model starting-time offsets.

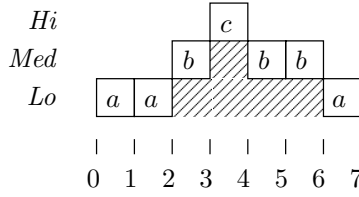


Fig. 1. Timeline for preemptive scheduling example.

Consider a system in which three tasks are required to execute on the same processor. A low-priority task requires 3 quanta, starting at time 0; a medium-priority task requires 3 quanta, starting at time 2; and a high-priority task requires 1 quantum, starting at time 3. We can model this system as three agents, assuming that action priorities obey the ordering $a \prec b \prec c$.

$$Lo \stackrel{\text{def}}{=} {}_0(a^3 \cdot \mathbf{0}) \quad Med \stackrel{\text{def}}{=} {}_2(b^3 \cdot \mathbf{0}) \quad Hi \stackrel{\text{def}}{=} {}_3(c \cdot \mathbf{0})$$

The following derivation steps show the behaviour of these three agents when composed using our scheduling operator. The sequence of timestamped events can be seen down the left. This is shown graphically in Figure 1. The derivation rules from Section 3.3 used at each step appear on the right. Laws from Section 3.2 are applied frequently and noted in interesting cases only.

$$\begin{array}{ll}
 Lo \mid Med \mid Hi & \\
 \xrightarrow{a@0} {}_1(a^2 \cdot \mathbf{0}) \mid {}_1Med \mid {}_1Hi & [\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_1 \text{ (twice)}] \\
 \xrightarrow{a@1} {}_2(a \cdot \mathbf{0}) \mid {}_2Med \mid {}_2Hi & [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_1 \text{ (twice)}] \\
 \xrightarrow{b@2} {}_3(b^2 \cdot \mathbf{0}) \mid {}_3(a \cdot \mathbf{0}) \mid {}_3Hi & [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_4 \text{ and } \mathbf{Sch}_1] \\
 \xrightarrow{c@3} {}_4(b^2 \cdot \mathbf{0}) \mid {}_4(a \cdot \mathbf{0}) & [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_4 \text{ and } (13)] \\
 \xrightarrow{b@4} {}_5(b \cdot \mathbf{0}) \mid {}_5(a \cdot \mathbf{0}) & [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_3] \\
 \xrightarrow{b@5} {}_6(a \cdot \mathbf{0}) & [\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } (13)] \\
 \xrightarrow{a@6} \mathbf{0} & [\mathbf{Act}_1 \text{ and } (6)]
 \end{array}$$

These steps model priority-based task preemption. After performing its first two ‘a’ quanta, the *Lo* agent is preempted by the ability of the *Med* agent to perform its first *b* action at time 2. The *Med* agent is itself preempted by the *Hi* agent at time 3. The shaded areas in Figure 1 thus show where a task was ready to perform an action but was preempted by a higher-priority one. Note that when a preempting task finishes, the lower-priority tasks resume execution in the correct sequence.

4.2 Repetitive Behaviours

Scheduling theory usually assumes that each task performs an infinite sequence of invocations, with the arrivals of successive invocations separated by a known

minimum time [2]. We can model repeated task arrivals using recursive agent constant definitions. Just as standard CCS allows agent constants to be parameterised via subscripts [15, §2.8], we allow them to be parameterised with contexts.

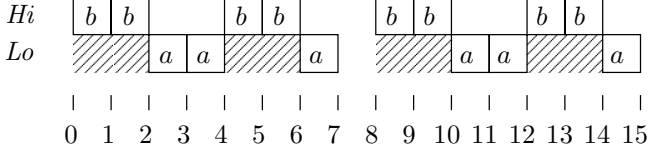


Fig. 2. Timeline for periodic scheduling example.

Consider a system consisting of two tasks. A low-priority task arrives every 8 time units and requires 3 quanta at each invocation, and a high-priority task arrives every 4 time units and requires 2 quanta at each invocation. This can be modelled via two recursive agent definitions, each parameterised with a time context t , assuming action priorities such that $a \prec b$.

$${}_tLo \stackrel{\text{def}}{=} {}_t(a^3 \cdot ({}_{t+8}Lo) \quad {}_tHi \stackrel{\text{def}}{=} {}_t(b^2 \cdot ({}_{t+4}Hi)$$

When composed using the scheduling operator, the derivation tree for this system is infinite. Its first seven steps are as follows, after which the cycle repeats. The behaviour of the system is illustrated graphically in Figure 2.

$$\begin{array}{ll}
 {}_0Lo \upharpoonright {}_0Hi & \\
 \xrightarrow{b@0} {}_1(b \cdot {}_4Hi) \upharpoonright {}_1(a^3 \cdot {}_8Lo) & [\text{Con}, \text{Act}_1 \text{ and Sch}_4] \\
 \xrightarrow{b@1} {}_4Hi \upharpoonright {}_2(a^3 \cdot {}_8Lo) & [\text{Act}_1 \text{ and Sch}_3] \\
 \xrightarrow{a@2} {}_3(a^2 \cdot {}_8Lo) \upharpoonright {}_4(b^2 \cdot {}_8Hi) & [\text{Con}, \text{Act}_1 \text{ and Sch}_2] \\
 \xrightarrow{a@3} {}_4(a \cdot {}_8Lo) \upharpoonright {}_4(b^2 \cdot {}_8Hi) & [\text{Act}_1 \text{ and Sch}_1] \\
 \xrightarrow{b@4} {}_5(b \cdot {}_8Hi) \upharpoonright {}_5(a \cdot {}_8Lo) & [\text{Con}, \text{Act}_1 \text{ and Sch}_4] \\
 \xrightarrow{b@5} {}_8Hi \upharpoonright {}_6(a \cdot {}_8Lo) & [\text{Act}_1 \text{ and Sch}_3] \\
 \xrightarrow{a@6} {}_8Lo \upharpoonright {}_8Hi & [\text{Act}_1 \text{ and Sch}_2]
 \end{array}$$

4.3 Blocking on Shared Resources

When tasks lock shared resources in static-priority scheduling, their active priority is temporarily raised, to hasten their execution, so that they do not unnecessarily block higher-priority tasks [2,6]. We can model this phenomenon by allowing a low-priority agent to perform high-priority actions, to represent periods when a resource shared with a high-priority task is locked.

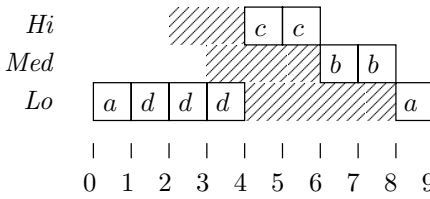


Fig. 3. Timeline for blocking example.

Consider a system of three tasks in which the lowest and highest priority tasks both access the same shared resource. A low-priority task arrives at time 0 requiring 5 quanta, but during its 2nd, 3rd and 4th quanta it needs access to the shared resource; a medium-priority task arrives at time 3 and requires 2 quanta; and a high-priority task arrives at time 2 and requires 2 quanta. This can be modelled using the following three agents. Assume that the priorities of actions are as follows: $a \prec b \prec c \prec d$. Action d represents times when the low-priority task has locked the shared resource, and is effectively executing at the same priority as the high-priority task.

$$Lo \stackrel{\text{def}}{=} {}_0(a.(d^3.(a.\mathbf{0}))) \qquad Med \stackrel{\text{def}}{=} {}_3(b^2.\mathbf{0}) \qquad Hi \stackrel{\text{def}}{=} {}_2(c^2.\mathbf{0})$$

The behaviour of these agents when executing on the same processor is modelled by the following derivation. This is illustrated graphically in Figure 3.

$Hi \upharpoonright Med \upharpoonright Lo$	
$\xrightarrow{a@0} {}_1(d^3.(a.\mathbf{0})) \upharpoonright {}_1Hi \upharpoonright {}_1Med$	$[\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_2 \text{ and } (3)]$
$\xrightarrow{d@1} {}_2(d^2.(a.\mathbf{0})) \upharpoonright {}_2Hi \upharpoonright {}_2Med$	$[\mathbf{Act}_1, \mathbf{Sch}_1 \text{ and } (3)]$
$\xrightarrow{d@2} {}_3(d.(a.\mathbf{0})) \upharpoonright {}_3Hi \upharpoonright {}_3Med$	$[\mathbf{Act}_1, \mathbf{Sch}_5 \text{ and } (3)]$
$\xrightarrow{d@3} {}_4(a.\mathbf{0}) \upharpoonright {}_4Hi \upharpoonright {}_4Med$	$[\mathbf{Act}_1, \mathbf{Sch}_5 \text{ and } (3)]$
$\xrightarrow{c@4} {}_5(c.\mathbf{0}) \upharpoonright {}_5Med \upharpoonright {}_5(a.\mathbf{0})$	$[\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } \mathbf{Sch}_4]$
$\xrightarrow{c@5} {}_6Med \upharpoonright {}_6(a.\mathbf{0})$	$[\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } (13)]$
$\xrightarrow{b@6} {}_7(b.\mathbf{0}) \upharpoonright {}_7(a.\mathbf{0})$	$[\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_3]$
$\xrightarrow{b@7} {}_8(a.\mathbf{0})$	$[\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } (13)]$
$\xrightarrow{a@8} \mathbf{0}$	$[\mathbf{Act}_1 \text{ and } (6)]$

At time 2 the *Hi* agent is ready to perform action *c*. However, because the *Lo* agent is already performing equal-priority *d* actions, i.e., has locked the shared resource, it continues to get preference. This is achieved in the formalism by making the *Lo* agent the left-hand, favoured operand of the scheduling operator. (Letting the high-priority task start executing would break the lock on the shared resource.) The high-priority task thus experiences ‘direct’ blocking [6] at

time 2. At time 3 the *Med* agent becomes ready to perform action *b*, but cannot because the *Lo* agent's *d* actions have higher-priority. This is an example of 'push-through' blocking—the medium-priority task is blocked even though it does not use the shared resource [6]. Once the *Lo* agent stops executing high-priority *d* actions, i.e., unlocks the shared resource, it is immediately preempted.

4.4 Multiprocessor Scheduling

In complex embedded systems, sequences of task invocations are linked to form end-to-end transactions. The tasks comprising a transaction may reside on different processors, so interprocessor precedence constraints must be enforced to ensure that they execute in the correct sequence. We can use our (true concurrency) parallelism operator to model multiple processors, and asynchronous communication to model precedence constraints.

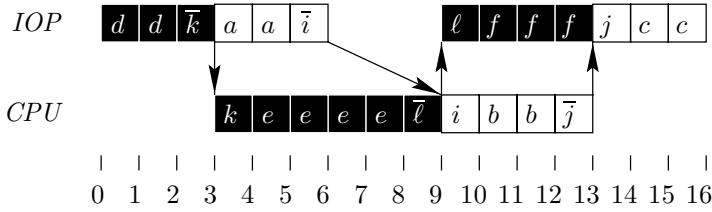


Fig. 4. Timeline for multiprocessor example.

Consider part of an embedded avionics system [9, §4.3.2.1]. A low-priority transaction consists of three task invocations. The first task performs sensor input operations and requires 2 quanta; the second task processes the data and requires 2 quanta; and the third task performs actuator output operations and requires 2 quanta. To ensure that the three tasks synchronise correctly, when the first task finishes it sends a message *i* to the second, and when the second finishes it sends a message *j* to the third. This can be modelled by the following three agents.

$$LoIn \stackrel{\text{def}}{=} a^2.(\bar{i}.0) \quad LoProc \stackrel{\text{def}}{=} i.(b^2.(\bar{j}.0)) \quad LoOut \stackrel{\text{def}}{=} j.(c^2.0)$$

A high-priority transaction is structured similarly. Its input task requires 2 quanta; its processing task requires 4 quanta; and its output task requires 3 quanta. Asynchronous messages *k* and *ℓ* are used for task synchronisation.

$$HiIn \stackrel{\text{def}}{=} d^2.(\bar{k}.0) \quad HiProc \stackrel{\text{def}}{=} k.(e^4.(\bar{\ell}.0)) \quad HiOut \stackrel{\text{def}}{=} \ell.(f^3.0)$$

We assume that all actions in the high-priority transaction have priority over all actions in the low-priority one: $\{a, b, c, i, j\} \prec \{d, e, f, k, \ell\}$.

Next the tasks are assigned to processors. All the input and output tasks are scheduled on a dedicated I/O Processor, and the two data processing tasks

are scheduled on a separate CPU. The complete system then consists of the I/O Processor and CPU running in parallel.

$$IOP \stackrel{\text{def}}{=} (LoIn \mid HiIn \mid LoOut \mid HiOut) \quad CPU \stackrel{\text{def}}{=} (LoProc \mid HiProc)$$

$$Sys \stackrel{\text{def}}{=} (IOP \mid CPU)$$

The behaviour of this system is modelled by the following derivation sequence. (This is not the only possible derivation because parallel, independent events can be interleaved differently. However, all such interleavings generate the same set of timestamped events.) The way events occur on the two processors is illustrated graphically in Figure 4. Arrows show the precedence constraints imposed by asynchronous communication.

$$\begin{array}{ll}
 Sys & \\
 \xrightarrow{d@0} ({}_1(d.(\bar{k}.0)) \mid {}_1LoIn \mid {}_1LoOut \mid {}_1HiOut) \mid & [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_3, \\
 CPU & \mathbf{Sch}_4 \text{ and } \mathbf{Par}_1] \\
 \xrightarrow{d@1} ({}_2(\bar{k}.0) \mid {}_2LoIn \mid {}_2LoOut \mid {}_2HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } \mathbf{Par}_1] \\
 CPU & \\
 \xrightarrow{\bar{k}@2} ({}_3LoIn \mid {}_3LoOut \mid {}_3HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_3, \mathbf{Par}_4 \text{ and } (13)] \\
 \{k@3\} CPU & \\
 \xrightarrow{a@3} ({}_4(a.(\bar{i}.0)) \mid {}_4LoOut \mid {}_4HiOut) \mid & [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_1] \\
 \{k@3\} CPU & \\
 \xrightarrow{a@4} ({}_5(\bar{i}.0) \mid {}_5LoOut \mid {}_5HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_1] \\
 \{k@3\} CPU & \\
 \xrightarrow{k@3} ({}_5(\bar{i}.0) \mid {}_5LoOut \mid {}_5HiOut) \mid & [\mathbf{Con}, \mathbf{Act}_2, \mathbf{Sch}_7, \mathbf{Par}_2 \text{ and } (8)] \\
 ({}_4(e^4.(\bar{\ell}.0)) \mid {}_4LoProc) & \\
 \xrightarrow{e@4} ({}_5(\bar{i}.0) \mid {}_5LoOut \mid {}_5HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_2] \\
 ({}_5(e^3.(\bar{\ell}.0)) \mid {}_5LoProc) & \\
 \xrightarrow{e@5} ({}_5(\bar{i}.0) \mid {}_5LoOut \mid {}_5HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_2] \\
 ({}_6(e^2.(\bar{\ell}.0)) \mid {}_6LoProc) & \\
 \xrightarrow{e@6} ({}_5(\bar{i}.0) \mid {}_5LoOut \mid {}_5HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_2] \\
 ({}_7(e.(\bar{\ell}.0)) \mid {}_7LoProc) & \\
 \xrightarrow{\bar{i}@5} ({}_6LoOut \mid {}_6HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_6, \mathbf{Par}_4 \text{ and } (13)] \\
 \{i@6\} ({}_7(e.(\bar{\ell}.0)) \mid {}_7LoProc) & \\
 \xrightarrow{e@7} ({}_6LoOut \mid {}_6HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } \mathbf{Par}_2] \\
 \{i@6\} ({}_8(\bar{\ell}.0) \mid {}_8LoProc) & \\
 \xrightarrow{\bar{\ell}@8} \{ \ell@9 \} ({}_6LoOut \mid {}_6HiOut) \mid & [\mathbf{Act}_1, \mathbf{Sch}_3, \mathbf{Par}_5 \text{ and } (13)] \\
 ({}_9, \{i@6\}) LoProc &
 \end{array}$$

$$\begin{array}{ll}
\begin{array}{l} \xrightarrow{i@9} \{ \ell@9 \} ({}_6LoOut \upharpoonright {}_6HiOut) \mid \\ {}_{10}(b^2 \cdot (\bar{j} \cdot 0)) \end{array} & [\mathbf{Con}, \mathbf{Act}_2 \text{ and } \mathbf{Par}_2] \\
\begin{array}{l} \xrightarrow{\ell@9} ({}_{10}(f^3 \cdot 0) \upharpoonright {}_{10}LoOut) \mid \\ {}_{10}(b^2 \cdot (\bar{j} \cdot 0)) \end{array} & [\mathbf{Con}, \mathbf{Act}_2, \mathbf{Sch}_7, \mathbf{Par}_1 \text{ and } (8)] \\
\begin{array}{l} \xrightarrow{b@10} ({}_{10}(f^3 \cdot 0) \upharpoonright {}_{10}LoOut) \mid {}_{11}(b \cdot (\bar{j} \cdot 0)) \\ \xrightarrow{b@11} ({}_{10}(f^3 \cdot 0) \upharpoonright {}_{10}LoOut) \mid {}_{12}(\bar{j} \cdot 0) \end{array} & [\mathbf{Act}_1 \text{ and } \mathbf{Par}_2] \\
\begin{array}{l} \xrightarrow{f@10} ({}_{11}(f^2 \cdot 0) \upharpoonright {}_{11}LoOut) \mid {}_{12}(\bar{j} \cdot 0) \\ \xrightarrow{\bar{j}@12} {}_{11}(f^2 \cdot 0) \upharpoonright ({}_{11}, \{j@13\})LoOut \end{array} & [\mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_1] \\
\begin{array}{l} \xrightarrow{f@11} {}_{12}(f \cdot 0) \upharpoonright ({}_{12}, \{j@13\})LoOut \\ \xrightarrow{f@12} ({}_{13}, \{j@13\})LoOut \end{array} & [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_6] \\
\begin{array}{l} \xrightarrow{j@13} {}_{14}(c^2 \cdot 0) \\ \xrightarrow{c@14} {}_{15}(c \cdot 0) \end{array} & [\mathbf{Con} \text{ and } \mathbf{Act}_2] \\
\begin{array}{l} \xrightarrow{c@15} 0 \end{array} & [\mathbf{Act}_1] \\
\begin{array}{l} \xrightarrow{c@15} 0 \end{array} & [\mathbf{Act}_1 \text{ and } (6)]
\end{array}$$

Initially only the I/O Processor's agents can make progress, and the *HiIn* agent is given preference. Note that while the *HiIn* agent is performing actions locally, time does not advance in the parallel *CPU* agent. At time 2 the *HiIn* agent performs action $\bar{k}$ which sends message k . This results in event $k@3$ being added to the context of the *CPU* agent, which enables the *HiProc* agent. In the particular interleaving shown above, event $k@3$ in the *CPU* agent occurs *after* event $a@4$ in the *IOP* agent. This is permissible because the true-concurrency timing model requires that only causally-related events must appear in chronological sequence [1,10]. However, when all the timestamped events are displayed as in Figure 4, it can be seen that the overall derivation is causally consistent. After the asynchronous send action $\bar{i}$ occurs at time 5, the *LoProc* agent is enabled, but it does not accept this message and begin executing until time 9, because the *HiProc* agent is preempting it. This separation between sends and receives demonstrates the need for asynchronous communication to model multi-processor precedence constraints.

5 Discussion and Future Work

The formalism presented above can be enhanced in a number of ways. For instance, rather than assuming each task always requires its worst-case quanta of computation time, we can use summation to define an abbreviation which allows tasks to have both minimum and maximum execution times [3,10].

We assumed static-priority scheduling above since this is the default in programming languages such as Ada. However, we could easily adopt Ben-Abdallah et al.'s approach for modelling dynamic-priority scheduling in which each action incorporates an expression defining its current dynamic priority [3].

We also assumed that interprocessor precedence constraints were enforced via asynchronous message-passing, but in some applications it may be preferable to use interrupts. This can be modelled by changing our asynchronous communication mechanism to treat receives as very high-priority actions that always occur as soon as they can.

More significantly, our algebra must be completed by defining its notion of weak equivalence and a full set of equational laws [15]. Many such laws are straightforward modifications of their standard CCS counterparts, but entirely new laws are needed for the scheduling operator and asynchronous communication. This work is ongoing.

In practice, the full potential of a formalism is achieved only when support tools become available for it. Model checking timed automata has been the subject of much research, but has been hindered by semantics in which parallel transitions are linked by the passage of time [16]. The true concurrency model used here allows non-causally-related transitions to occur independently, and even out of their chronological sequence [1,10], and is thus particularly amenable to efficient partial order reduction model checking techniques.

6 Conclusion

We have shown how multi-tasking behaviour can be modelled in a process algebra. This was done by starting with a timed, causal CCS variant and then introducing both an asymmetric interleaving operator, to model intraprocessor scheduling decisions, and asynchronous communication, to model interprocessor precedence constraints. The resulting formalism was shown to be capable of representing a wide variety of scheduling behaviours.

Acknowledgements Thanks to Antonio Cerone, Padmanabhan Krishnan and the anonymous referees for their comments. This work began while the author was visiting the University of York's Real-Time Systems Research group, courtesy of a University of Queensland Travel Award. Thanks to Andy Wellings for hosting the visit and Iain Bate for helpful discussions. This research was funded by the Information Technology Division of the Australian Defence Science and Technology Organisation.

References

1. L. Aceto and D. Murphy. Timing and causality in process algebra. *Acta Informatica*, 33:317–350, 1996.
2. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
3. H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie. A process algebraic approach to the schedulability analysis of real-time systems. *Real-Time Systems*, 15:189–219, 1998.

4. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer-Verlag, 1998.
5. L. Breveglieri, S. Crespi-Reghizzi, and A. Cherubini. Modeling operating systems schedulers with multi-stack-queue grammars. In G. Ciobanu and G. Păun, editors, *Fundamentals of Computation Theory (FCT'99)*, volume 1684 of *Lecture Notes in Computer Science*, pages 161–172. Springer-Verlag, 1999.
6. G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.
7. J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transaction on Software Engineering*, 22(7):461–483, July 1996.
8. J. S. Dong, N. Fulton, L. Zucconi, and J. Colton. Formalising process scheduling requirements for an aircraft operational flight program. In *Proc. IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 161–169. IEEE Press, November 1997.
9. J. D. G. Falardeau. Schedulability analysis in rate monotonic based systems with application to the CF-188. Master's thesis, Department of Electrical and Computer Engineering, Royal Military College of Canada, May 1994.
10. C. J. Fidge and J. J. Žic. An expressive real-time CCS. In *Proc. Second Australasian Conference on Parallel and Real-Time Systems (PART'95)*, pages 365–372, Fremantle, September 1995.
11. P.-A. Hsiung, F. Wang, and Y.-S. Kuo. Scheduling system verification. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, 1999.
12. D. M. Jackson. Experiences in embedded scheduling. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 445–464. Springer-Verlag, 1996.
13. J. Jacky. Analyzing a real-time program in Z. In *Proc. Z User's Meeting (ZUM'98)*, 1998.
14. Z. Liu, M. Joseph, and T. Janowski. Verification of schedulability for real-time programs. *Formal Aspects of Computing*, 7(5):510–532, 1995.
15. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
16. M. Minea. Partial order reduction for model checking of timed automata. In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99: Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 431–446. Springer-Verlag, 1999.
17. R. van Glabbeek and P. Rittgen. Scheduling algebra. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 1999.
18. R. J. van Glabbeek. The meaning of negative premises in transition system specifications II. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium*, volume 1099 of *Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 1996. Extended abstract.
19. Z. Yuhua and Z. Chaochen. A formal proof of the deadline driven scheduler. In H. Langmaack, W.-P. de Roever, and J. Vytupil, editors, *Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 756–775. Springer-Verlag, 1994.

A Causal Semantics for Timed Default Concurrent Constraint Programming^{*}

Simone Tini and Andrea Maggiolo-Schettini

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125, Pisa, Italy
{tini,maggiolo}@di.unipi.it

Abstract. We define a *causal semantics* for Timed Default Concurrent Constraint Programming which permits associating each output with its cause. We show by an example how this information can simplify program debugging. We prove that the classic operational semantics of the language can be recovered from our causal semantics, and we prove that our causal semantics and the causal semantics of Concurrent Constraint Programming coincide for programs belonging to both languages.

1 Introduction

Timed Default Concurrent Constraint Programming [17,18] (**tdccp**) is a *synchronous model* of concurrent computations in *reactive systems*.

We recall that, according to [11,8], a computation system is reactive if it continuously reacts to stimuli from the external environment by producing responses within temporal bounds fixed by it. Synchronous models are based on the *synchronous hypothesis* [2] which states that reactions of a system are instantaneous, so that outputs from the system are available as soon as inputs from the environment are. This is indeed an abstraction, and amounts to requiring that the environment be invariant w.r.t. the system during reactions.

tdccp extends Concurrent Constraint Programming (**ccp**) [15] with constructs to deal with defaults for negative information and discrete time. **ccp** assumes a notion of computation as deduction over systems of partial information. Information accumulates monotonically in a distributed context: a multiset of *agents* cooperate to produce *constraints* on *shared variables*. A set of primitive constraints (*tokens*) are assumed to specify possibly partial information on the values of a set of variables. Tokens are equipped with an *entailment relation* $\vdash$ and a *conjunction operation* $\wedge$: $a \vdash b$ holds if the information given by b follows from the information given by a ; $a \wedge b$ gives information given by both a and b .

A **ccp** program consists of a multiset of agents which cooperate by posting tokens in a shared *store* and by querying the store about tokens being entailed. So, the agent “**tell** a ” posts the token a in the store and the agent “**if** a **then** A ”

^{*} Research partially supported by ESPRIT Working Group COTIC, Project Number 23677, and by MURST progetto TOSCA.

behaves as the agent A when the store contains tokens entailing a . **ccp** cooperating agents are not synchronized, in the sense that each agent proceeds at its own rate. The construct **if _ then _** may be used to achieve synchronization.

On the contrary, **tdccp** cooperating agents are perfectly synchronized, in the sense that they share the same clock. **tdccp** assumes a discrete notion of time (i.e. time is a sequence of instants) and offers a construct, **next**, to sequentialize interaction between agents and the store: “**next** A ” will behave as A at the next instant of time. According to the synchronous hypothesis, agents post tokens in the store and query the store without consuming time, while **next** consumes exactly one unit of time. A **tdccp** program has a discrete behavior: at each instant it reacts to an input, namely to a set of tokens posted in the store by the external environment. The reaction implies an accumulation of tokens in the store, so that the resulting store is interpreted as the response of the program. The store is refreshed between any instant and the subsequent one. Finally, **tdccp** agents may query the store also about the nonvalidity of tokens: “**if** a **else** A ” behaves as A if the store does not entail a at the current instant.

According to the operational semantics of [17], an observer of a **tdccp** program $\mathcal{P}$ can observe sequences of pairs of sets of tokens $(i_1, o_1), \dots, (i_n, o_n), \dots$, with i_n the set of tokens posted in the store by the environment at the n^{th} instant and o_n the set of tokens produced by $\mathcal{P}$ at the same instant. The observer cannot associate to each token $a \in o_n$ its *causes*, i.e. the subset of tokens produced either by the environment or by $\mathcal{P}$ itself up to the n^{th} instant which are sufficient to have the token a as a response of $\mathcal{P}$ at the n^{th} instant. (We are considering a notion of causality different w.r.t. the concept of causality in synchronous programs which is at the origin of the known “paradoxes of causality”.)

Example 1. Let A be the agent $[A_0, A_1]$, where $A_i \equiv \text{if } a_i \text{ then next tell } b_i$ (A_0 and A_1 run in parallel). If the environment produces both a_0 and a_1 at the first instant, the observer observes b_0 and b_1 at the second instant, but it is unable to deduce that b_0 is caused by a_0 only and that b_1 is caused by a_1 only.

Our aim is to endow **tdccp** with a *causal semantics*, consistent with the semantics of [17], which highlights causes of tokens produced by programs. We quote at least two applications in which this additive information may be used:

- *Modeling application-level causality.* Constraint based languages have been extensively used to model physical systems (see [6]). System components are modeled as transducers which accept input and control signals, operate in a given set of modes and produce output signals. In order to *schedule* such a system one must determine input and control signals which must be supplied to the system to cause a certain output. To do this one is interested in knowing the minimal cause of the output considered, namely the minimal set of input and control signals causing a certain answer of the system.
- *Observer monitoring verification.* This technique has been proposed in [10,9] for the verification of safety properties of reactive systems. (Note that, as argued in [14], most interesting properties of reactive systems are safety

properties.) A safety property ϕ , expressing logic relations over inputs and outputs of an agent A , is mapped to an agent Ω_ϕ which runs in parallel with A and *observes* the behavior of A by reading inputs and outputs of A . So, Ω_ϕ is able to detect when A violates ϕ , and, in this case, it produces an “alarm” token α . The model checking problem for the agent A and the property ϕ is reduced to checking whether the parallel execution of A and Ω_ϕ reaches a state where α can be produced (note that **tdccp** agents are always finite state systems, as it has been proved in [17,18]). Now, given a computation trace where the alarm α appears, if we can observe the causes of α then we can isolate the part of A responsible of the violation of ϕ .

We recall that a causal semantics for **ccp** has been given in [7] as a finer semantics w.r.t. that of [16]. According to [16], the observer of a **ccp** program observes the final store resulting from its execution, namely the set of tokens produced by the program. In [7] the observable final store consists of a set of *contexted tokens* of the form a^b , where the *context* b contains exactly the token causing a , namely the token causing the execution of an agent **tell** a .

Now, if we consider the construct “**if** _ **else** _”, tokens may be caused by the absence of other tokens. Moreover, if we consider the construct “**next** _”, tokens produced by a **tdccp** program at the n^{th} instant may be caused by the presence or by the absence of tokens at instants $i \leq n$. Therefore, we will consider contexted tokens of the form $a^{\{(D_0,0),\dots,(D_{n-1},n-1)\}}$, with D_i a set of the form $\{a_{i,1}^+, \dots, a_{i,h}^+, a_{i,h+1}^-, \dots, a_{i,k}^-\}$, representing that a is produced at the n^{th} instant because the store, at the $(n-i)^{th}$ instant, entails every token $a_{i,j}$ such that $1 \leq j \leq h$ and does not entail any token $a_{i,j}$ such that $h+1 \leq j \leq k$.

In section 2 we recall **tdccp**. In section 3 we give our causal semantics and we show by an example how it can be used to improve the verification phase of programs. In section 4 we prove that our causal semantics is consistent with the operational semantics of [17], and we prove that our causal semantics and the causal semantics of [7] coincide for programs belonging to both **tdccp** and **ccp**.

2 An Overview of **tdccp**

A constraint system is a system of partial information, consisting of a set $\mathcal{D}$ of tokens (first-order formulas over an infinite set of variables Var), closed under conjunction, and a decidable inference relation (logical entailment) $\vdash$ relating tokens to tokens. The set $\mathcal{D}$ is ranged over by $a, b, \dots$. The entailment relation induces, through symmetric closure, a logical equivalence relation which is denoted with $\approx$. More precisely, $a \approx b$ if and only if $a \vdash b$ and $b \vdash a$.

Definition 1. A constraint system is a structure $\langle \mathcal{D}, \wedge, \vdash, Var \rangle$ s.t. $\mathcal{D}$ is a set of first order formulas over Var , $\wedge : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is a total function, and $\vdash \subseteq \mathcal{D} \times \mathcal{D}$ is a decidable relation s.t.: 1) $a \vdash a$; 2) $a \vdash a'$ and $a' \wedge a'' \vdash b$ implies $a \wedge a'' \vdash b$; 3) $a \wedge b \vdash a$; 4) $a \wedge b \vdash b$; 5) $a \vdash b_1$ and $a \vdash b_2$ implies $a \vdash b_1 \wedge b_2$.

Sets of tokens closed under $\vdash$ are called *constraints*.

We assume tokens $true, false \in \mathcal{D}$ with $a \vdash true$ and $false \vdash a$ for every $a \in \mathcal{D}$. Token $true$ can be interpreted as the conjunction of tokens $a_1, \dots, a_n$, where $n = 0$, and $false$ can be interpreted as the conjunction of all tokens in $\mathcal{D}$.

Example 2. The constraint system *Gentzen* has been used in [17] to represent binary signals. Atomic propositions $X, Y, Z, \dots$ are tokens, where X, Y, Z range over variables. Every variable X is associated with a binary signal, and the token X represents the presence of this signal. The entailment relation is such that $a_1 \wedge \dots \wedge a_n \vdash a$ if and only if $a = a_i$ for some $1 \leq i \leq n$.

In the following, we assume that we are working within a given constraint system $\langle \mathcal{D}, \wedge, \vdash, Var \rangle$. We assume the syntax for **tdccp** given in [17]:

$$A ::= \text{skip} \mid \text{tell } a \mid \text{if } a \text{ then } A \mid \text{if } a \text{ else } A \mid \text{next } A \mid [A, A] \mid \text{rec } P. A \mid P$$

where a, A and P range over tokens, agents and *recursion variables*, respectively (**rec** is denoted by “ μ ” in [17]). We identify agents and programs and we denote by $\equiv$ the syntactic identity over agents.

Agent **skip** does nothing at each instant. Agent **tell** a posts the token a in the store and then terminates, namely it behaves as **skip**. Both **if** a **then** A and **if** a **else** A query the store about the validity of token a . If the store entails a then the first agent behaves as A else it terminates. Symmetrically, if the store entails a then the second agent terminates else it behaves as A . Agent **next** A will behave as A at the next instant. Agent $[A_1, A_2]$ is the synchronous parallel composition of A_1 and A_2 . Construct **rec** is the classical recursion construct. To be sure that at each instant the computation is finite, it is required that recursion be *guarded*, namely that recursion variables appear in bodies of **next** $\dots$.

Cooperating agents can communicate, in the sense that tokens posted in the store by an agent are immediately available to agents running in parallel. Instantaneous communications may give rise to *paradoxes of causality* which originate either non-reactivity (i.e. the inability to react) or nondeterminism.

As an example, the agent **if** a **else** **tell** a is non-reactive if the store does not entail a . In fact, the token a is posted in the store if and only if it is non-entailed by the store at the same instant, so no reaction is admissible. The agent **[if** a **else** **tell** b , **if** b **else** **tell** a **]** is nondeterministic if the store entails neither a nor b . In fact, either the left branch produces b and the right branch terminates without producing a , or the right branch produces a and the left branch terminates without producing b .

Both reactivity and determinism are decidable (see [17] or Prop. 3, Sec. 4). Only reactive and deterministic agents are accepted in [17]. (Note that [17] does not require the stronger property of *constructiveness* recently defined in [1].)

We recall now the operational semantics of [17]. We begin with describing the behavior of agents at an instant, in terms of instantaneous transitions between configurations. A configuration is a pair of multisets of agents: the agents currently active and the agents that will be executed at the next instant. Multisets

of agents are ranged over by $\Gamma, \Delta, \dots$. The construct $[\_, \_]$ is assumed to be commutative and associative, and a multiset $\Gamma \equiv A_1, \dots, A_n$ denotes the parallel composition of $A_1, \dots, A_n$. Moreover, **skip** denotes $[A_1, \dots, A_n]$ if $n = 0$.

For any multiset of agents Γ , let $\sigma(\Gamma)$ denote the token $\bigwedge \{a \mid \mathbf{tell} \ a \in \Gamma\}$.

$$\begin{array}{c}
\frac{\sigma(\Gamma) \vdash a}{((\Gamma, \mathbf{if} \ a \ \mathbf{then} \ A), \Delta) \rightarrow_b ((\Gamma, A), \Delta)} \qquad \frac{b \not\vdash a}{((\Gamma, \mathbf{if} \ a \ \mathbf{else} \ A), \Delta) \rightarrow_b ((\Gamma, A), \Delta)} \\
((\Gamma, [A, B]), \Delta) \rightarrow_b ((\Gamma, A, B), \Delta) \qquad ((\Gamma, \mathbf{skip}), \Delta) \rightarrow_b (\Gamma, \Delta) \\
((\Gamma, \mathbf{rec} \ P.A), \Delta) \rightarrow_b ((\Gamma, A[\mathbf{rec} \ P.A/P]), \Delta) \quad ((\Gamma, \mathbf{next} \ A), \Delta) \rightarrow_b (\Gamma, (A, \Delta))
\end{array}$$

Table 1. The relation $\rightarrow_b$.

We define binary transition relations on configurations, denoted as $\rightarrow_b$, where b is the final guessed store. Relation $\rightarrow_b$ is defined in Table 1. The output of executing agent A on input a is determined by means of function $r_o(A)$ s.t.:

$$r_o(A)(a) = \{b \in \mathcal{D} \mid \exists b'. ([A, \mathbf{tell} \ a], \emptyset) \rightarrow_{b'}^* (\Gamma, \Delta) \not\vdash_{b'}, b' = \sigma(\Gamma), b \approx b'\}$$

The operational semantics described in Table 1 can be used to compute the result of running an agent in a given store only if the *final store* is known beforehand. The nondeterminism can be made effective by backtracking (see [17]).

Now, we describe the temporal behavior of agents in terms of a transition relation $\hookrightarrow$ over agents:

$$\frac{\exists b \in \mathcal{D} \mid (A, \emptyset) \rightarrow_b^* (\Gamma', \Delta) \not\vdash_b, \sigma(\Gamma') = b, \Delta \equiv A_1, \dots, A_n}{A \hookrightarrow [A_1, \dots, A_n]}$$

The output of executing agent A on a sequence of inputs $a_1, \dots, a_n$ is determined by means of function $rt_o(A)$ such that:

$$rt_o(A)(a_1, \dots, a_n) = \{b_1, \dots, b_n \mid A_0 \equiv A, r_o(A_i)(a_i) = b_i, [A_i, \mathbf{tell} \ a_i] \hookrightarrow A_{i+1}\}$$

An agent A is reactive and deterministic if and only if for every token a , there exists a token b such that $b \in rt_o(A)(a)$, and b is unique modulo $\approx$.

Example 3. Let A be the agent $[\mathbf{if} \ a \ \mathbf{else} \ \mathbf{tell} \ b, \mathbf{if} \ b \ \mathbf{else} \ \mathbf{tell} \ a]$. Since $b \in rt_o(A)(\mathbf{true})$ and $a \in rt_o(A)(\mathbf{true})$, the agent A is nondeterministic.

We conclude this section with an example of a bus arbiter specified in **tdccp**. This example was originally treated in [12].

Example 4. Let us assume a bus shared by users $U_1, \dots, U_n$. User U_i may request to use the bus by sending a request signal r_i . In this case, U_i is allowed to use the bus only if it receives an acknowledgment signal a_i .

The arbiter is implemented as a ring of n cells, $C_1, \dots, C_n$. Each user is connected to a cell which receives requests through input signal *Req_in* and acknowledges requests through output signal *Ack_out*. A token ring is introduced to ensure weak fairness. Cell C_i receives the *token* through signal *T_in* at j^{th} instant, for j such that $j = i + kn$, and sends the token through signal *T_out* at instant $j + 1$. So, every cell “stores” the token for one instant. When receiving the token *T_in*, a cell is allowed to acknowledge a request *Req_in*, since it has higher priority w.r.t. other cells. When receiving the token *T_in*, if a cell does not receive the request *Req_in* then it passes a *grant* around the ring through output signal *G_out* immediately. A cell is allowed either to acknowledge a request or to pass the grant also when it receives the grant, namely when it receives the input signal *G_in*. So, the grant allows cells to acknowledge requests without having the token. Introducing the grant is needed to avoid that, at some cycle, there are requests from users and none of them is satisfied.

Let us assume the constraint system *Gentzen*. Agents *Cell* and *ring₃* below specify a cell and the ring of three cells, respectively. Component **tell** t_1 in *ring₃* gives the token to C_1 at the first instant.

Cell ::= **rec** P . ([**if** *T_in* **then** **if** *Req_in* **then** **tell** *Ack_out*,
 if *G_in* **then** **if** *Req_in* **then** **tell** *Ack_out*,
 if *T_in* **then** **if** *Req_in* **else** **tell** *G_out*,
 if *G_in* **then** **if** *Req_in* **else** **tell** *G_out*,
 if *T_in* **then** **next** **tell** *T_out*,
 next P]).

ring₃ ::= [**tell** t_1 ,
 $Cell[r_1/Req\_in, t_1/T\_in, g_1/G\_in, t_2/T\_out, g_2/G\_out, a_1/Ack\_out]$,
 $Cell[r_2/Req\_in, t_2/T\_in, g_2/G\_in, t_3/T\_out, g_3/G\_out, a_2/Ack\_out]$,
 $Cell[r_3/Req\_in, t_3/T\_in, g_3/G\_in, t_1/T\_out, g_1/G\_out, a_3/Ack\_out]$]

The arbiter *ring₃* does not satisfy the fairness property ϕ : “if user U_3 requests the bus at two subsequent instants, then user U_2 cannot obtain the bus at both instants”. In fact, if U_1 does not request the bus at the first instant, then U_2 can obtain the bus at the first and second instant. To apply the observer monitoring, we map ϕ to the **tdccp** agent Ω_ϕ :

rec P . [**if** r_3 **then** **if** a_2 **then** **next** (**if** r_3 **then** **if** a_2 **then** **tell** α), **next** P]

and we consider the behavior of $[ring_3, \Omega_\phi]$. According to the classical semantics of [17], the observer of $[ring_3, \Omega_\phi]$ can observe the pair of tokens $(r_2 \wedge r_3, a_2 \wedge t_1 \wedge g_2)$ at the first instant and the pair of tokens $(r_2 \wedge r_3, a_2 \wedge t_2 \wedge \alpha)$ at the second instant (i.e. $rt_o([ring_3, \Omega_\phi])(r_2 \wedge r_3, r_2 \wedge r_3) = a_2 \wedge t_1 \wedge g_2, a_2 \wedge t_2 \wedge \alpha$) and deduce that ϕ is violated by *ring₃*. The observer cannot deduce the reasons for which *ring₃* violates ϕ . We will see that our causal semantics supplies the observer with this information.

3 The Causal Semantics of $\mathbf{tdccp}$

In this section we present our causal semantics for $\mathbf{tdccp}$. Technically, we begin with giving a Structural Operational Semantics (SOS) [13] in terms of a proved transition system (PTS) [4,3], namely a labeled transition system with transition labels encoding transition deduction trees (proofs). PTS states correspond to $\mathbf{tdccp}$ agents, PTS transitions correspond to agent reactions, and PTS labels carry detailed information on reactions. In particular, labels carry sufficient information to establish which component of an agent posts a given token in the store. This information permits to infer causality between tokens.

Given an agent A , we denote with $[A]_{pts}$ the portion of PTS reachable from state A , and with $[A]_{pt}$ the tree obtained by unfolding $[A]_{pts}$. Following [5], we call $[A]_{pt}$ a *proved tree*. Then, we define a *causal observation function* $\mathcal{C}$ that relabels a given proved tree $[A]_{pt}$ into a *causal tree* $\mathcal{C}([A]_{pt})$. Labels of this causal tree contain contexted tokens highlighting causality between tokens. We consider $\mathcal{C}([A]_{pt})$ as the causal semantics of A .

Before giving the PTS, we introduce some notations.

Let $\mathcal{D}^\pm$ be the set $\{a^+, a^- \mid a \in \mathcal{D}\}$ and let γ range over $\mathcal{D}^\pm$. Given $a \in \mathcal{D}$, the symbol a^+ (resp. a^-) denotes the validity (resp. nonvalidity) of a . We assume a function $\neg: \mathcal{D}^\pm \rightarrow \mathcal{D}^\pm$ such that $\neg\gamma = a^-$ if $\gamma = a^+$, and $\neg\gamma = a^+$ if $\gamma = a^-$.

An *event* D is a subset of $\mathcal{D}^\pm$ s.t. there do not exist symbols $a_1^+, \dots, a_n^+, a^- \in D$ s.t. $a_1 \wedge \dots \wedge a_n \vdash a$. It is interpreted as the assumption that the store entails (resp. does not entail) every token a s.t. $a^+ \in D$ (resp. $a^- \in D$).

Two events D and D' are *consistent*, written $D \uparrow D'$, iff $D \cup D'$ is an event.

Definition 2. Given a string $\vartheta \in (\mathcal{D}^\pm \cup \{\|_0, \|_1\})^*$ and a token $a \in \mathcal{D}$, ϑa is a proof term with ϑ as proof and a as action.

A proof term ϑa refers to the action, performed by a $\mathbf{tell} \ a$ agent, of posting the token a in the store. The proof ϑ highlights the syntactic context in which this agent is embedded. The symbol $\|_0$ (resp. $\|_1$) in ϑ means that $\mathbf{tell} \ a$ is in the left (resp. right) side of a $[-, \_]$. The symbol b^+ (resp. b^-) in ϑ means that $\mathbf{tell} \ a$ is in the body of a $\mathbf{if} \ b \ \mathbf{then} \ \_$ (resp. $\mathbf{if} \ b \ \mathbf{else} \ \_$). So, a proof term reflects also instantaneous causality between tokens.

Given a proof ϑ , we denote with $|\vartheta|$ the set $\{\gamma \in \mathcal{D}^\pm \mid \gamma \text{ appears in } \vartheta\}$. We will denote with ϵ the empty string.

Definition 3. A label is a pair $l = \langle D_l, \mathcal{E}_l \rangle$, where D_l is an event and $\mathcal{E}_l$ is a set of proof terms such that $\bigcup_{\vartheta a \in \mathcal{E}_l} |\vartheta| \cup \{a^+\} = D_l$.

We will denote with $\mathcal{L}$ the set of labels as in Def. 3. Let us assume a label $l \in \mathcal{L}$ of a PTS transition $A \xrightarrow{l} A'$. The event D_l describes the information in the store at the end of the reaction corresponding to $A \xrightarrow{l} A'$. Proof terms in $\mathcal{E}_l$ refer to actions that are performed by $\mathbf{tell}$ agents in the body of A . Given a token $a \in D_l$, either a is posted in the store by the environment, or $\vartheta a \in \mathcal{E}_l$ for some ϑ and a is produced by a $\mathbf{tell}$ agent in the body of A .

The PTS is defined by the transition system specification in Table 2.

$\text{skip} \xrightarrow{\langle \emptyset, \emptyset \rangle} \text{skip} \quad (\text{skip})$	$\text{tell } a \xrightarrow{\langle \{a^+\}, \{\epsilon a\} \rangle} \text{skip} \quad (\text{tell})$
$\frac{A \xrightarrow{l} A' \quad \{a^+\} \uparrow D_l}{\text{if } a \text{ then } A \xrightarrow{a^+(l)} A'} \quad (\text{then\_0})$	$\text{if } a \text{ then } A \xrightarrow{\langle \{a^-\}, \{a^- \text{true}\} \rangle} \text{skip} \quad (\text{then\_1})$
$\frac{A \xrightarrow{l} A' \quad \{a^-\} \uparrow D_l}{\text{if } a \text{ else } A \xrightarrow{a^-(l)} A'} \quad (\text{else\_0})$	$\text{if } a \text{ else } A \xrightarrow{\langle \{a^+\}, \{a^+ \text{true}\} \rangle} \text{skip} \quad (\text{else\_1})$
$\frac{A[\text{rec } P. A/P] \xrightarrow{l} A'}{\text{rec } P. A \xrightarrow{l} A'} \quad (\text{rec})$	$\frac{A_0 \xrightarrow{l_0} A'_0 \quad A_1 \xrightarrow{l_1} A'_1 \quad D_{l_0} \uparrow D_{l_1}}{[A_0, A_1] \xrightarrow{l_0 \otimes l_1} [A'_0, A'_1]} \quad (\text{par})$
	$\text{next } A \xrightarrow{\langle \{\text{true}^+\}, \{\epsilon \text{true}\} \rangle} A \quad (\text{next})$

Table 2. The proved transition system for **tdccp**.

Rule *skip* states that **skip** does nothing at every instant.

Rule *tell* states that **tell** *a* posts the token *a* in the store and will behave as **skip**. The proof term ϵa expresses that no condition on the store is required to produce *a*. This reflects that **tell** *a* does not appear in bodies of statements **if** _ **then** _ and **if** _ **else** _.

Given $l \in \mathcal{L}$ and $\gamma \in \mathcal{D}_+^+$ such that $D_l \uparrow \{\gamma\}$, we denote with $\gamma(l)$ the label:

$$\gamma(l) = \langle D_l \cup \{\gamma\}, \{\gamma \vartheta a \mid \vartheta a \in \mathcal{E}_l\} \rangle$$

Let us consider the agent **if** *a* **then** *A*. Rule *then_0* states that if the store entails *a* then **if** *a* **then** *A* reacts as *A*. The label $a^+(l)$ highlights that actions performed by agents in the body of *A* require that the store entails *a*. Rule *then_1* states that if the store does not entail *a* then **if** *a* **then** *A* terminates, namely it posts the token *true* in the store and it will behave as **skip**.

Example 5. Let *A* be the agent **if** *a* **then** **tell** *b*. We have $A \xrightarrow{l_1} \text{skip}$, $l_1 = \langle \{a^+, b^+\}, \{a^+b\} \rangle$, and $A \xrightarrow{l_2} \text{skip}$, $l_2 = \langle \{a^-\}, \{a^- \text{true}\} \rangle$.

Rules *else_0* and *else_1* are analogous to *then_0* and *then_1*, respectively.

Rule *next* states that **next** *A* will behave as *A* at the next instant. At the current instant only *true* is posted in the store.

Rule *rec* is a standard recursion rule.

Rule *par* states that $[A_0, A_1]$ performs both reactions of A_0 and A_1 . Given labels l_0 and l_1 such that $D_{l_0} \uparrow D_{l_1}$, let $l_0 \otimes l_1$ be the following label:

$$l_0 \otimes l_1 = \langle D_{l_0} \cup D_{l_1}, \{\|_0 \vartheta a \mid \vartheta a \in \mathcal{E}_{l_0}\} \cup \{\|_1 \vartheta a \mid \vartheta a \in \mathcal{E}_{l_1}\} \rangle$$

which reflects that actions performed by agents in the body of A_0 (resp. A_1) are performed by agents in the left (resp. right) branch of $[\_, \_]$. This information permits to relate tokens produced at the n^{th} instant with their causes at instants up to n . In fact, given transitions $A_1 \xrightarrow{l_1} \dots \xrightarrow{l_n} A_{n+1}$, if $\vartheta a \in \mathcal{E}_{l_n}$, $\vartheta' \text{true} \in$

$\mathcal{E}_{l_{n-k}}$ and $\vartheta' \vdash \{\|_0, \|_1\}$ is a prefix of $\vartheta \vdash \{\|_0, \|_1\}$ then **tell** a corresponding to ϑa is executed at the n^{th} instant because each b with $b^+ \in \vartheta'$ (resp. $b^- \in \vartheta'$) was entailed (resp. nonentailed) by the store at the $(n-k)^{th}$ instant.

Example 6. Let A be as in Example 1. We have $A \xrightarrow{l} [\mathbf{tell} \ b_0, \mathbf{tell} \ b_1] \xrightarrow{l'} [\mathbf{skip}, \mathbf{skip}]$, $l = \langle \{a_0^+, a_1^+, true^+\}, \{\|_0 a_0^+ true, \|_1 a_1^+ true\} \rangle$, $l' = \langle \{b_0^+, b_1^+\}, \{\|_0 b_0, \|_1 b_1\} \rangle$. We deduce that b_0 (resp. b_1) is produced at the second instant because a_0 (resp. a_1) was valid at the first one from the fact that $\|_0 a_0^+ \vdash \{\|_0, \|_1\}$ is a prefix of $\|_0 \vdash \{\|_0, \|_1\}$ and $\|_1 a_1^+ \vdash \{\|_0, \|_1\}$ is a prefix of $\|_1 \vdash \{\|_0, \|_1\}$.

We define now the causal observation function $\mathcal{C}$.

Definition 4. *The causal observation function $\mathcal{C}$ relabels any computation*

$A_1 \xrightarrow{l_1} \dots \xrightarrow{l_n} A_{n+1} \dots$ of $[A_1]_{pt}$ as $A_1 \xrightarrow{\mathcal{C}(l_1)} \dots \xrightarrow{\mathcal{C}(l_n)} A_{n+1} \dots$, where, for each $\vartheta a \in \mathcal{E}_{l_n}$, we have $a^K \in \mathcal{C}(l_n)$, with K the set of pairs (D, k) such that either $k = 0$ and $D = |\vartheta|$, or $k > 0$ and $D = |\vartheta'|$ for some $\vartheta' true \in \mathcal{E}_{l_{n-k}}$ such that $\vartheta' \vdash \{\|_0, \|_1\}$ is a prefix of $\vartheta \vdash \{\|_0, \|_1\}$.

Given a computation $A_1 \xrightarrow{l_1} \dots \xrightarrow{l_n} A_{n+1}$ of $[A_1]_{pt}$, for each action of posting a in the store performed by an agent **tell** a in the body of A_n , a contexted token a^K is in $\mathcal{C}(l_n)$. Now, for each $0 \leq k \leq n-1$, K contains the pair (D, k) if among the causes of the execution of the considered agent there is the validity (resp. nonvalidity) of every token b such that $b^+ \in D$ (resp. $b^- \in D$) at the $(n-k)^{th}$ instant of time. So, k is a backward pointer in the causal tree.

Note that causal trees highlight only direct causes of production of tokens. Indirect causes can be obtained by transitivity.

Example 7. Given the computation $A \xrightarrow{l} [\mathbf{tell} \ b_0, \mathbf{tell} \ b_1] \xrightarrow{l'} [\mathbf{skip}, \mathbf{skip}]$ of Example 6, we have that $\mathcal{C}(l) = \{true^{\{(\{a_0^+\}, 0)\}}, true^{\{(\{a_1^+\}, 0)\}}\}$ and $\mathcal{C}(l') = \{b_0^{\{(\{a_0^+\}, 1), (\emptyset, 0)\}}, b_1^{\{(\{a_1^+\}, 1), (\emptyset, 0)\}}\}$.

We say that a token a *triggers* a transition $A \xrightarrow{l} A'$ of a causal tree if this transition represents a reactions of A to a store entailing a . Formally, given $A \xrightarrow{l} A'$ in $\mathcal{C}([A]_{pt})$ such that $l = \{b_1^{K_1}, \dots, b_m^{K_m}\}$, $(D_i, 0) \in K_i$, we say that a triggers $A \xrightarrow{l} A'$ if and only if there exists a sequence of indexes $i_1, \dots, i_m$ such that $\{i_1, \dots, i_m\} = \{1, \dots, m\}$ and a sequence of tokens $a_1, \dots, a_{m+1}$ such that:

- $a_1 = a$, $a_{j+1} = a_j \wedge b_{i_j}$ for $1 \leq j \leq m$;
- $a_j \vdash \bigwedge_{d^+ \in D_{i_j}} d$;
- $a_{m+1} \not\vdash d$ for every $d^- \in \bigcup_{1 \leq i \leq m} D_i$.

Let us return to the agent $[ring_3, \Omega_\phi]$ of Example 4. We observe the computation trace $[ring_3, \Omega_\phi] \xrightarrow{l} \xrightarrow{l'}$ in $\mathcal{C}([ring_3, \Omega_\phi]_{pt})$, where the contexted token $\alpha^{\{(\{a_2^+, r_3^+\}, 0), (\{a_2^+, r_3^+\}, 1)\}}$ in l' shows that $ring_3$ violates ϕ because both a_2 and r_3 are posted in the store at the first and second instant. The tokens $t_1^{\{(\emptyset, 0)\}}$,

$g_2^{\{(\{t_1^+, r_1^-\}, 0)\}}$, $a_2^{\{(\{g_2^+, r_2^+\}, 0)\}}$ in l reveal the reason for which a_2 is produced at the first instant: a_2 is caused by g_2 , namely C_2 can give the bus to U_2 because it receives the grant, and t_1 is among the causes of g_2 , namely C_1 can send the grant to C_2 because it has the token. The tokens $a_2^{\{(\{t_2^+, r_2^+\}, 0), (\emptyset, 1)\}}$, $t_2^{\{(\{t_1^+\}, 1), (\emptyset, 0)\}} \in l'$ reveal the reason for which a_2 is produced at the second instant: a_2 is caused by t_2 , namely C_2 can give the bus to U_2 because it receives the token, and C_2 receives the token because C_1 had it at the previous instant.

So, $\mathcal{C}([ring_3, \Omega_\phi]_{pt})$ highlights information on interactions among components of $ring_3$ that permits to deduce the reason for which Ω_ϕ produces α .

Now, in [17,18] a compiler is given which maps agents to Finite State Automata (FSA). As suggested in [10], one may exploit such a compiler for verification purposes. In fact, in order to check whether an agent A violates a property ϕ , one can compile $[A, \Omega_\phi]$ into an FSA and check whether the token α can be produced in a state reachable by the FSA. Then, to supply a user with information about the reason for which A violates ϕ , it is sufficient to construct the piece of $\mathcal{C}([A, \Omega_\phi]_{pt})$ corresponding to the finite computation trace where α appears. We do not need to construct the whole PTS and causal tree. It follows that the efficiency of the overall process depends on the efficiency of the compiler and is only marginally affected by the efficiency of the construction of $\mathcal{C}([A, \Omega_\phi]_{pt})$.

4 Properties of Causal Trees

In this section we show that from our causal trees we can recover the operational semantics of [17]. Moreover, we show that our causal semantics and the causal semantics of [7] coincide for agents belonging to both **tdccp** and **ccp**.

We begin with introducing a notion of normal form for agents, and with giving axioms that permit to transform agents into equivalent normal forms.

Given a string $\vartheta \in (\mathcal{D}_-^+)^*$, we denote with **if** ϑ **then** A the agent:

$$\text{if } \vartheta \text{ then } A \equiv \begin{cases} A & \text{if } \vartheta = \epsilon \\ \text{if } a \text{ then if } \phi \text{ then } A & \text{if } \vartheta = a^+ \phi \\ \text{if } a \text{ else if } \phi \text{ then } A & \text{if } \vartheta = a^- \phi. \end{cases}$$

Definition 5. An agent A is a normal form if there exist agents $A_1, \dots, A_n$ such that $A \equiv [A_1, \dots, A_n]$, $A_i \equiv \text{if } \vartheta_i \text{ then } B_i$, and either $B_i \equiv \text{tell } a_i$, or $B_i \equiv \text{next } A'_i$, or $B_i \equiv \text{skip}$, where A'_i is an arbitrary agent, $1 \leq i \leq n$.

Let us denote by “=” the least congruence satisfying the axioms in Table 3.

Given tokens a and b and agents A and B such that $A = B$, we have that $b \in rt_o(A, a)$ and $[A, \text{tell } a] \hookrightarrow A'$ if and only if $b \in rt_o(B, a)$ and $[B, \text{tell } a] \hookrightarrow A'$, i.e. the operational semantics of [17] does not distinguish between A and B . We prove now that A and B are equivalent also for our causal semantics.

$$\begin{aligned}
[A_0, A_1] &= [A_1, A_0] \quad (\parallel_1) & [A_0, [A_1, A_2]] &= [[A_0, A_1], A_2] \quad (\parallel_2) \\
[A_0, \mathbf{skip}] &= A_0 \quad (\parallel_3) & [A_0, A_0] &= A_0 \quad (\parallel_4) \\
\mathbf{if } a \mathbf{ then } [A_0, A_1] &= [\mathbf{if } a \mathbf{ then } A_0, \mathbf{if } a \mathbf{ then } A_1] \quad (\text{then}) \\
\mathbf{if } a \mathbf{ else } [A_0, A_1] &= [\mathbf{if } a \mathbf{ else } A_0, \mathbf{if } a \mathbf{ else } A_1] \quad (\text{else}) \\
\mathbf{rec } P. A &= A[\mathbf{rec } P. A/P] \quad (\text{rec})
\end{aligned}$$

Table 3. axioms for **tdccp**.

Proposition 1. *Given agents A and B such that $A = B$, a transition $A \xrightarrow{l} A'$ is in $\mathcal{C}([A]_{pt})$ if and only if a transition $B \xrightarrow{l} B'$ is in $\mathcal{C}([B]_{pt})$ and $A' = B'$ can be inferred by axioms $\parallel_1, \parallel_2, \parallel_3, \parallel_4$.*

Proof. (sketch) One can prove that a transition $A \xrightarrow{l} A'$ is in $[A]_{pts}$ if and only if a transition $B \xrightarrow{l'} B'$ is in $[B]_{pts}$, where $D_l = D_{l'}$ and $\vartheta a \in \mathcal{E}_l$ if and only if $\vartheta' a \in \mathcal{E}_{l'}$ and $\vartheta \upharpoonright \mathcal{D}_-^+ = \vartheta' \upharpoonright \mathcal{D}_-^+$. Moreover, it is immediate that $B = B'$ can be inferred by means of axioms $\parallel_1$ - $\parallel_4$. From these properties the thesis follows. $\square$

We prove now that every agent can be transformed in a normal form by means of our axioms.

Proposition 2. *Given an agent A , there exists a normal form B s.t. $A = B$.*

Proof. By structural induction over A .

Basic case: if either $A \equiv \mathbf{tell } a$ or $A \equiv \mathbf{skip}$ then the thesis is immediate because A is a normal form.

Inductive step: we must consider the following cases:

- $A \equiv \mathbf{if } a \mathbf{ then } A_0$: by inductive hypothesis, there exists a normal form B_0 such that $A_0 = B_0$. Since $=$ is a congruence, we have that $A = \mathbf{if } a \mathbf{ then } B_0$. Now, B_0 is of the form $[B_1, \dots, B_n]$, $n \geq 1$. If $n = 1$ then $\mathbf{if } a \mathbf{ then } B_0$ is a normal form. If $n \geq 2$ then we apply axiom *then* and we infer $A = [\mathbf{if } a \mathbf{ then } B_1, \dots, \mathbf{if } a \mathbf{ then } B_n]$, which is a normal form.
- $A \equiv \mathbf{if } a \mathbf{ else } A_0$: this case is analogous to that above.
- $A \equiv \mathbf{next } A_0$: the thesis is immediate because A is a normal form.
- $A \equiv [A_0, A_1]$: by inductive hypothesis, there exist normal forms B_0 and B_1 such that $A_0 = B_0$ and $A_1 = B_1$. Since $=$ is a congruence, we have that $A = [B_0, B_1]$, which is a normal form.
- $A \equiv \mathbf{rec } P. A_0$: by axiom *rec* we infer $A = A_0[A/P]$. By inductive hypothesis, there exists a normal form B_0 such that $A_0 = B_0$. So, we have that $A_0[A/P] = B_0[A/P]$, which is a normal form. $\square$

We prove now that our semantics is consistent with the semantics of [17].

Theorem 1. *Given an agent A and tokens a and b , the following facts are equivalent:*

- $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A)(a)$;
- a triggers a transition $A \xrightarrow{l} A'$ and $b \approx a \wedge \bigwedge_{d \in l} d$.

Proof. We can assume that A is a normal form. In fact, if A is not a normal form, then by Prop. 1 and 2 it follows that there exists a normal form B such that $A = B$ and, therefore, $A \xrightarrow{l} A'$ if and only if $B \xrightarrow{l} B'$, where $A' = B'$ can be inferred by means of axioms $\|_1$ – $\|_4$. Moreover, $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A, a)$ if and only if $[B, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(B, a)$. So, let us assume that A is a normal form and let us reason by structural induction over A .

Basic case: if $A \equiv \text{tell } c$ then $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A)(a)$ iff $A' \equiv \text{skip}$ and $b \approx a \wedge c$. We have $A \xrightarrow{l} A'$ iff $l = \{c^{\{(\emptyset, 0)\}}\}$ and $A' \equiv \text{skip}$. Since $\bigwedge_{d \in l} d = c$ and a triggers $A \xrightarrow{l} A'$, the thesis follows. The case $A \equiv \text{skip}$ is analogous.

Inductive step: we must consider the following cases:

- $A \equiv \text{if } c \text{ then } B$: we distinguish two cases:
 - $a \vdash c$: we have $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A)(a)$ iff $[B, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(B)(a)$. By inductive hypothesis, this is equivalent to having that a triggers $B \xrightarrow{l'} A'$ and $b \approx a \wedge \bigwedge_{d \in l'} d$. Now, $A \xrightarrow{l} A'$ iff $B \xrightarrow{l'} A'$, where $d^{K'} \in l'$ and $(D, 0) \in K'$ iff $d^K \in l$ and $(D \cup \{c^+\}, 0) \in K$. If $a \vdash c$ then a triggers $B \xrightarrow{l'} A'$ iff a triggers $A \xrightarrow{l} A'$. Since $a \wedge \bigwedge_{d \in l'} d \approx a \wedge \bigwedge_{d \in l} d$, the thesis follows.
 - $a \not\vdash c$: we have $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A)(a)$ iff $A' \equiv \text{skip}$ and $b \approx a$. We have that a triggers $A \xrightarrow{l} A'$ iff $A' \equiv \text{skip}$ and $l = \{\text{true}^{\{(\{c^-\}, 0)\}}\}$. Since $\bigwedge_{d \in l} d \approx \text{true}$, the thesis follows.
- $A \equiv \text{if } c \text{ else } B$: this case is analogous to that above.
- $A \equiv \text{next } B$: we have $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A)(a)$ iff $A' \equiv B$ and $b \approx a$. We have $A \xrightarrow{l} A'$ iff $A' \equiv B$ and $l = \{\text{true}^{\{(\emptyset, 0)\}}\}$. Since $\bigwedge_{d \in l} d = \text{true}$ and a triggers $A \xrightarrow{l} A'$, the thesis follows.
- $A \equiv [A_1, \dots, A_n]$: since A is a normal form, $[A, \text{tell } a] \hookrightarrow [A'_1, \dots, A'_n]$ and $b \in \text{rt}_o(A, a)$ iff there exist indexes $\{i_1, \dots, i_n\}$ and tokens $a_1, \dots, a_n, b_1, \dots, b_n$ such that:

1. $\{i_1, \dots, i_n\} = \{1, \dots, n\}$;
2. $a_1 = a, a_{j+1} = b_j$ for $1 \leq j < n, b_n = b$;
3. $b_j \in \text{rt}_o(A_{i_j})(a_j)$ and $[A_{i_j}, \text{tell } a_j] \hookrightarrow A'_{i_j}$;
4. $[A_{i_j}, \text{tell } a_j] \rightarrow_b^* (\Gamma_{i_j}, A'_{i_j}) \not\vdash_b$ and $\sigma(\Gamma_{i_j}) = b_j$.

By inductive hypothesis, items 1-3 are equivalent to having that:

1. $\{i_1, \dots, i_n\} = \{1, \dots, n\}$;
2. $a_1 = a, a_{j+1} = b_j$ for $1 \leq j < n, b_n = b$;
3. a_j triggers $A_{i_j} \xrightarrow{l_j} A'_{i_j}$ and $b_j \approx a_j \wedge \bigwedge_{d \in l_j} d$.

Since A is a normal form, $l_j = \{c_j^{K_j}\}$ and $K_j = \{(D_j, 0)\}$, for some c_j and D_j . Item 4 above is equivalent to having that $b \not\vdash d$ for every $d^- \in \bigcup_{1 \leq j \leq n} D_j$.

These facts are equivalent to having that a triggers $A \xrightarrow{l} [A'_1, \dots, A'_n]$ and $b \approx \bigwedge_{d \in l} d \wedge a$, where $l = l_1 \cup \dots \cup l_n$.

- $A \equiv \text{rec } P.B$: we have $[A, \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(A)(a)$ if and only if $[B[A/P], \text{tell } a] \hookrightarrow A'$ and $b \in \text{rt}_o(B[A/P])(a)$. By inductive hypothesis, this is equivalent to having that a triggers $B[A/P] \xrightarrow{l} A'$ and $b \approx a \wedge \bigwedge_{d^K \in l} d$. The thesis follows because $B[A/P] \xrightarrow{l} A'$ iff $A \xrightarrow{l} A'$. $\square$

The following proposition states that $\mathcal{C}([A]_{pt})$ carries sufficient information to determine whether A is reactive and deterministic.

Proposition 3. *An agent A is reactive and deterministic if and only if every token $a \in \mathcal{D}$ triggers exactly one transition $A \xrightarrow{l} A'$ in $\mathcal{C}([A]_{pt})$.*

Proof. “If”: directly from Theorem 1. “Only if”: by Theorem 1, if $b \in \text{rt}_o(A)(a)$ then a triggers a transition $A \xrightarrow{l}$ and $b \approx a \wedge \bigwedge_{d^K \in l} d$. This transition is unique because, if $A \xrightarrow{l_1} A_1$ and $A \xrightarrow{l_2} A_2$, then $\bigcup_{(D,0) \in K|d^K \in l_1} D \not\sim \bigcup_{(D,0) \in K|d^K \in l_2} D$, as it can be proved immediately. So, a cannot trigger both transitions. $\square$

Example 8. Let A be as in Example 3. We have $A \xrightarrow{l_1} [\text{skip}, \text{skip}]$, $A \xrightarrow{l_2} [\text{skip}, \text{skip}]$, $l_1 = \{b^{\{\{a^-\}.0\}}\}, \text{true}^{\{\{b^+\}.0\}}\}$, $l_2 = \{a^{\{\{b^-\}.0\}}\}, \text{true}^{\{\{a^+\}.0\}}\}$. Label l_1 (resp. l_2) reflects that if the store does not entail a (resp. b) then A reacts by producing b (resp. a). Now, token true entails both transitions.

We recall now the causal semantics of constructs **tell**, **if _ then _** and $[-, -]$ as given in [7]. The language of [7] offers also a construct of nondeterministic choice and a recursion construct. These constructs are not included in **tdccp** because this rejects nondeterminism and allows only temporal guarded recursion.

The behavior of **ccp** agents is described in terms of transitions between configurations, where a configuration Γ is a multiset of *contexted agents* of the form A^a , satisfying the following requirement:

$$\text{if } A^a \in \Gamma \text{ then } \sigma(\Gamma) \vdash a$$

where we identify A and A^{true} , and $\sigma(\Gamma)$ is defined as follows:

$$\sigma(\Gamma) = \begin{cases} \sigma(A_1^{b_1}) \wedge \dots \wedge \sigma(A_n^{b_n}) & \text{if } \Gamma \equiv A_1^{b_1}, \dots, A_n^{b_n} \\ a & \text{if } \Gamma \equiv (\text{tell } a)^b \\ \text{true} & \text{if } \Gamma \equiv [A_0, A_1]^b \text{ or } \Gamma = (\text{if } a \text{ then } A_0)^b. \end{cases}$$

The transition relation is defined by the following rules:

$$\Gamma, [A_0, A_1]^a \rightarrow \Gamma, A_0^a, A_1^a \qquad \frac{\sigma(\Gamma) \vdash b}{\Gamma, (\text{if } b \text{ then } A)^a \rightarrow \Gamma, A^{b \wedge a}}$$

A configuration Γ contains an agent A^a if A has been caused by the token a . So, we require that $\sigma(\Gamma) \vdash a$, namely that a is entailed by the store.

We use x to range over contexted tokens of the form a^b . Now, the output of executing agent A on input a is determined by function $rc_o(A)$ such that:

$$rc_o(A)(a) = x \mid [A^{true}, (\text{tell } a)^a] \rightarrow^* \Gamma \not\vdash \text{ and } \rho(\Gamma) = x,$$

$$\text{and } \rho(\Gamma) = \begin{cases} \rho(A_1^{b_1}) \wedge \dots \wedge \rho(A_n^{b_n}) & \text{if } \Gamma \equiv A_1^{b_1}, \dots, A_n^{b_n} \\ a^b & \text{if } \Gamma \equiv (\text{tell } a)^b \\ true^{true} & \text{if } \Gamma \equiv [A_0, A_1]^b \text{ or } \Gamma = (\text{if } a \text{ then } A_0)^b. \end{cases}$$

In [7] it is defined a notion of equivalence over contexted tokens, and function $rc_o(A)$ does not return a contexted token x as above, but it returns the equivalence class of x .

We show now that our causal semantics and the causal semantics of [7] coincide for agents belonging to both **ccp** and **tdccp**.

Theorem 2. *Given a ccp agent A and a token a , the following facts are equivalent:*

- $rc_o(A)(a) = b_1^{c_1} \wedge \dots \wedge b_m^{c_m} \wedge a^a$;
- a triggers a transition $A \xrightarrow{l} A'$, $l = \{b_1^{K_1}, \dots, b_m^{K_m}, true^{K_{m+1}}, \dots, true^{K_n}\}$, $K_i = (D_i, 0)$ and $\bigwedge_{d^+ \in D_i} d \approx c_i$ for $1 \leq i \leq m$.

Proof. We can assume that A is a normal form. In fact, if A is not a normal form, by Prop. 1 and 2 it follows that there exists a normal form B s.t. $A = B$ and, therefore, $A \xrightarrow{l} A'$ iff $B \xrightarrow{l} B'$. Moreover, it is immediate that if $A = B$ then A and B have the same causal semantics according to [7]. So, let us assume that A is a normal form and let us reason by structural induction over A .

Basic case: $A \equiv \text{tell } e$. We have $rc_o(A)(a) = e^{true} \wedge a^a$ and $A \xrightarrow{l} \text{skip}$, where $l = \{e^{\{(\emptyset, 0)\}}\}$. Now, a triggers $A \xrightarrow{l} A'$ and the thesis follows.

Inductive step: we must consider the following cases:

- $A \equiv \text{if } e \text{ then } B$: we must distinguish two cases:
 - $a \vdash e$: we have that $rc_o(B)(a) = b_1^{c_1} \wedge \dots \wedge b_m^{c_m} \wedge a^a$ iff $rc_o(A)(a) = b_1^{c_1 \wedge e} \wedge \dots \wedge b_m^{c_m \wedge e} \wedge a^a$. By inductive hypothesis, this is equivalent to having that a triggers $B \xrightarrow{l} A'$, $l = \{b_1^{K_1}, \dots, b_m^{K_m}\} \cup \{true^{K_i} \mid m+1 \leq i \leq n\}$, $K_i = (D_i, 0)$ and $\bigwedge_{d^+ \in D_i} d \approx c_i$. If $a \vdash e$ then this is equivalent to having that a triggers $A \xrightarrow{l'} A'$ where $l' = \{b_1^{K'_1}, \dots, b_m^{K'_m}\} \cup \{true^{K'_i} \mid m+1 \leq i \leq n\}$, $K'_i = (D_i \cup \{e^+\}, 0)$. Since $\bigwedge_{d^+ \in D_i} d \approx c_i$, we have that $\bigwedge_{d^+ \in (D_i \cup \{e\})} d \approx c_i \wedge e$ and the thesis follows.
 - $a \not\vdash e$: we have $rc_o(A)(a) = a^a$ and we have that $A \xrightarrow{l} \text{skip}$, where $l = \{true^{\{(\{e^-\}, 0)\}}\}$. Now, a triggers $A \xrightarrow{l} A'$ and the thesis follows.
- $A \equiv [A_1, \dots, A_n]$, $A_i \equiv \text{if } \vartheta_i \text{ then tell } b_i$: we have that $rc_o(A)(a) = b_{i_1}^{c_1} \wedge \dots \wedge b_{i_m}^{c_m} \wedge a^a$ iff there exist tokens $a_1, \dots, a_m$ such that $a_1 = a$, $a_{j+1} = a_j \wedge b_{i_j}$, $rc_o(A_{i_j})(a_j) = b_{i_j}^{c_j} \wedge a_j^{a_j}$, $a_j \vdash c_j$ and $a \wedge b_{i_1} \wedge \dots \wedge b_{i_m} \not\vdash \bigwedge_{d^+ \in \{\vartheta_h\}} d$, for $h \in \{1, \dots, n\} \setminus \{i_1, \dots, i_m\}$. By inductive hypothesis this is equivalent to having that a_j triggers $A_{i_j} \xrightarrow{l_j} A'_{i_j}$, $l_j = \{b_{i_j}^{K_j}\}$, $K_j = (D_j, 0)$, $\bigwedge_{d^+ \in D_j} d \approx c_j$ and $a_j \vdash c_j$, and to having that $A_h \xrightarrow{l_h} A'_h$, $l_h = \{true^{\{(D_h, 0)\}}\}$, where $a_{m+1} \not\vdash d_h$ for some $d_h^- \in D_h$, $h \in \{1, \dots, n\} \setminus \{i_1, \dots, i_m\}$. This is equivalent to having that a triggers $A \xrightarrow{l} A'$, where $l = l_1 \cup \dots \cup l_n$. So, the thesis follows. $\square$

5 Conclusions

We have given a causal semantics for `tdccp`. This semantics is useful for at least two kinds of applications: scheduling of physical systems, and verification of safety properties of systems modeled as `tdccp` programs. We have proved that our semantics agrees with the original operational semantics of `tdccp` and, for programs belonging to both `tdccp` and `ccp`, with the causal semantics of `ccp`.

References

1. Berry, G.: *The Constructive Semantics of Pure Esterel*. Version 3.0, 1999.
URL: <http://www.inria.fr/meije/personnel/Gerard.Berry.html>.
2. Berry, G. and Gonthier, G.: *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*. Science of Computer Programming **19**, 1992.
3. Boudol, G. and Castellani, I.: *A non-Interleaving Semantics for CCS based on Proved Transitions*. Fundamenta Informaticae **11**(4), 1988.
4. Degano, P., De Nicola, R. and Montanari, U.: *Partial Ordering Derivations for CCS*. In FCT '85, Springer LNCS 199, 1985.
5. Degano, P. and Priami, C.: *Proved Trees*. In ICALP '92, Springer LNCS 623, 1992.
6. Fromherz, M. and Saraswat, V.A.: *Model-based Computing: Using Concurrent Constraint Programming for Modeling and Model Compilation*. In CP '95, Springer LNCS 976, 1995.
7. Gupta, V., Jagadeesan, R. and Saraswat, V.A.: *Truly Concurrent Constraint Programming*. To appear in Theoretical Computer Science.
8. Halbwachs, N.: *Synchronous Programming of Reactive Systems*. The Kluwer Academic Publishers, 1993.
9. Halbwachs, N., Lagnier, F. and Raymond, P.: *Synchronous Observers and the Verification of Reactive Systems*. In AMAST '93, 1993.
10. Halbwachs, N., Lagnier, F. and Ratel, C.: *Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE*. IEEE Transactions on Software Engineering **18**(9), 1992.
11. Harel, D. and Pnueli, A.: *On the Development of Reactive Systems*. In Logic and Models of Concurrent Systems, NATO, ASI-13, Springer, 1985.
12. Mc Millan, K.L.: *Symbolic Model Checking*. The Kluwer Academic Publishers, 1993.
13. Plotkin, G.: *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, University of Aarhus, 1981.
14. Pnueli, A.: *How Vital is Liveness? Verifying Timing Properties of Reactive and Hybrid Systems*. In CONCUR '92, Springer LNCS 630, 1992.
15. Saraswat, V.A.: *Concurrent Constraint Programming*. The MIT Press, 1993.
16. Saraswat, V.A., Rinard, M. and Panangaden, P.: *Semantics Foundation of Concurrent Constraint Programming*. In POPL '91, ACM Press, 1991.
17. Saraswat, V.A., Jagadeesan, R. and Gupta, V.: *Default Timed Concurrent Constraint Programming*. In POPL '95, ACM Press, 1995.
18. Saraswat, V.A., Jagadeesan, R. and Gupta, V.: *Timed Default Concurrent Constraint Programming*. Journal of Symbolic Computation **11**, 1996.

CASL-CHART: A Combination of Statecharts and of the Algebraic Specification Language CASL*

Gianna Reggio and Lorenzo Repetto

DISI - Università di Genova, Italy

reggio@disi.unige.it

<http://www.disi.unige.it/person/ReggioG/>

1 Introduction

In this paper we present CASL-CHART a formal visual specification language for reactive systems obtained by *combining* an already existing language for reactive systems, precisely the statecharts as supported by STATEMATE ([6,7]), with an already existing language for the specification of data structures, precisely the algebraic specification language CASL ([12,17]).

We think that is valuable to try to combining an existing specification language, intended for some particular applications following some particular paradigm, with another one aimed at the specification of the data, indeed usually the former is rather poor for what concerns the data part, and that prevents to use it effectively and productively for non-toy applications.

LOTOS, [9], a well established and used specification language for concurrent systems, has been developed when trying to use the CCS (“Calculus of Communicating Systems” [10]) for realistic applications, precisely the specification of protocols. At that point, it became clear the need to extend CCS with the possibility of specifying non trivial data; this extension was done by combining CCS with the already existing algebraic specification language ACT ONE [5].

There are many reasons because the designers of a specification language neglect the data part. In general their main efforts concern the peculiar constructs of the language, while the data part is filled in some way. Sometimes, they even think that the data are not relevant, since they have not tried to use their language for a non trivial application. In other cases, they just think that data and their transformations can be specified indirectly by using the specific constructs of the language. This is usually true from a computational point of view, but not if we consider the use of the language in practice; look, for example, to the specification of queues and stacks realized by particular CCS processes in [10].

On the other hand, the algebraic specification languages, which are extremely apt to specify data structures, cannot be straightly used to specify reactive, concurrent, parallel, ... systems; thus in the literature of the last years we can find many attempts to extend such languages in some way to cope with such applications (see [1] for an extended survey).

* Work supported by CoFI (ESPRIT Working Group 29432).

The above problem has been considered in the European “Common Framework Initiative” (CoFI) for the algebraic specification of software and systems, partially supported by the EU ESPRIT program [12]¹, which brings together research institutions from all over Europe. A result of CoFI is the development of a specification language called CASL (“Common Algebraic Specification Language” [17]) that intends to set a standard unifying the various approaches to algebraic specification and specification of abstract data types. Indeed within CoFI, the “Reactive Systems Task Group” has the “... aim and scope of proposing and develop extensions of the common framework to deal with reactive, concurrent and parallel systems ...”.

CASL-CHART is an attempt to work out a possible extension of CASL for the specification of reactive systems.

We have chosen the statecharts, as the language for the specification of the reactive systems, more precisely the statechart variant supported by STATEMATE [7], because it is visual, formal, well established, widely spread and used in industry. Moreover, statecharts, also if with a very different semantics, have been incorporated in the UML [18], the OMG recent standard notation for object oriented systems.

CASL-CHART is a combination of two formal specification languages. For us “combination” means that the original features of the two languages with their original semantics will be present in the combination; thus all data used in a CASL-CHART specification of a reactive system will be specified by using CASL, while the behaviour of such system will be specified by a statechart.

As a result we have that whenever someone uses CASL-CHART, she/he does not specify algebraically the reactive aspects, but instead she/he uses the statechart machinery (events, steps, ...), and she/he specifies the data with CASL using its particular logic (many-sorted partial first-order logic). Thus, a CASL/STATEMATE user may take advantage of its previous know-how on using such languages, when she/he uses CASL-CHART.

The semantics of the combined languages will be given by “combining” the original semantics of the two languages. Indeed, here we are not interested to give the semantics of CASL-CHART by using CASL, and so in some sense to translate the statechart into CASL (also if that it is possible, see for example in [14] a semantics of the UML statecharts given by using CASL).

Here, we present CASL-CHART on a toy example, a pocket calculator, that it is small but it is sufficient to illustrate the novelties of CASL-CHART w.r.t. the classical statecharts supported by STATEMATE. A full presentation of CASL-CHART (precise visual syntax, reference manual, static and dynamic semantics) can be found in [15]. Unfortunately, we have not here the space to present algebraic specification and statecharts; the reader may refer, e.g., to [2], for the former, and to [8], for the latter.

At this time we are not aware of other attempts to combine statecharts with an algebraic specification language, whereas there are many proposals for putting

¹ More information on CoFI at <http://www.brics.dk/Projects/CoFI/>.

together algebraic specification languages with other notations for coping with concurrency, parallelism, reactivity, and so on, see the extended survey in [1].

There are also proposals for combining statecharts with other specification languages; for example in the German EXPRESS project a specification language $\mu\mathcal{SZ}$ combining the statecharts with Z has been developed ([4,19]); however in this case both Z and the statecharts languages have been extended/and or modified.

2 The Example: A Pocket Calculator

In this paper we use, as a running example, a pocket calculator with a keyboard, a display, and a printer. More precisely it is a small reactive system simulating a pocket calculator (think of, for example, a small application simulating a graphical calculator on the desktop of your PC).

This calculator may

- receive keys from a keyboard: either digits, used to express numbers, or commands (arithmetic operations and printer commands);
- echo the numbers on a display;
- compute the commands corresponding to operations and shows the results on the same display;
- execute the printer commands (print the display content and start a new line).

We concurrently structure the calculator systems as the parallel of four processes: three drivers, taking care of the interactions with the keyboard, the display and the printer respectively, and a computing unit, executing the operations. In Fig. 1 we show using a visual-informal notation how such components cooperate to realize the functionalities of the calculator.

In the following sections we show how we have specified the calculator system using CASL-CHART, showing in the meantime the various features of this visual/formal specification language.

3 The Calculator Specification: A CASL-CHART

A CASL-CHART specification of a reactive system consists of

- a specification of the data used by the system, presented using the algebraic specification language CASL,
- and of a statechart that uses such data.

3.1 The Data Part: A Casl Specification

The informal description of our design of the calculator, see Fig. 1, uses keys (that are digits and commands, that are in turn arithmetic operations, commands for the printer and the equal), numbers (i.e., sequences of digits), and characters. We have specified these data are specified by means of the following CASL specification, which shows many features of that language.

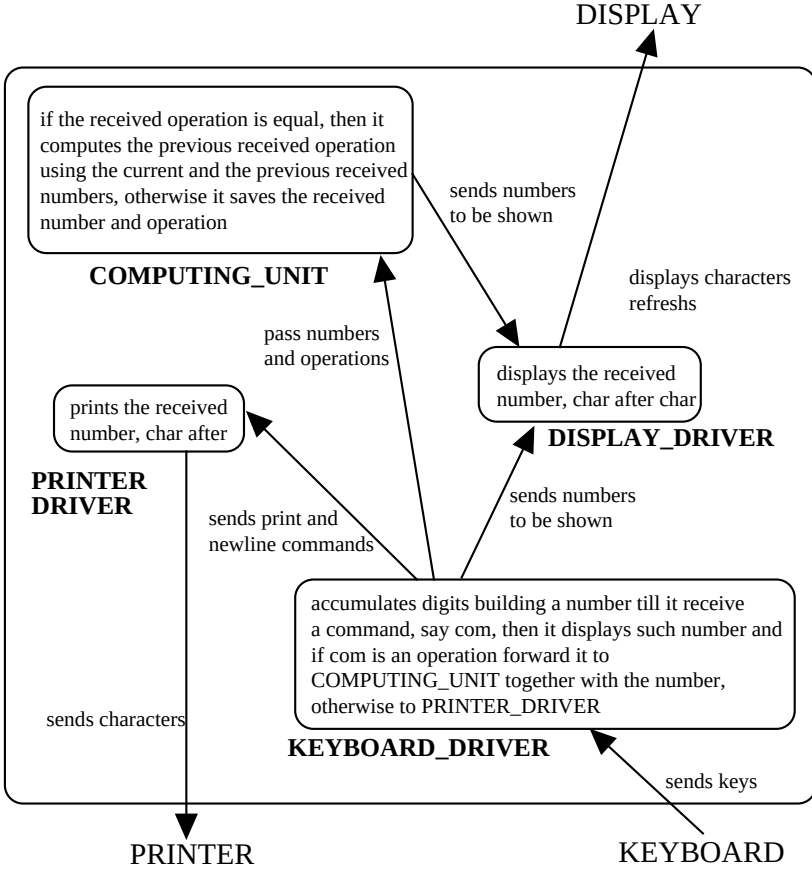


Fig. 1. Concurrent Structuring of the Calculator System

```

spec DATA =
  CHAR then
free {
  types
    Digit ::= 0 | 1 | ... | 9;
    Operation ::= plus | mult | min | div;
    Command ::= sort Operation | pr | nl | eq;
    Key ::= sort Digit | sort Command;
    Num ::= null | ... (Digit; Num); %% numbers as sequences of digits
  ops first : Num →? Digit
    rest : Num →? Num
    _ to_char : Digit → Char
  pred is_null : Num
  vars d:Digit; n: Num

```


- $first(d\ n) = d$
- $rest(d\ n) = n$
- $0\ to\_char = '0'$
- $\dots$
- $9\ to\_char = '9'$
- $is\_null(null)$

} end

DATA is the extension (CASL keyword **then**) of the specification CHAR, provided by the standard CASL libraries [16], with some data types, some operations, and a predicate. The CASL construct **types** allows one to provide for the given types, the constructors (either constant, as *null*, or with parameters, as $-\ . \_$, for *Num*), and possible subtypes (*Operation* is a subtype of *Command*, and *Key* is the disjoint union of *Digit* and *Command*).

CASL allows the users to freely define the syntax of the operations and predicates in a specification; e.g., $-\ to\_char$ states that *to_char* is a postfix operation, and $-\ . \_$ that $\_$ is infix.

The underlying logic of CASL is many-sorted partial first-order logic. Thus in a CASL specification it is possible to declare total (as *to_char*) and partial operations (as *first*), using $\rightarrow$ and $\rightarrow?$ respectively, and predicates (as *is_null*).

“=” in axioms stands for *strong equality*: $t = t'$ holds iff both terms are defined and have the same value, or both are undefined². It is possible to require the definedness of a term with the special atom *def t*.

The keyword **free** states that the specification DATA has an initial semantics; such semantics is characterized by the fact that an atom (either an equation or a predicate application) holds in such model iff it can be proved in the sound and complete deductive system for the CASL logic.

Thus, in the initial model of DATA we have that *is_null* does not hold on $0\ .\ null$, and that *first(null)* is undefined because we cannot prove that *is_null(0 . null)* and *def first(null)*.

CASL provides also a rich set of constructs for structuring specifications and for “architectural specifications” that are not used in this simple example.

3.2 The Whole System Behaviour: A Chart

We specify the behaviour of a reactive system with a statechart, denominated in CASL-CHART simply *chart*, whose form is very similar to that of the STATEMATE statecharts. In Fig. 2 we show the chart defining the whole calculator system.

 is the icon for the chart, while **CALCULATOR** is the name of the chart, or better of its upper level state.

A reactive system may interact with its external environment in a discrete and in a continuous way. In a chart the first kind of interaction is provided by the *events*. The events may be received from outside (input events) graphically

² CASL provides also for existential equality $=_e$: $t =_e t'$ holds iff both terms are defined and have the same value.

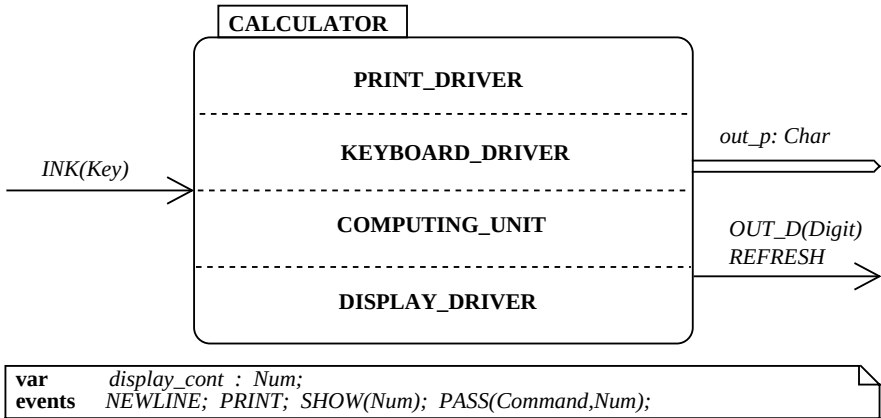


Fig. 2. Chart: **CALCULATOR**

presented in CASL-CHART by a simple incoming arrow attached to the chart icon, and sent outside (output events) graphically presented by an outgoing simple arrow. Moreover, in CASL-CHART events may have parameters, that are values of the types defined by the CASL specification of the data.

The chart **CALCULATOR** has an input event *INK* parameterized by an element of type *Key*, and two output events *OUT_D* and *REFRESH*, the latter without parameters.

The *variables* provide the continuous interactions in a chart. Similarly to the events, they are distinguished in input (which can be only read by the chart) and output (that can be only written by the chart) and are typed using the basic types specified by the CASL part. The variables are graphically presented in CASL-CHART by double arrows, incoming for input and outgoing for output ($\longleftrightarrow$).

The calculator has a unique output variable *out_p* of type *Char*.

The not box below the chart contains the declarations of the *local events* and of the *local variables* that will be used, instead, only by the chart itself.

A reactive system may be composed by several components operating in parallel; in CASL-CHART such components are named *hierarchical charts*, and the parallel decomposition of a chart is shown by splitting the chart icon in slots by means of dashed lines.

CALCULATOR is made of four components: **PRINTER_DRIVER**, **KEYBOARD_DRIVER**, **COMPUTING_UNIT** and **DISPLAY_DRIVER**.

In this case we present the hierarchical charts corresponding to the four components on different drawings, but we could also have put them inside the various slots of **CALCULATOR**.

The chart describes a reactive system that moves step after step. At each step depending on the received input events, on the local events generated in

the previous step, and on the values of the variables, its various components will perform a step, clearly those that can do it, generating output and local events and modifying the values of the variables.

3.3 The System Components: Hierarchical Charts

Display Driver The display driver is a very simple example of hierarchical chart. Indeed, its behaviour is very simple: it waits until it detects the occurrence of the local event *SHOW(n)*, then it records *n* in two local variables *to_display* and *display_cont*; thus it forwards the digits composing *n* to the display one after the other; finally it goes back to wait.

The hierarchical chart in Fig. 3 visually presents such behaviour.

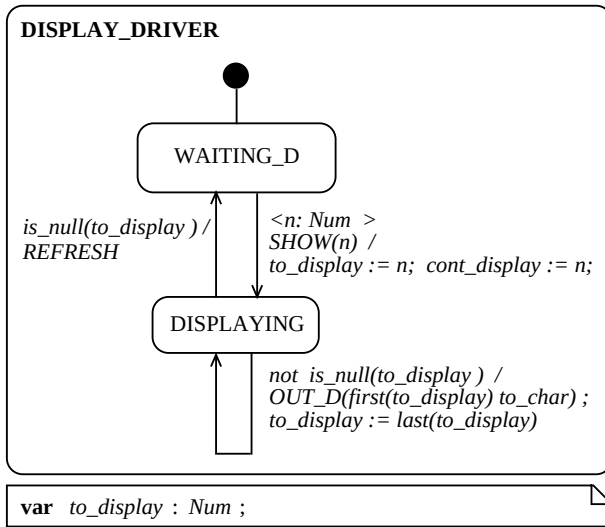
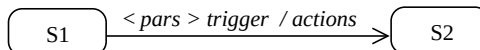


Fig. 3. Hierarchical Chart: **DISPLAY_DRIVER**

A hierarchical chart is “hierarchically/sequentially” decomposed in several states, and in any moment only one of them is active. **DISPLAY_DRIVER** has three states: the initial state, represented by ●, which will be active at the beginning, **WAITING_D**, and **DISPLAYING**.

The local variable *to_display* is declared in the box below the chart icon, because its scope consists of just this hierarchical chart; while the scope of *display_cont* is the whole chart **CALCULATOR**.

The three transitions of this hierarchical chart have different forms, but the general form of a transition in CASL-CHART is



where

- S1 and S2 are two states of the chart, as in classical statecharts (source and target states).
- *trigger* is the transition trigger, and is a CASL formula (thus a formula of the many-sorted partial first-order logic) built using also the chart variables (input and local) and special atoms related to the statechart machinery, checking, for example, for the happening of events and for a state of whole chart to be active.

The CASL-CHART transition trigger is a combination of the trigger and of the condition part of the transitions of the classical statecharts.

- *actions* are the statements describing what to do when the transition is fired, such statements include, for example, assignments to the variables (local and output) and the generation of events; this part is similar to the corresponding one of the classical statecharts.
- *pars* are the transition parameters, this part, differently from the others, is not present in any form in the classical statecharts.³ It consists of a list of variables typed using the types of the CASL specification of the data followed by a condition

$\langle x_1 : t_1, \dots, x_n : t_n \bullet \text{cond} \rangle,$

$x_1, \dots, x_n$ are the transition parameters and may appear in *trigger* and in *actions*.

For any instantiation of the transition parameters with $v_1, \dots, v_n$, values of the appropriate types, s.t. *cond* holds there is a transition obtained by replacing $x_1, \dots, x_n$ by $v_1, \dots, v_n$.

There exists a default value for any part of the transition to be used if such part is lacking: the default for the trigger is the always true formula, that for the actions is the null statement, and that for the parameters is the empty set of parameters.

At each step, a transition whose source state is active and whose trigger holds will be fired; if several transitions with the same source state may fire one of them will be nondeterministically chosen. After the firing the associated actions will be executed, and the target state will become active (clearly the source will be not active except when it coincides with the target).

The transition from WAITING_D to DISPLAYING means that for any value n of type *Num*, if the event *SHOW*(n) has happened, then n will be assigned to the variables *to_display* and *display_cont*.

The transition from DISPLAYING to itself is not parameterized, it may be fired when the content of *to_display* is not null, and when it will fire the event

OUT_D(*first*(*to_display*) *to_char*)

will be generated and the content of the variable *to_display* will be modified. Notice that *is_null*, *first*, *rest*, and *__ to_char* are predicates and operations specified in *DATA*.

³ UML statecharts have something of similar because also their events are parameterized, see [18].

The last transition of the hierarchical chart for the display driver is very simple; indeed it has just the trigger part consisting of checking whether the content of *to_display* is null.

Printer Driver The hierarchical chart specifying the behaviour of the printer driver is quite similar to that of the display driver, see Fig. 4. They differ only because the printer driver passes the character to be printed to the printer by using the output variable *out_p*, instead of generating an event, and gets the number to be printed by looking at the local variable *display_cont*, instead of receiving it as a parameter of the event firing the printing procedure.

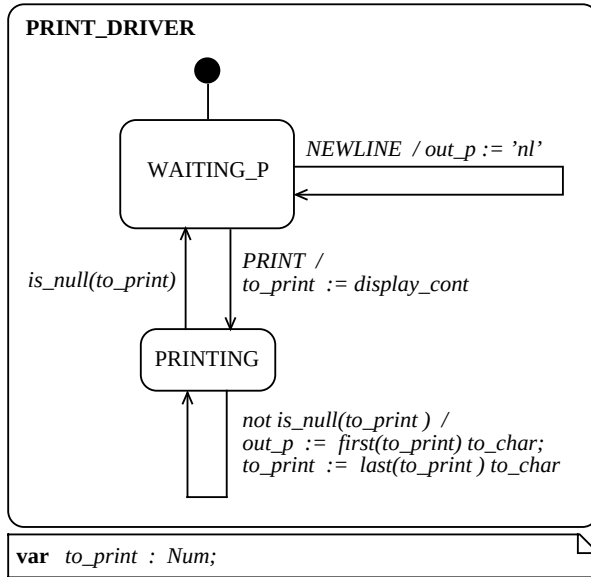


Fig. 4. Hierarchical Chart **Printer_Driver**

Keyboard Driver The behaviour of the keyboard driver is more complex and we specify it with the hierarchical chart in Fig. 5.

The keyboard driver receives keys from the keyboard by means of the events *INK(k)*, then its behaviour depends on *k*. If *k* is a digit, checked by $k \in \text{Digit}^4$, then it is accumulated to build a number, otherwise the accumulated number is shown on the display. “in WAITING_D” checks that the display driver is in the state WAITING_D, and thus it is not showing another number. Afterwards,

⁴ In the specification DATA *Digit* is a subtype of *Key*, and CASL provides special predicates “ $\in$ ” for checking if an element belongs to a subtype.

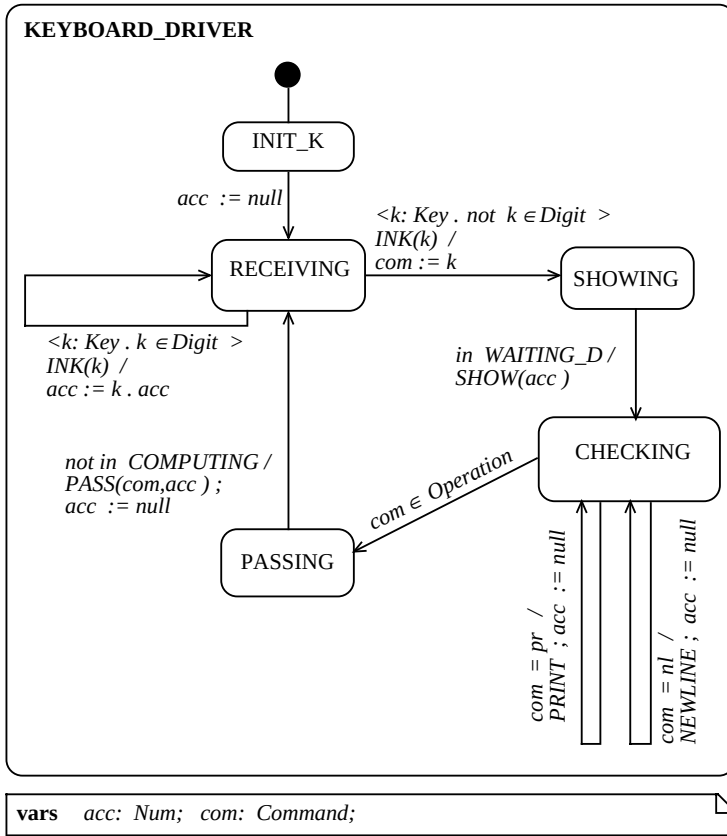


Fig. 5. Hierarchical Chart **Keyboard_Driver**

if k is the print or newline command, then it is passed to the printer driver by generating the corresponding events; otherwise, i.e., it is an operation, checked by $k \in \text{Operation}$ (Operation is a subtype of Command), it is passed to the computing unit together the accumulated number.

Computing Unit The computing unit is the last component of the **CALCULATOR** and is specified by the hierarchical chart in Fig. 6.

The essential role of this component is to compute the result of the application of the received operation to two numbers, represented by sequences of digits. We specify its behaviour rather simply by using a partial operation

$$\text{apply} : \text{Operation} \times \text{Num} \times \text{Num} \rightarrow ? \text{Num},$$

specified by using CASL instead of using the statechart machinery. Technically, we extend the union of the specification DATA with that of the integers INTEGER by *apply* using also some auxiliary operations (*code* and *decode*).

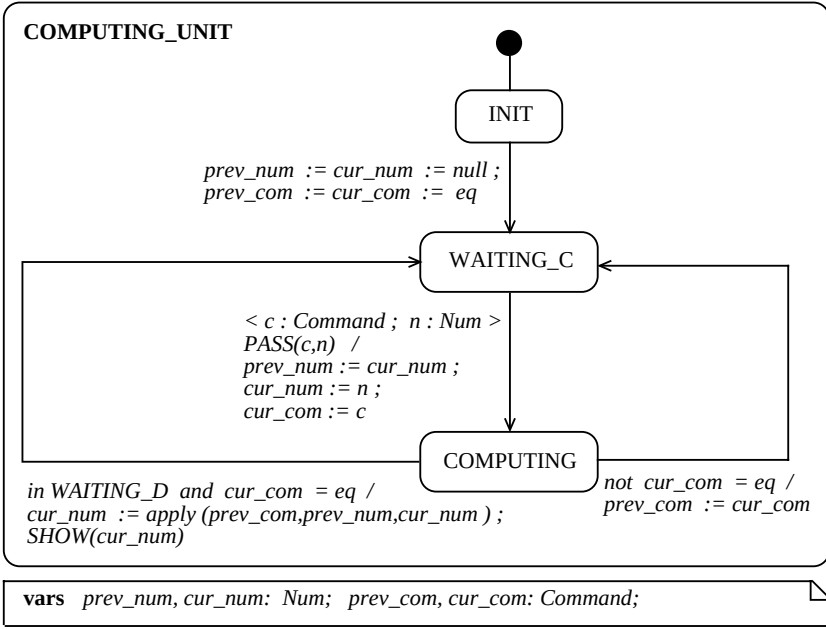


Fig. 6. Hierarchical Chart **Computing_Unit**

spec APPLY =

DATA and INTEGER then

ops $apply : Operation \times Num \times Num \rightarrow? Num$

$code : Integer \rightarrow? Num$

$decode : Num \rightarrow Integer$

vars $n, n' : Num;$

axioms

$apply(plus, n, n') = code(decode(n) + decode(n'))$

$apply(mult, n, n') = code(decode(n) * decode(n'))$

$apply(min, n, n') = code(decode(n) - decode(n'))$

$apply(div, n, n') = code(decode(n) / decode(n'))$

$\forall n : Num \bullet code(decode(n)) = n$

$\forall i : Integer \bullet i > 0 \Rightarrow decode(code(i)) = i$

} end

This example shows how in CASL-CHART the data transformations may be specified algebraically, and thus in a quite abstract way. Moreover, they can be checked by using the various tools supporting the algebraic specifications (as theorem provers and rapid prototypers), see, e.g., [11].

3.4 Other CASL-CHART Constructs

To produce the specification of the pocket calculator, presented in the preceding sections, we have did not used all the CASL-CHART constructs. We want to recall that in CASL-CHART we have also:

- multilevel charts. In the examples shown in this paper, the states of the hierarchical charts are simple states, but they can be decomposed by associating with them a chart, that can be drawn either inside the state icon or on a separate sheet. The charts and hierarchical charts used in the decompositions may be accompanied by declarations of local variables, as we have seen before, and of local events; while the input/output variables and events are allowed only for the upper level chart, describing the whole system (as **CALCULATOR** in our example).
- The “real time” temporal combinators of STATEMATE statecharts may be used in the triggers; precisely: *before*, *since* and *at*, requiring that some condition holds before x time unit, since x time unit, at the time x.
- there are special events corresponding to enter/to exit a state.

4 Semantics

The complete static and dynamic semantics of CASL-CHART is in [15] and we cannot report it here due to lack of space.

The semantics of the CASL specification of the data results in a set of algebras (first-order structures) $\mathcal{D}$ defining the data used in that particular specification. Given $D \in \mathcal{D}$, we define a possible semantics of the CASL-CHART specification using D following the semantics of the STATEMATE statecharts of [7], to be more precise we give two semantics as in [7], a micro-step and a macro-step semantics.

The only difference between our semantics and [7] is when evaluating conditions and expressions, in such point we follow the CASL semantics (see [13]) by considering the special terms (as the statechart variables and the transition parameters) as extra constant operations and the special atoms (as those checking whether an event has happened or that some state of the chart is active) as extra zero-ary predicates.

This combination of the semantics should allow also for a combination of existing support tools. For example, if we restrict the used CASL specifications to conditional specifications, then the STATEMATE toolset could be extended to a toolset for CASL-CHART by replacing the modules taking care of the evaluation of expressions and of conditions with, for example, a rewrite tool for CASL.

5 Conclusion and Future Works

We have presented, on an example, CASL-CHART, that is a combination of the algebraic specification language CASL and of the statecharts as supported by STATEMATE. See [15] for a complete presentation of the syntax and of the semantics of CASL-CHART.

The CASL-CHART specification language is truly a combination of CASL and of the statecharts, because both the ways to specify data structures and the statechart machinery have been preserved, at the syntactic level and also at the semantic level. We have thus obtained a CASL extension for the specification of reactive systems, that is a way to use CASL for relevant practical applications, and, in our opinion, also a better variant of statecharts.

The advantages of CASL-CHART w.r.t. STATEMATE statecharts are

CASL-CHART specifications may be more abstract because the used data and their transformations may be axiomatically specified (e.g., in our small example we have not detailed described as to perform the coding, decoding of numbers, but we have just asserted that such operations are one the inverse of the other, see the specification APPLY). In STATEMATE statecharts, instead such data elaborations have to be described in a very detailed way by using the statechart machinery.

CASL-CHART specifications are more compact because in CASL-CHART we have the possibilities

- of defining apart, in the CASL specification, the user defined data with an user defined syntax, and the operations and the predicates to operate on them;
- of parameterizing events, and thus transitions, over values.

The above features, allows the CASL-CHART users to write statecharts whose visual diagram is simpler and smaller than a corresponding STATEMATE one. We think that this aspect of CASL-CHART is rather valuable, because one of the problems with the visual languages is to avoid that the dimensions of the drawings become too large.

STATEMATE offers three visual notations for the specification of a reactive system during its development: statecharts, that we have considered in this paper, for describing reactive components, *activity charts* to describe the structure of the system in terms of logical components (statecharts and of other kinds), and *module charts* to describe the structure of the system at the implementation level (similar to the deployment diagrams of UML [18]). We plan to extend CASL-CHART to cope with activity and module charts. For what concerns module charts, we plan to investigate their relationships with the architectural specification of CASL (see [3]) developed with similar aims.

Statecharts are one of the many notations incorporated by UML ([18]), and so also if their semantics is partly different from that of the STATEMATE statecharts, we think that this work could offer a starting point for developing a combination of UML with the algebraic specification language CASL, allowing to offer an alternative to OCL, soundly founded on a formal notation.

References

1. E. Astesiano, M. Broy, and G. Reggio. Algebraic Specification of Concurrent Systems. In E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors, *IFIP*

- WG 1.3 Book on Algebraic Foundations of System Specification*, pages 467 – 520. Springer Verlag, 1999.
2. E. Astesiano, B. Krieg-Bruckner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*. Springer Verlag, 1999.
 3. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98)*, Amazonia, Brazil, Lecture Notes in Computer Science, pages 341–357. Springer Verlag, Berlin, 1999.
 4. R. Bussow, R. Geisler, and M. Klar. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1998.
 5. H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with two Levels of Semantics. Technical Report 83-01, TUB, Berlin, 1983.
 6. 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 of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):396–406, 1990.
 7. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
 8. D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The State-mate Approach*. McGraw Hill, 1998.
 9. I.S.O. ISO 8807 Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS, International Organization for Standardization, 1989.
 10. R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
 11. T. Mossakowski. CASL: From Semantics to Tools (tool). In *Proc. TACAS 2000*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000. To appear.
 12. P.D. Mosses. CoFI: The Common Framework Initiative for Algebraic Specification and Development. In M. Bidoit and M. Dauchet, editors, *Proc. TAPSOFT '97*, number 1214 in Lecture Notes in Computer Science, pages 115–137, Berlin, 1997. Springer Verlag.
 13. The CoFI Task Group on Semantics. CASL The Common Algebraic Specification Language: Semantics CoFI Note S-9. Technical report, 1999. <ftp://ftp.brics.dk/Projects/CoFI/Notes/S-9/>.
 14. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In *Proc. FASE 2000 - Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2000. To appear.
 15. G. Reggio and L. Repetto. CASL-CHART : Syntax and Semantics. Technical Report DISI-TR-00-1, DISI – Università di Genova, Italy, 2000. <ftp://ftp.disi.unige.it/person/ReggioG/ReggioRepetto00a.ps>.
 16. M. Roggenbach and T. Mossakowski. Basic Data Types in CASL. CoFI Note L-12. Technical report, 1999. <http://www.brics.dk/Projects/CoFI/Notes/L-12/>.
 17. The CoFI Task Group on Language Design. CASL The Common Algebraic Specification Language Summary. Version 1.0. Technical report, 1999. Available on <http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/>.
 18. UML Revision Task Force. *OMG UML Specification*, 1999. Available at <http://uml.shl.com>.

19. M. Weber. Combining Statecharts and Z for the Design of Safety-Critical Control Systems. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, number 1051 in Lecture Notes in Computer Science, pages 307–326. Springer Verlag, Berlin, 1996.

Message Authentication through *Non Interference*^{*}

Riccardo Focardi¹, Roberto Gorrieri², and Fabio Martinelli³

¹ Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy.
`focardi@dsi.unive.it`

² Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy.
`gorrieri@cs.unibo.it`

³ Istituto per le Applicazioni Telematiche C.N.R., Pisa, Italy.
`Fabio.Martinelli@iat.cnr.it`

Abstract. Authentication is a slippery security property that has been formally defined only recently; among the recent definitions, a rather interesting one has been proposed for the spi-calculus in [1, 2]. On the other hand, in a recent paper [10], we have proved that many existing security properties can be seen uniformly as specific instances of a general scheme based on the idea of non interference. The purpose of this paper is to show that, under reasonable assumptions, also spi-authentication can be recast in this general framework, by showing that it is equivalent to the non interference property called NDC of [8, 9].

1 Introduction

Authentication is, intuitively, the process of reliably verifying the identity of someone or something. This is usually achieved by some form of binding; e.g., a person is recognized on the net by some information (e.g., password or key) (s)he only knows. Authentication has, however, many facets. For instance, while *entity* authentication refers to the capability of identifying the other partner engaged in a session, *message* authentication usually refers to authenticity of both the origin and the content of the message.

Even if there is a widespread agreement on what authentication should be, under a closer scrutiny one realizes that it is a very slippery security property. As a matter of fact, formal definitions of authentication have rarely been given, not widely agreed upon, usually not compared and only recently proposed in the literature (see, e.g., [4, 12, 14, 22]). This is sometimes due to the fact that we first need a formal model on which the problem is defined (and this is often a source of possible proliferation of different proposals) and then a formal definition w.r.t. the chosen model. Moreover, even when a formal definition is given, usually this

^{*} Work partially supported by MURST Progetto “Certificazione automatica di programmi mediante interpretazione astratta” and Progetto “Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi”; CNR Progetto “Modelli e Metodi per la Matematica e l'Ingegneria”; CSP Progetto “ISA: Isp Secured trAnsactions”.

is not (easily) comparable to others, due to different mathematical assumptions of the model.

The main aim of our current research is to find a uniform approach for defining the many variants of security properties (authentication in particular) in such a way that they can be all seen as specific instances of a general scheme (called GNDC [10]). This is badly needed in order to compare, classify and evaluate the merits of the various definitions and, possibly, to provide general, effective proof techniques that can be applied suitably for all properties.

To this aim, in [10] we have presented a process algebra, called CryptoSPA (in turn an improvement of SPA [9] which borrows some concepts from the language defined in [17]), that is expressive enough to model a large class of systems, e.g., (non mobile) security protocols. CryptoSPA has been chosen as the common model for comparing the various properties through the general, unifying scheme. The idea behind is essentially *non interference*, proposed many years ago [11] in a completely different context to study information flow in computer systems and widely studied in [8, 9, 18]. Roughly, a system is secure if its behaviour cannot be significantly altered (hence, with no interference) when executed in a hostile environment. This property is a direct generalization of *Non deducibility on composition* (NDC for short) we have proposed in [8, 9].

Some security properties (e.g., authentication as in [14] and denial of service as in [21]) have been shown as instances of our general scheme in [10]. The main goal of this paper is to show that also the rather different authentication property defined by Abadi and Gordon in [2], once adapted for CryptoSPA, can be formulated in our framework under some reasonable, mild assumptions. This is interesting for the following reasons:

- It strengthens our claim that non interference plays an important role in the specification and analysis of security protocols. Indeed, non interference seems the strongest property that can be defined for cryptographic protocols (see also [10]).
- It helps comparing the various authentication properties among them and with respect to other different security properties, as they are now defined uniformly (in the same language) as instances of the same general scheme.
- It contributes to clarify the spi authentication property: in fact, in its original definition, based on a testing-like semantics, the tester plays both the role of intruder and observer at the same time; in its new equivalent formulation, the two roles are clearly separated. Moreover, the new formulation does not require the explicit definition of the secure specification.
- It may contribute with proof techniques for spi-authentication, as there are already available techniques, also implemented in existing tools [6, 9], for NDC, that is at the base of the corresponding formulation.

2 The Model

In this section we report from [10] the language we use for the specification of authentication properties and protocols. It is called *Cryptographic Security*

Process Algebra (CryptoSPA for short), and it is basically a variant of value-passing CCS [19], where the processes are provided with some primitives for manipulating messages. In particular, processes can perform message encryption and decryption, and also construct complex messages by composing together simpler ones.

2.1 The CryptoSPA Syntax

CryptoSPA syntax is based on the following elements:

- A set $I = \{c_1, c_2, \dots\}$ of *input* channels, a set $O = \{\bar{c}_1, \bar{c}_2, \dots\}$ of *output* ones;
- A set M of basic messages and a set K of encryption keys with a function $\cdot^{-1} : K \rightarrow K$ such that $(k^{-1})^{-1} = k$. The set $\mathcal{M}$ of all messages is defined as the least set such that $M \cup K \in \mathcal{M}$ and $\forall m \in \mathcal{M}, \forall k \in K$ we have that (m, m') and $\{m\}_k$ also belong to $\mathcal{M}$;
- A set $C \subseteq I \cup O$ (s.t. $c \in C$ iff $\bar{c} \in C$) of *public* channels; these channels represent the insecure network where the enemy can intercept and fake messages;
- A family $\mathcal{U}$ of sets of messages and a function $Msg(c) : I \cup O \rightarrow \mathcal{U}$ which maps every channel c into the set of possible messages that can be sent and received along such a channel. Msg is such that $Msg(c) = Msg(\bar{c})$.
- A set $Act = \{c(m) \mid c \in I, m \in Msg(c)\} \cup \{\bar{c}m \mid \bar{c} \in O, m \in Msg(c)\} \cup \{\tau\}$ of actions (τ is the internal, invisible action), ranged over by a ; we also have a function $chan(a)$ which returns c if a is either $c(m)$ or $\bar{c}m$, and the special channel *void* when $a = \tau$; we assume that *void* is never used within a restriction operator (see below).
- A set $Const$ of constants, ranged over by A .

The syntax of CryptoSPA agents is defined as follows:

$$E ::= \underline{0} \mid c(x).E \mid \bar{c}e.E \mid \tau.E \mid E + E \mid E \parallel E \mid E \setminus L \mid E[f] \mid \\ \mid A(m_1, \dots, m_n) \mid [e = e']E; E \mid [\langle e_1 \dots e_r \rangle \vdash_{rule} x]E; E$$

where x is a variable, $m_1, \dots, m_n$ are messages, $e, e_1, \dots, e_r$ are messages (possibly containing variables) and L is a set of input channels. Both the operators $c(x).E$ and $[\langle e_1 \dots e_r \rangle \vdash_{rule} x]E; E'$ bind the variable x in E . It is also necessary to define constants as follows: $A(x_1, \dots, x_n) \stackrel{\text{def}}{=} E$ where E is a CryptoSPA agent which may contain no free variables except $x_1, \dots, x_n$, which must be distinct.

Besides the standard value-passing CCS operators, we have an additional one that has been introduced in order to model message handling and cryptography. Informally, the $[\langle m_1 \dots m_r \rangle \vdash_{rule} x]E_1; E_2$ process tries to deduce an information z from the tuple of messages $\langle m_1 \dots m_r \rangle$ through one application of rule $\vdash_{rule}$; if it succeeds then it behaves like $E_1[z/x]$, otherwise it behaves like E_2 ; for example, given a rule $\vdash_{dec}$ for decryption, process $[\langle \{m\}_k, k^{-1} \rangle \vdash_{dec} x]E_1; E_2$ decrypts message $\{m\}_k$ through key k^{-1} and behaves like $E_1[m/x]$ while $[\langle \{m\}_k, k' \rangle \vdash_{dec} x]E_1; E_2$ (with $k' \neq k^{-1}$) tries to decrypt the same message with the wrong inverse key k' and (since it is not permitted by $\vdash_{dec}$) it behaves like E_2 .

We call $\mathcal{E}$ the set of all the CryptoSPA terms, and we define $sort(E)$ to be the set of all the channels syntactically occurring in the term E .

$$\begin{array}{c}
 \frac{m \ m'}{(m, m')} (\vdash_{pair}) \quad \frac{(m, m')}{m} (\vdash_{fst}) \quad \frac{(m, m')}{m'} (\vdash_{snd}) \\
 \frac{m \ k}{\{m\}_k} (\vdash_{enc}) \quad \frac{\{m\}_k \ k^{-1}}{m} (\vdash_{dec})
 \end{array}$$

Fig. 1. Inference System for message manipulation, where $m, m' \in \mathcal{M}$ and $k, k^{-1} \in K$.

$$\begin{array}{c}
 (input) \frac{m \in Msg(c)}{c(x).E \xrightarrow{c(m)} E[m/x]} \quad (output) \frac{m \in Msg(c)}{\bar{c}m.E \xrightarrow{\bar{c}m} E} \quad (internal) \frac{}{\tau.E \xrightarrow{\tau} E} \\
 (\parallel_1) \frac{E \xrightarrow{a} E'}{E \parallel E_1 \xrightarrow{a} E' \parallel E_1} \quad (\parallel_2) \frac{E \xrightarrow{c(m)} E' \quad E_1 \xrightarrow{\bar{c}m} E'_1}{E \parallel E_1 \xrightarrow{\tau} E' \parallel E'_1} \quad (+1) \frac{E \xrightarrow{a} E'}{E + E_1 \xrightarrow{a} E'} \\
 (=1) \frac{m \neq m' \quad E_2 \xrightarrow{a} E'_2}{[m = m']E_1; E_2 \xrightarrow{a} E'_2} \quad (=2) \frac{m = m' \quad E_1 \xrightarrow{a} E'_1}{[m = m']E_1; E_2 \xrightarrow{a} E'_1} \quad ([f]) \frac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]} \\
 (\backslash L) \frac{E \xrightarrow{a} E' \quad chan(a) \notin L}{E \backslash L \xrightarrow{a} E' \backslash L} \quad (def) \frac{E[m_1/x_1, \dots, m_n/x_n] \xrightarrow{a} E' \quad A(x_1, \dots, x_n) \stackrel{def}{=} E}{A(m_1, \dots, m_n) \xrightarrow{a} E'} \\
 (\mathcal{D}_1) \frac{\langle m_1 \dots m_r \rangle \vdash_{rule} m \quad E_1[m/x] \xrightarrow{a} E'_1}{[\langle m_1 \dots m_r \rangle \vdash_{rule} x]E_1; E_2 \xrightarrow{a} E'_1} \quad (\mathcal{D}_2) \frac{\bar{\beta}m : \langle m_1 \dots m_r \rangle \vdash_{rule} m \quad E_2 \xrightarrow{a} E'_2}{[\langle m_1 \dots m_r \rangle \vdash_{rule} x]E_1; E_2 \xrightarrow{a} E'_2}
 \end{array}$$

Fig. 2. Operational semantics (symmetric rules for $+1$, $\parallel_1$ and $\parallel_2$ are omitted).

2.2 The Operational Semantics of CryptoSPA

In order to model message handling and cryptography, in Figure 1 we define an inference system which formalizes the way messages may be manipulated by processes. It is indeed quite similar to those used by many authors (see, e.g., [13, 16]). In particular it can combine two messages obtaining a pair (rule $\vdash_{pair}$); it can extract one message from a pair (rules $\vdash_{fst}$ and $\vdash_{snd}$); it can encrypt a message m with a key k obtaining $\{m\}_k$ and finally decrypt a message of the form $\{m\}_k$ only if it has the corresponding (inverse) key k^{-1} (rules $\vdash_{enc}$ and $\vdash_{dec}$). We denote with $\mathcal{D}(\phi)$ the set of messages that can be deduced by applying the inference rules on the messages in ϕ . Note that we are assuming encryption as completely reliable. Indeed we do not allow any kind of cryptographic attack, e.g., the guessing of secret keys. This permits to observe the attacks that can be carried out even if cryptography is completely reliable. The formal behaviour of a CryptoSPA term is described by means of the *labelled transition system* $\langle \mathcal{E}, Act, \{\xrightarrow{a}\}_{a \in A} \rangle$, where $\xrightarrow{a}_{a \in A}$ is the least relation between CryptoSPA terms induced by axioms and inference rules of Figure 2.

2.3 Hostile Environments

In this section we report the characterization of hostile environments as given in [10]. Such a characterization is necessary to analyze protocols where some infor-

mation is assumed to be secret, as it always happens in cryptographic protocols. Basically, a *hostile environment* is an agent which tries to attack a protocol by stealing and faking the information which is transmitted on the CryptoSPA *public* channels in set C . In principle, such an agent could be modeled as a generic process X which can communicate only through the channels belonging to C . However, in this way we obtain that X is a completely powerful attacker which is able to “guess” every secret information and is thus not suitable when analyzing cryptographic protocols. This problem can be solved by imposing some constraints on the initial data that are known by the intruders. Given a process E , we call $ID(E)$ the set of messages that syntactically appear in E . Intuitively, this set contains all the messages that are initially known by E . Now, let $\phi_I \subseteq \mathcal{M}$ be the initial knowledge that we would like to give to the intruders X , i.e., the public information such as the names of the entities and the public keys, plus some possible private data of the intruders (e.g., their private key or nonces). For a certain intruder X , we want that all the messages in $ID(X)$ are deducible from ϕ_I . We thus define the set $\mathcal{E}_C^{\phi_I}$ of hostile processes as $\mathcal{E}_C^{\phi_I} = \{X \mid \text{sort}(X) \subseteq C \text{ and } ID(X) \subseteq \mathcal{D}(\phi_I)\}$.

2.4 Semantic Equivalences

In this subsection we define some semantic equivalences that we will use to formalize some security properties. These equivalences equate processes which have an *indistinguishable behaviour*, depending on what we are interested to observe of the processes.

Trace Equivalence: Most of the security properties that have been proposed for the analysis of security protocols are based on the simple notion of *trace*: two processes are equivalent if they exactly show the same execution sequences (called *traces*). We need a transition relation $E \xRightarrow{\alpha} E'$ which does not consider internal τ moves. It is a shorthand for $E(\xrightarrow{\tau})^* E_1 \xrightarrow{\alpha} E_2(\xrightarrow{\tau})^* E'$. For a trace $\gamma = \alpha_1 \dots \alpha_n$ we write $E \xRightarrow{\gamma} E'$ if $E \xRightarrow{\alpha_1} E_1 \xRightarrow{\alpha_2} \dots \xRightarrow{\alpha_{n-1}} E_{n-1} \xRightarrow{\alpha_n} E'$ for some $E_1, \dots, E_{n-1}$. The set $T(E)$ of *traces associated with* E is then defined as $T(E) = \{\gamma \in (Act \setminus \{\tau\})^* \mid \exists E' : E \xRightarrow{\gamma} E'\}$.

Definition 1. We write $E \leq_{\text{trace}} F$ if $T(E) \subseteq T(F)$. E and F are trace equivalent (notation $E \approx_{\text{trace}} F$) iff $E \leq_{\text{trace}} F$ and $F \leq_{\text{trace}} E$. ■

Testing Equivalence for CryptoSPA: In this section we give a notion of behavioural equivalence which incorporates the idea of *unguessable secrets* discussed in the previous section. This equivalence will allow us to rephrase in our model the notion of authentication used in the spi-calculus [1, 2]. The underlying idea is that two processes are equivalent if and only if they cannot be distinguished by any process that does not know the secret values. It is basically a weaker version of classical testing equivalence where equivalent processes should be indistinguishable by *any* environment [5].

We define the notion of experiment as given in the spi-calculus. A *test* is a couple (T, β) , where T is a process called tester and β is a *barb*, i.e., a channel c or $\bar{c}$. We say that a process P *immediately passes a test* (T, β) (denoted by $P \parallel T \Downarrow_\beta$) iff for some message m we have $P \parallel T \xrightarrow{\beta m}$. We also say that a process P *passes a test* (T, β) (denoted by $P \parallel T \Downarrow_\beta$) iff we have that $P \parallel T(\xrightarrow{\tau})^* P' \parallel T'$ and $P' \parallel T' \Downarrow_\beta$.

We parameterize the notion of equivalence by the set of messages ϕ_I which are supposed to be known by the testers. Thus, we obtain the following set of possible testers: $\mathcal{E}^{\phi_I} = \{X \mid X \in \mathcal{E} \text{ and } ID(X) \subseteq \mathcal{D}(\phi_I)\}$. Note that it is strictly larger than the set of hostile environments $\mathcal{E}_C^{\phi_I}$ defined in the previous section. Indeed, such hostile environments can only communicate over the public channels in C .

Definition 2. $P \leq_{\text{may}} Q$ iff $\forall T \in \mathcal{E}^{\phi_I} : P \parallel T \Downarrow_\beta$ implies $Q \parallel T \Downarrow_\beta$. ■

We say that a process P is *may-testing equivalent* to a process Q iff $P \leq_{\text{may}} Q$ and $Q \leq_{\text{may}} P$. We denote that by $P \simeq Q$. It is easy to prove that $\simeq$ is an equivalence relation.

Testing equivalence has been exploited in a very elegant way for the formal analysis of security protocols in the spi-calculus as it implicitly provides a check over every possible hostile environment. In fact, a test can play at the same time the role of the attacker and the role of the observer that checks the outcomes of this attack. Thus, if two processes are may-testing equivalent, this means that they behave in the same way also when they are executed in a hostile environment.

Indeed, our notion of may-testing equivalence differs from the definition given for the spi-calculus, where a test can be every possible spi-calculus process. This difference is due to how secret messages are dealt with. Indeed, the restriction operator of the (s)pi-calculus supports the specification of *new*, an thus unguessable, messages directly inside a process. In this way, a special quantification over a subset of testers (with knowledge limited by ϕ_I) is not needed, and a classic testing-equivalence suffices.¹ The following holds:

Proposition 1. $P \leq_{\text{trace}} Q$ implies $P \leq_{\text{may}} Q$. ■

This derives from the standard result that classical may-testing preorder (with the quantification over any possible test) corresponds to trace preorder. It is interesting to observe that the opposite implication does not hold. As a counterexample consider a process $A(m, k) = [\langle m, k \rangle \vdash_{\text{enc}} x] \bar{c}x$ that encrypts m with k and sends it out on channel c . If k_A is secret (and also all the messages encrypted with k_A) then $A(M_1, k_A) \simeq A(M_2, k_A)$ even if $M_1 \neq M_2$.

On the other hand, if $M_1 \neq M_2$ then we have $A(M_1, k_A) \not\leq_{\text{trace}} A(M_2, k_A)$ because $\bar{c}\{M_1\}_{k_A}$ is a trace for $A(M_1, k_A)$ but not for $A(M_2, k_A)$.

¹ It is worthwhile noticing that in [15, 17] a language similar to CryptoSPA is indeed equipped with a form of secret generation.

3 Message Authentication in the spi-Calculus

In [1, 2] an interesting notion of authentication is proposed. Consider a protocol $S(M)$, which has the aim of transmitting message M from one party A to another one B . $S(M)$ is guaranteeing authentication of message M if whenever a message m is delivered to B by $S(M)$, then m must be the same message as M .

The basic idea behind the verification of this property is the following: we have to generate from $S(M)$ a specification $S_{spec}(M)$ which guarantees authentication by construction, i.e., where the only message that can be delivered to B is indeed M . Then, it is verified whether $S(M)$ is may-testing equivalent to the specification $S_{spec}(M)$, i.e., whether there exists or not an attacker which can induce $S(M)$ to behave wrongly with respect to the delivery of M .

In [1, 2] a general method for generating the specification $S_{spec}(M)$ is not provided and the construction of $S_{spec}(M)$ is indeed illustrated by several example protocols. Intuitively, it can be obtained as follows: in the protocol $S(M)$, every time B is accepting a message m , such a message is replaced by M . In the following we formalize this intuition obtaining a general method for the construction of $S_{spec}(M)$ starting from $S(M)$. We feel that our formalization is general enough to model the significant examples of cryptographic protocols reported in [1, 2] (see also Section 6).

We assume that $S(M)$ is always composed of two parts: one strictly concerning the protocol *execution* and another one representing the *continuation* of the protocol. The latter is usually denoted by $F(x)$, where x represents the received message. The execution part can only send messages on public channels $c \in C$, while the continuation, by definition, does not take part in the protocol, and so it must use channels which are not in C . When may-testing equivalence is checked, the tester ideally acts as an attacker on the public channels and as an observer on the continuations. We give a simple protocol specified in this style.

Example 1. We specify a trivial protocol where Alice sends to Bob a message M as plaintext over an insecure channel c :

$$A(x) \stackrel{\text{def}}{=} \bar{c}x \quad B \stackrel{\text{def}}{=} c(y).F(y) \quad S(x) \stackrel{\text{def}}{=} A(x) \parallel B$$

When Bob receives the message in y , he just behaves like $F(y)$ which will possibly use the message in the future. The corresponding secure specification is:

$$A(x) \stackrel{\text{def}}{=} \bar{c}x \quad B_{spec}(z) \stackrel{\text{def}}{=} c(y).F(z) \quad S_{spec}(x) \stackrel{\text{def}}{=} A(x) \parallel B_{spec}(x)$$

where we can give to Bob a message z that is “magically” substituted for y in the continuation F . Indeed, in $S_{spec}(M)$ we have that Bob always behaves like $F(M)$ whatever he receives on channel c . This represents the secure version of $S(M)$, i.e., a version where the received message is always authentic by construction. Clearly, $S(M)$ does not guarantee any authentication of M : any external user can introduce a fake message M' on c that will be accepted by Bob. As a consequence, $S(M)$ can move to $F(M')$ while $S_{spec}(M)$ cannot.

Note that we can rewrite this protocol in a style that separates more evidently the execution part from the continuation one:

$$A(x) \stackrel{\text{def}}{=} \bar{c}x \quad B' \stackrel{\text{def}}{=} c(y).\bar{p}y \quad S'(x) \stackrel{\text{def}}{=} (A(x) \parallel B' \parallel p(z).F(z)) \setminus p$$

As a matter of fact, $A(x) \parallel B'$ represents the execution, while $p(z).F(z)$ is the continuation with a “guard” $p(z)$ which has the special purpose of enabling the execution of $F(z)$ at the right time. It is possible to show that this form $S'(x)$ for the protocol is trace and may-testing equivalent to the initial one $S(x)$. ■

As we have seen in the example above it can be useful to write a protocol in a particular style that we call *normal form*. In general, more than one continuation could be present. Given a system S , we denote all of its occurrences of continuations as $\{F_1(x_1), \dots, F_n(x_n)\}$, where x_i represents the only free variable of F_i .

We say that a system $S(m_1, \dots, m_n)$ is in normal form if it is written as:

$$(P' \parallel \Pi_{i \in 1 \dots n} p_{F_i}(x_i).F_i(x_i)) \setminus \mathbf{p} \quad (1)$$

where P' is the process S in which every continuation $F_i(x_i)$ is replaced by $\overline{p_{F_i}}(x_i)$, and $\mathbf{p} = \{p_{F_1}, \dots, p_{F_n}\}$ is a set of channels that are used neither in S nor in F_i and are not contained in C . Note that the channels in $\mathbf{p}$ are indexed with the continuations F_i . This is useful for managing multiple concurrent sessions between senders and receivers which could be modeled by considering n copies of the sender and n copies of the receiver in parallel.²

An important assumption (common to the spi-calculus) over the continuations is that they should not know the protocol secrets. Indeed, in such a case they could help the tester attacking the protocol by making public such secrets.

Assumption 1 *We assume that for all continuations $F(x)$ and $\forall m \in \mathcal{D}(\phi_I)$ we have $F(m) \in \mathcal{E}_{\text{Sort}(F(m))}^{\phi_I}$.* ■

Given this particular form for the protocols, it is quite natural to derive a secure specification. Indeed, it is sufficient to define $S_{\text{spec}}(m_1, \dots, m_n)$ as follows:

$$(P' \parallel \Pi_{i \in 1 \dots n} p_{F_i}(x_i).F_i(m_i)) \setminus \mathbf{p} \quad (2)$$

Note that every continuation has its own correct message m_i .³

Remark 1. In the following, we will always consider systems in the normal form (1) and relative specifications in the form (2).

² Indeed, we implicitly assume that equal continuations (up to renaming of free variables) correspond to the same protocol between two users but in different sessions. See Section 6 for an example of multiple session protocol.

³ Please also note that in the case of multiple sessions, this simply requires that a “correct” multiset of messages is delivered from one agent to another one, in whatever possible order. See Section 6 for an example.

Moreover, as done in the spi-calculus, we will always assume that the messages M_i to be delivered by the protocol are not secret, otherwise it would not be possible to observe them with testing equivalence.

Assumption 2 *When we consider a message M to be delivered by a protocol we always assume that $M \in \mathcal{D}(\phi_I)$.* ■

Hence, we can formalize in our framework the notion of authentication proposed in the spi-calculus.

Definition 3. *A system S guarantees weak spi-authentication iff for all vector of messages $(M_1, \dots, M_n)$ we have $S(M_1, \dots, M_n) \simeq S_{\text{spec}}(M_1, \dots, M_n)$.* ■

In [1], it is proposed another notion of authentication where there is also a quantification over the continuations F_i . In this way it is indeed possible to study if the protocol guarantees authentication in every possible context, rather than just in a special situation characterized by a specific continuation. To clarify this point we give the following simple example:

Example 2. Consider again the simple protocol of Example 1. Now, consider the special case where $F(x) = \mathbf{0}$. We obtain that for every M , $S(M)$ is equivalent to $S_{\text{spec}}(M)$. Indeed $S(M)$ and $S_{\text{spec}}(M)$ become syntactically the same process if $F(x) = \mathbf{0}$. However, we have already observed that this trivial protocol cannot guarantee the authenticity of M . On the contrary, if we replace $F(x)$ with $\overline{\text{out}}\ x$ then we may detect the attack. The continuations play thus a central role. In particular, if the behaviour of a continuation does not depend on the message x then it cannot help detecting authentication attacks. ■

Let $\mathbf{F}$ be a vector of continuations $\langle F_1, \dots, F_n \rangle$. We will denote with $S^{\mathbf{F}}$ the system S where its continuations are replaced by $\mathbf{F}$. Moreover, the initial vector of a system S will be denoted by $\mathbf{F}^*$. When we quantify over all the possible vector of continuations, we require that equal continuations are always replaced by equal continuations. More formally, we say that two vectors $\langle F_1, \dots, F_n \rangle, \langle F'_1, \dots, F'_n \rangle$ are *compatible* if and only if $\forall i, j \in \{1, \dots, n\} : F_i = F_j \text{ iff } F'_i = F'_j$. We then consider only the vectors of continuations that are compatible with the initial one $\mathbf{F}^*$. This condition is necessary if we want to analyze multiple sessions where many instances (all with the same continuation) of an agent are considered (see Section 6). Hence, we obtain the following more general definition:

Definition 4. *A system S guarantees (strong) spi-authentication iff for all the vectors of continuations $\mathbf{F}$ compatible with $\mathbf{F}^*$, and for all vectors of messages $(M_1, \dots, M_n)$ we have $S^{\mathbf{F}}(M_1, \dots, M_n) \simeq S^{\mathbf{F}}_{\text{spec}}(M_1, \dots, M_n)$.* ■

In the next section we will see that there exists a *canonical continuation* $\mathbf{F}'$ such that the weak spi-authentication of $S^{\mathbf{F}'}$ implies spi-authentication of S . The solution of using may-testing equivalence for authentication verification is indeed very elegant, but mixing the tester and the intruder could generate some confusion. Indeed, it would be much more intuitive to separate the intrusion activity from the equivalence check. This is what is done in the property we

are going to present in the next section. Moreover, here the definition of the specification is somewhat arbitrary and unclear. In the next section we will show that the definition of a secure specification is not necessary anymore when we adopt a notion of authentication based on Non-interference.

4 NDC-Based Authentication

We recall the notion of Non-Interference (NI) [11]. NI was proposed in system security as a model for the detection of all possible interferences from a certain group of users to another one. It has been formalized in different ways (see, e.g., [8, 11, 20, 23]), and also applied in the verification of various properties of security protocols [6, 7], among which authentication. Indeed, the correctness of a protocol can be proved by guaranteeing that an enemy is not able to interfere at all with the execution of the protocol, i.e., that the protocol with and without the enemy behaves exactly in the same way. In this respect, we have already noticed a similarity in the two approaches.

In the following we will use the generalization of NI to the process algebraic setting proposed in [8] and called *Non Deducibility on Compositions (NDC)*. The idea is the following: the group U_H of users (or high level processes) that must not interfere with the other users (in group U_L of low level processes) is characterized by the set of actions that its components can execute. Let C be the set of channels which processes in U_H can use for communication. A system S is NDC if every possible process $X \in U_H$ composed with S is not able to modify the behaviour of S observed from the point of view of the users in U_L . Thus, U_H corresponds to $\mathcal{E}_C^{\phi_I}$ and NDC can be defined as follows, where we always consider processes S which are in form (1).

Definition 5. *A process S is NDC iff $\forall X \in \mathcal{E}_C^{\phi_I} (S \parallel X) \setminus C \approx_{trace} S \setminus C$.* ■

The only difference with respect to the definition given in the original (SPA) model is that the knowledge of processes $X \in U_H$ is bounded by ϕ_I . In CryptoSPA, this is required to guarantee reliable encryption. NDC requires that high level processes in $\mathcal{E}_C^{\phi_I}$ are not able to change the low level behaviour of the system (represented by $S \setminus C$). As a matter of fact $S \setminus C$ is the system where no high level activity is allowed. If it is equivalent to $(S \parallel X) \setminus C$ this clearly means that X is not able to modify in any way the execution of S .

Note that NDC, in its original definition, is based on trace equivalence. Here we give a definition of may-testing based NDC (TNDC) in order to compare it with the authentication notion defined in the previous section.

Definition 6. *A process S is TNDC iff $\forall X \in \mathcal{E}_C^{\phi_I} (S \parallel X) \setminus C \simeq S \setminus C$.* ■

An intuitive notion of authentication can be given in a natural way through TNDC/NDC as follows:

Definition 7. *Let S be a protocol. Then, S guarantees (T)NDC-authentication iff for all messages $M_1, \dots, M_n$ we have that $S(M_1, \dots, M_n) \in (T)NDC$.* ■

Actually in the definition above we do not take care of the continuations. In particular, $S(M_1, \dots, M_n) \in (T)NDC$ requires that $S(M_1, \dots, M_n)$ composed with whatever enemy $X \in \mathcal{E}_C^{\phi_I}$ is equivalent to $S(M_1, \dots, M_n) \setminus C$. Since C represents the set of channels over which the parties communicate, then $S(M_1, \dots, M_n) \setminus C$ corresponds to a secure specification (execution with protected channels). In this approach we obtain, in a sense, the secure specification for free.

It is interesting to see that since we restrict the behaviour of the system on the channels in C we actually do not observe directly the communication, but only the behaviour on the continuations. Indeed, we obtain a specification that is somewhat simpler than the one needed in the spi-calculus authentication approach. As an example, consider again the protocol $S(x)$ of Example 1. We obtain that $S(M) \setminus C$ is trace and may-testing equivalent to the process $F(M)$, which is just the expected (correct) continuation for the protocol.

5 Comparison

In this section, we formally compare the notions of authentication presented in the previous sections, i.e., the one proposed for the spi-calculus with the TNDC/NDC-based one.

We have already pointed out some of the similarities between them: both of the properties are based on a notion of behavioral equivalence; moreover, they both check whether the “system under attack” behaves like a secure specification. It is however important to notice that this is done in a quite different way. In the spi-calculus the system is implicitly checked against all the possible interactions with the (hostile) environment through the use of the may-testing equivalence. On the contrary, the TNDC/NDC-based approach does it explicitly through the quantification over possible attackers.

The first interesting result shows that authentication in the spi-calculus is at least as discriminating as the TNDC-based one, when S_{spec} guarantees TNDC-authentication.

Proposition 2. *Let S be a protocol and S_{spec} be a secure specification (for S) that guarantees TNDC-authentication. If S guarantees spi-authentication then S also guarantees TNDC-authentication. ■*

One of the hypotheses of the proposition above is that S_{spec} guarantees TNDC-authentication. As a matter of fact, we can prove the results of this section, under the following well-formedness conditions:

- (i) For all vector of messages $\mathbf{M}$, $S(\mathbf{M}) \setminus C \simeq S_{spec}(\mathbf{M}) \setminus C$
- (ii) For all vector of messages $\mathbf{M}$, $S_{spec}(\mathbf{M}) \setminus C \simeq \prod_{k \in 1..n} F_k(M_k)$

They are both very natural. In particular (i) is an obvious requisite for S_{spec} : the system S and its specification S_{spec} behave the same when the public channels are protected through the restriction (when no attack is possible). Condition (ii) requires that all the continuations of the specifications when there is no attacker at all, are eventually enabled, i.e., the specification is well-formed since

it does not contain unreachable continuations. If this is not the case some useless redundancy is present in S_{spec} . By exploiting condition (ii), we can prove:

Lemma 1. S_{spec} guarantees TNDC-authentication. ■

This lemma gives, in a sense, a formal evidence of the correctness of how we define specifications S_{spec} . Thus, in general the spi-authentication is at least as discriminating as the TNDC-based one is:

Corollary 1. *If a system S guarantees spi-authentication then S also guarantees TNDC-authentication.* ■

The other implication is less obvious. As a matter of fact, since in TNDC the action of the intruder is separated from the observation activity, we have to be sure that this “splitting” can be performed without loosing discriminating power. As our intruders are not able to communicate over the channels of the continuations, one may think, at a first glance, that we may miss some attack. We show that this is not the case.

First of all, we simplify the notion of spi-authentication by defining *canonical* continuations, i.e., special continuations that allow us to avoid the universal quantification over all the possible vectors $\mathbf{F}$. In particular, we consider the special vector of continuations $\mathbf{F}'$ where $F'_i(x_i) = \overline{out_{F_i^*}}(x_i)$ and such that the channels $out_{F_i^*}$ are not public, i.e., are not in C . Please note that this vector is compatible with $\mathbf{F}^*$. By exploiting condition (i), we can prove:

Proposition 3. *We have that S guarantees spi-authentication iff $S^{\mathbf{F}'}$ guarantees weak spi-authentication.* ■

This interesting result basically states that there exists a *canonical continuation* $\mathbf{F}'$ that can be considered in order to prove spi-authentication for every possible continuation. Moreover, this result can also be exploited to compare such notion of authentication with the NDC-based one. The following holds:

Lemma 2. *If $S^{\mathbf{F}'}$ guarantees TNDC-authentication then $S^{\mathbf{F}'}$ also guarantees weak spi-authentication.* ■

From the previous results it can be easily proved the following.

Theorem 1. *A system S guarantees (strong) spi-authentication if and only if $S^{\mathbf{F}'}$ guarantees TNDC-authentication.* ■

Now we show that the two notions of message authentication based on non interference, i.e. TNDC and NDC, actually coincide when we study canonical forms. This is very useful to avoid the implicit quantification in TNDC due to testing equivalence. We give the following proposition which can be applied when P is the product of the canonical continuations $\mathbf{F}'$ in order to obtain the final result (Theorem 2).

Proposition 4. *Let P be a CryptoSPA process which can only execute a finite number of output actions and such that $ID(P) \subseteq \mathcal{D}(\phi_I)$. Then, for all Q , $P \simeq Q$ iff $P \approx_{trace} Q$.* ■

Finally, the following result completes the comparison:

Theorem 2. *A system S guarantees (strong) spi-authentication if and only if $S^{F'}$ guarantees NDC-authentication. ■*

6 An Example: The Wide Mouthed Frog Protocol

In this section we show how to use NDC to analyze a simplified version (also studied in [2]) of the Wide Mouthed Frog Protocol [3].

We consider two processes A and B respectively sharing keys k_{AS} and k_{BS} with a trusted server S . In order to establish a secure channel with B , A sends a fresh key k_{AB} encrypted with k_{AS} to the server S . Then, the server decrypts the key and forwards it to B , this time encrypted with k_{BS} . Now B has the key k_{AB} and A can send a message m_A encrypted with k_{AB} to B . The protocol should guarantee that when B receives m_A , such a message has been indeed originated by A . The protocol is composed of the following three messages:

Message 1 $A \rightarrow S : A, \{B, k_{AB}\}_{k_{AS}}$
 Message 2 $S \rightarrow B : \{A, k_{AB}\}_{k_{BS}}$
 Message 3 $A \rightarrow B : \{m_A\}_{k_{AB}}$

The main difference with respect to the original protocol is that here messages 1 and 2 do not contain timestamps. This makes the protocol sensitive to a replay attack (as already remarked in [1]). We specify the protocol as the following CryptoSPA normal form:⁴

$$\begin{aligned} A(m, k) &\stackrel{\text{def}}{=} \overline{c_1}(A, \{(B, k)\}_{k_{AS}}) \cdot \overline{c_3}\{m\}_k \\ B &\stackrel{\text{def}}{=} c_2(y) \cdot [\langle y, k_{BS} \rangle \vdash_{dec} z] [z \vdash_{snd} s] c_3(t) \cdot [\langle t, s \rangle \vdash_{dec} w] \overline{p_F} w \\ S &\stackrel{\text{def}}{=} c_1(u) \cdot [u \vdash_{snd} x] [\langle x, k_{AS} \rangle \vdash_{dec} y] [y \vdash_{snd} z] \overline{c_2}\{(A, z)\}_{k_{BS}} \cdot S \\ P &\stackrel{\text{def}}{=} (A(m_A, k_{AB}) \parallel A(m'_A, k'_{AB}) \parallel B \parallel B \parallel S \parallel p_F(z).F(z) \parallel p_F(z).F(z)) \backslash p_F \end{aligned}$$

where c_1 , c_2 and c_3 are the three channels where messages 1, 2 and 3 are communicated, respectively. Note that we have considered two instances of A and B in order to observe the replay attack. Indeed, if we consider only one instance of A and B no attack is possible here. Note also that A has different messages and session keys in the two sessions.

In order to check if P satisfies (strong) spi-authentication we can just verify if $P^{F'}$ guarantees NDC-authentication. Note that $P^{F'}$ is obtained from P by replacing $F(w)$ with $\overline{out_F} w$. Note also that in the two instances of B the continuation is exactly the same (this allows to model multiple sessions). We find out that $P^{F'}$ does not guarantee such a property. Indeed, it is easy to

⁴ This protocol specification is quite simplified since there are only two users and the possible sessions are fixed in advance. However, this is sufficient here to show how the attack can be revealed.

see that P with no enemy, i.e., $P \setminus \{c_1, c_2, c_3\}$, is trace equivalent to process $\overline{out}_F m_A \parallel \overline{out}_F m'_A$. As a matter of fact the two sessions can be executed in any possible interleaving. This represent the intended execution of the protocol. Now, consider the enemy $X \stackrel{\text{def}}{=} c_2(x).\overline{c_2} x.\overline{c_2} x.c_3(y).\overline{c_3} y.\overline{c_3} y$. It is easy to see that $(P \parallel X) \setminus C$ is able to execute the trace $\overline{out}_F m_A.\overline{out}_F m_A$ that is not a trace for process $\overline{out}_F m_A \parallel \overline{out}_F m'_A$. Hence, $(P \parallel X) \setminus C \not\leq_{\text{trace}} \overline{out}_F m_A \parallel \overline{out}_F m'_A$ and NDC-authentication is not satisfied.

The enemy X basically intercepts messages 2 and 3 and replays them, inducing B to commit twice on message m_A (as shown by trace $\overline{out}_F m_A.\overline{out}_F m_A$). This attack is quite critical in some situations. As an example, m_A could be a request of money transfer that would be executed twice. In order to avoid this attack, it could be possible to modify the protocol (as done in [2]) by adding nonce handshakes. We do not show this here for lack of space.

7 Future Work

In order to compare various formalizations of security properties, we have defined in [10] a general scheme that permits to capture a number of properties (e.g., authentication as in [14] and denial of service as in [21]) as particular instances of the scheme itself. The results presented in this paper have allowed us to extend the set of properties defined in the scheme. Our main issue is now to find comparison results in order to obtain a complete classifications in the style of [8, 14], which could help in evaluating the relative merits of all such properties.

Another aim of our current research is to provide general, effective proof techniques that can be suitably applied to a set of security properties. In this paper, we have seen that the spi-authentication corresponds to a notion of authentication based on NDC. This permits, in principle, the reuse of automatic checking techniques for NDC in order to check spi-authentication over CryptoSPA protocols; indeed if ϕ_I is finite then it is possible to find a most-general intruder Top such that NDC is reduced to just one check $(S^{F'}(M_1, \dots, M_n) \parallel Top) \setminus C \approx_{\text{trace}} S^{F'}(M_1, \dots, M_n) \setminus C$ (see [10] for more details) that could be verified using the tool in [6]. The problem of finding a finite set of vectors of messages which could avoid the universal quantification over all the possible messages $\mathbf{M}$, is still open.

We are currently studying the relations between spi and NDC authentication also in the spi-calculus model, where the possibility of communicating channel names between processes gives rise to new interesting issues. These complicate very much the comparison, but we feel that at least some of the results presented here also hold in such a model.

References

- [1] M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In *Proc. of CONCUR'97*, pages 59–73. LNCS 1243, 1997.
- [2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

- [3] M. Burrows, M. Abadi, and R. Needham. "A Logic of Authentication". *Proc. of the Royal Society of London*, 426:233–271, 1989.
- [4] C. Bodei, P. Degano, R. Focardi, and C. Priami. Authentication via localized names. In *Proc. of CSFW'99*, pages 98–110. IEEE press, 1999.
- [5] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
- [6] A. Durante, R. Focardi, and R. Gorrieri. CVS: A compiler for the analysis of cryptographic protocols. In *Proc. of CSFW'99*, pages 203–212. IEEE press, 1999.
- [7] R. Focardi, A. Ghelli, and R. Gorrieri. Using non interference for the analysis of security protocols. In *Proc. of DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.
- [8] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1994/1995.
- [9] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, 1997.
- [10] R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In *Proc. of World Congress on Formal Methods (FM'99)*, pages 794–813. Springer, LNCS 1708, 1999.
- [11] J. A. Goguen and J. Meseguer. Security policy and security models. In *Proc. of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1982.
- [12] D. Gollmann. What do we mean by entity authentication? In *Proc. of Symposium in Research in Security and Privacy*, pages 46–54. IEEE Press, 1996.
- [13] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of TACAS'96*, pages 146–166. LNCS 1055, 1996.
- [14] G. Lowe. A hierarchy of authentication specification. In *Proc. of the 10th Computer Security Foundation Workshop*, pages 31–43. IEEE press, 1997.
- [15] D. Marchignoli and F. Martinelli. Automatic verification of cryptographic protocols through compositional analysis techniques. In *Proc. of TACAS'99*, volume 1579 of LNCS, pages 148–163, 1999.
- [16] W. Marrero, E. Clarke, and S. Jha. A model checker for authentication protocols. In *Proc. of DIMACS Workshop on Design and Formal Verification of Security Protocols*. Rutgers University, Sep. 1997.
- [17] F. Martinelli. Languages for description and analysis of authentication protocols. In *Proc. of ICTCS'98*, pages 304–315. World Scientific, 1998.
- [18] F. Martinelli. Partial model checking and theorem proving for ensuring security properties. In *Proc. of CSFW'98*, pages 44–52. IEEE press, 1998.
- [19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [20] P. Y. A. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. of CSFW'99*, pages 214–227. IEEE press, 1999.
- [21] S. Schneider. Formal analysis of a non-repudiation protocol. In *Proc. of CSFW'98*, pages 54–65. IEEE Press, 1998.
- [22] S. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9), September 1998.
- [23] J. T. Wittbold and D. M. Johnson. "Information Flow in Nondeterministic Systems". In *Proc. of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 144–161. IEEE Computer Society Press, 1990.

Plugging Data Constructs into Paradigm-Specific Languages: Towards an Application to UML *

Egidio Astesiano, Maura Cerioli, and Gianna Reggio

DISI–Dipartimento di Informatica e Scienze dell’Informazione,
Università di Genova, Via Dodecaneso, 35, 16146 Genova, Italy,
{astes,cerioli,reggio}@disi.unige.it

Abstract. We are interested in the composition of languages, in particular a data description language and a paradigm-specific language, from a pragmatic point of view. Roughly speaking our goal is the description of languages in a component-based style, focussing on the data definition component. The proposed approach is to substitute the constructs dealing with data from the “data” language for the constructs describing data that are not specific to the particular paradigm of the “paradigm-specific” language in a way that syntax, semantics as well as methodologies of the two components are preserved.

We illustrate our proposal on a toy example: using the algebraic specification language CASL, as data language, and a “pre-post” condition logic *à la Hoare*, as the paradigm specific one.

A more interesting application of our technique is fully worked out in [16] and the first step towards an application to UML, that is an analysis of UML from the data viewpoint, following the guidelines given here, is sketched at the end.

Introduction

Languages, including specification languages, oriented to particular paradigms, like for instance object-oriented, (higher-order) functional, concurrent languages, frequently have poor, or insufficiently investigated constructs to specify the used data, sometimes even no constructs at all.

This is somehow reasonable, because the main efforts of the language designers are spent, especially if the language is part of an open research proposal, on the most interesting constructs, those qualifying the language itself and the attention of the users should focus on the paradigm-specific part. Moreover, in many cases it is possible to realize the interesting data structures using the paradigm-specific constructs (for instance, static data by object-oriented classes,

* Partially supported by CoFI, ESPRIT Working Group 29432, and Murst - Saladin (Software Architecture and Languages to coordinate Distributed Mobile Components).

queues and stacks by CCS processes, see, e.g., [15]), though this solution is not very efficient and leads to unnaturally complicated specifications (programs).

Though many designers (mainly in the “formal method community”) regard the data part as not relevant, this opinion cannot be shared from the pragmatic point of view of an end user, because any realistic application requires some nontrivial data. Indeed, for instance, the first attempt to use CCS [15] for some realistic applications, precisely protocol specification, led to design LOTOS [12], where powerful constructs for data were provided by an algebraic specification language ACTONE [7]. Moreover, in the literature of the last decades many attempts to combine paradigm-specific languages with data languages have been presented. For example, combinations of an algebraic specification language with

- a process calculus: other than LOTOS, we have PSF, [13,14], a combination of a process calculus similar to ACP with ASF (Algebraic Specification Formalism) [4];
- Petri nets [17]: many proposals see, e.g., [18,2,6,23];
- statecharts [9]: see in [16] a combination obtained following the method proposed here.

But, there are also combinations using Z for the data part, for example with statecharts [5,24], and with an object-oriented language as Object-Z [19].

On the other hand, fixing some (even powerful) data constructs in a paradigm-specific language can restrict the community of potential users, if some of them are already familiar with different style(s) of data description or if some community has very specific needs in terms of datatypes.

Thus, the best solution, in our opinion, is to divorce the choice of the language for the data definition (from now on DL) from that of the language for paradigm-specific constructs (from now on PSL) as much as possible, aiming at a sort of modular language design, or *component based* language design, where the data part can be plugged in.

Since the existing languages have not been introduced in this style, we first have to cope with the problem of extracting from their descriptions the *components* we want to combine. Therefore, we have to analyze the constructs of a paradigm-specific language in order to identify the “just-data” part, that we will replace by our DL. Section 1 will be devoted to the description of a layering method.

Once we have fixed the components we want to combine, the next step is the definition of (the semantics of) the resulting composition.

Several examples of compositions in literature are the origin of new theories and techniques, to get a more powerful framework, where the constructs of both languages can be given a semantics (e.g., algebras as states, concurrent ML).

Another approach quite widespread to solve this problem is using the features of one language/formalism to express concepts of both the data and the paradigm-specific levels, in particular, for instance, coding one of the languages into the other (e.g., the use of Z to specify processes seen as sets of traces; the use of a higher-order logic language to specify axiomatically processes seen

as infinite traces modelled as functions; LTL [1], where the algebraic specification techniques and related constructs are extended to handle processes seen as labelled transition systems).

Neither solution is palatable to us, for pragmatic reasons. Indeed, in either case one or both the starting languages (or at least their semantics) is redefined, so that tools, results, theorems and “pieces of code” cannot be simply inherited but have to be somehow adjusted (validated). In the most drastic cases, even the “programming” techniques cannot be lifted from the original framework(s) to the combination and this is clearly unacceptable from the user point of view. We want to keep the original semantics of each construct, building a mechanism to pass the results of computations from one framework to the other.

Therefore, in Section 2, we will sketch a method to integrate the semantics of the two parts that does not require new theory. Of course there is no guarantee that such a method works in general; indeed, we need (very few and most reasonable) assumptions on the way the syntax and the semantics of the two starting languages are given. We will use as a running example the integration of the CASL algebraic specification language and of HLL a simple logic *à la Hoare*.

1 A Data-Driven Language Taxonomy

The problem we are tackling is the composition of two languages: ideally, one of them, say PSL, having constructs for some specific paradigm, that we will call *paradigm-specific language*, but with questionable or even missing constructs for datatypes; the other, say DL, providing a rich language for describing datatypes, that we will call *data language*. It is interesting to note that from any paradigm-specific language having a reasonable set of constructs for data definition, like for instance ML, the data sublanguage can be extracted and used as DL in our construction.

In Table 1 and Table 2 we present examples of both categories of languages.

CASL [21]	algebraic specification language
Z [20]	mathematical oriented notation
ML (the purely functional sublanguage of) [11]	functional programming language
PASCAL data sublanguage	the PASCAL sublanguage consisting of the type declarations and of the expressions

Table 1. Data Languages

1.1 Data (Part of a) Language

In order to illustrate the concepts on our running example, let us start with a short introduction on the data language we are going to use, that is CASL.

STATECHART [9,10]	visual notation for reactive systems
CCS with value passing [15]	specification language for processes communicating by handshaking along channels
CCS with value passing and parametric channels	specification language for processes communicating by handshaking along channels, where also channel names may be communicated
Petri nets [17]	visual notation for concurrent parallel systems
PASCAL	imperative programming language
Linear time temporal logic	a logic for processes
ML [11]	functional programming language
UML plus OCL [22]	visual object oriented notation (a subset/profile having a formal semantics)

Table 2. Paradigm Specific Languages

Casl (Common Algebraic Specification Language) The specification language CASL has been designed within the CoFI ¹ initiative, an open group with representatives from almost all the European groups working in this area, started by ESPRIT BRA COMPASS in cooperation with IFIP WG 1.3 (Foundations of Systems Specification) and recently evolved in an ESPRIT Working Group.

Though CASL is reasonably concise and expressive, restrictions (e.g., for interfacing with existing tools) and extensions (e.g., for dealing with specific applications, like reactive systems) are expected to be defined. Indeed, CASL has been planned from the beginning as a family of languages coherent on the common parts and what we call CASL should be more precisely called *basic* CASL.

We have no pretension to describe the syntax nor the semantics of CASL, for which we refer to [21,3] or to the web site:

<http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/index.html>.

We shortly list the main features of CASL, as a reminder for readers already familiar with the language.

Basic specifications allow the declaration (definition) of sorts, subsorts, functions (both partial and total) and predicates. Terms are built, starting from typed variables, by applying functions to terms of any subsort of the expected argument sort (that is, coercion to the supersort is implicit) or projecting a term t of some sort onto a subsort s (t as s). Terms are used to state axioms, that are first-order formulae built from equations, predicate applications (with the same rules as for function application), definedness assertions, and subtyping membership predicates. Finally, datatype declarations are provided for concise specification of sorts together with some constructors and (optional) selectors.

The semantics of a basic specification consists of a signature of the form $(S, TF, PF, P, \leq_S)$ (where S is the set of *sorts*, TF and PF are respectively *total* and *partial functions*, P are *predicates* and $\leq_S$ is the *subsorting relation*) and a class of many-sorted partial first-order structures, i.e., algebras where the functions are partial or total, and where also predicates are allowed.

¹ See <http://www.brics.dk/Projects/CoFI/>.

Structured specifications provide constructs for translation, reduction, union, (free) extension² of specifications. Generic specifications may also be defined. The semantics of structured specifications belongs to the same domain as that of basic specifications.

Other important features of CASL are the *architectural specifications* and the *specifications libraries*, that are not relevant here.

Data Language Taxonomy The parts of a language concerning data are basically those used to describe types (if the language is typed, of course), their elements (the data used in the computations) and to express conditions on the data. Therefore, we are working in a context where we have the following *ingredients*.

Types The types are sets, and are denoted by *type expressions* (in CASL, sorts), that are built using *type constructors*, whose semantics is a function resulting in a set (of values in a suitable universe). Such type constructors are sometimes user defined and introduced by definitional constructs. Moreover, we have a set of rules describing how the type constructors are used to build correct type expressions.

In CASL there are no type constructors, but only a definitional mechanism for specifications, giving the semantics of sorts together with that of functions and predicates.

Elements The elements are the individual elements of types, called *data*. Analogously to the type case, data are denoted by *expressions*, that are built using *operations*, interpreted as functions on data (so having types as source and target) and such operations may be user defined and introduced by definitional constructs. For instance, in CASL operations correspond to functions, introduced by function declarations and function definitions or as constructors/selectors of data types. Again we have rules expressing how operations are used to build correct expressions. In most cases, we are interested not only in closed (constant) expressions, but in open expressions as well, used for instance as bodies of functions (the variables play the role of parameters) or in conditions (the variables are somehow quantified). The evaluation of such expressions is based on concepts like *environments* (or *evaluations*) and the mechanism is always the same: variables are added as possible basic expressions and the evaluation of expressions is based on an extra parameter (the environment) used to give the value of this new basic case, while the semantics of other constructs remains the same, with this silent extra parameter.

In CASL expressions are called terms and are built by function application, starting from constants and (logical) variables. The functions that can be used are those declared in specifications, embedding into supersorts and projections onto subsorts.

Conditions The conditions are logical formulae used to express properties on data, and are built using *predicates* (i.e., functions yielding values in a set of

² Initiality is a special case of this construction.

special *truth values*), *logical connectives* and *quantifiers*. In particular, such predicates are quite commonly user defined and introduced by definitional constructs; this is the case of CASL, where predicates can be user defined. Again we have rules expressing how conditions are built using these ingredients.

In CASL conditions correspond to the non-terminal **FORMULA**, whose elements are partial first-order formulae.

In the following we will use *data language* to denote languages having only the above constructs. In the literature there are several examples where data languages are actually the declarative part of larger languages and we first have to evict that part.

1.2 Paradigm-Specific Language

Let us now sketch a rather simple variant of Hoare's logic to be used as our running example of paradigm-specific language, apt to express requirements on elementary imperative programs, consisting of sequences of statements using only global variables (programs here for short).

HLL (Hoare's Logic Language) A HLL specification consists of a collection of imperative variable declarations, together with a logical constraint on state transformation.

Syntax

```

HSPEC      ::= use DEC*; constr COND
DEC         ::= VAR : TYPE;
TYPE        ::= B_TYPE | Pointer B_TYPE
B_TYPE      ::= Int | Real
COND        ::= ATOM | (COND) | COND ∧ COND | COND ∨ COND | ¬COND | COND ⇒ COND
ATOM        ::= EXP = EXP | IEXP < IEXP | REXP < REXP
EXP         ::= IEXP | REXP | nil | VAR
IEXP        ::= NUM | IEXP + IEXP | IEXP - IEXP | IEXP * IEXP | LEXP
REXP        ::= R-NUM | REXP + REXP | REXP - REXP | REXP * REXP | REXP / REXP | LEXP
LEXP        ::= VAR | VAR↑ | @VAR | @VAR↑

```

The actual form choosen to represent variables and numbers is immaterial; hence we do not list the corresponding productions.

↑ denotes pointer dereferencing and @ the initial value of variables.

The rules for building correct expressions are the standard ones, about the correct typing (and existence) of variables, and are based on static environments $SENV = [VAR \rightarrow_p TYPE]$, associating types with variables. In the following, we use $\rightarrow_p$ to denote partial functions and, for each $f : A \rightarrow_p B$, $D(f) \subseteq A$ to denote the definition domain of f .

Static environment are created by declarations accordingly to the following rules.

$$\frac{}{\vdash \Delta \triangleright \emptyset} \quad \frac{\vdash decs \triangleright se}{\vdash decs \ x : t; \triangleright se[t/x]} \ x \notin D(se)$$

Let us now simply sketch some judgments using static environments. Notice that the judgments on expressions are explicitly giving the type, using the form $se \vdash exp : t$, while those on conditions, for instance, are simply stating their correctness, using the form $se \vdash cond$.

$$\begin{array}{c}
 \frac{}{se \vdash x : se(x)} \quad x \in D(se) \qquad \frac{se \vdash x : t}{se \vdash @x : t} \\
 \\
 \frac{se \vdash x : \mathbf{Pointer} \ t}{se \vdash x \uparrow : t} \qquad \frac{se \vdash x : \mathbf{Pointer} \ t}{se \vdash @x \uparrow : t} \\
 \\
 \frac{}{se \vdash \mathbf{nil} : \mathbf{Pointer} \ t} \qquad \frac{se \vdash exp_1 : t \quad se \vdash exp_2 : t}{se \vdash exp_1 = exp_2}
 \end{array}$$

Semantics HLL is used to express properties on *programs*, written in any given imperative language. Programs are classified accordingly to the typed variables they use, i.e., a static environment, and their semantics describes a state transformation.

In the following, let $\wp$ denote the power set function and $\mathbb{B}$ the domain of standard first-order logic; we will denote truth by T and falsity by F , and use the standard connectives of such logic with standard notation as well. Moreover, let LOC be a denumerable set (of locations) and $\underline{n} \notin LOC$ a distinct value corresponding to the **nil** pointer.

$$\begin{aligned}
 VALUE &= \mathbb{R} \cup \mathbb{Z} \cup LOC \cup \{\underline{n}\} \\
 ENV &= [\mathbf{VAR} \rightarrow_p LOC] \\
 STATE &= [LOC \rightarrow_p VALUE] \\
 \mathcal{P} : \mathbf{HSPEC} &\rightarrow [[(ENV \times STATE) \rightarrow_p STATE] \rightarrow \mathbb{B}] \\
 \mathcal{T} : \mathbf{TYPE} &\rightarrow \wp(VALUE) \\
 \mathcal{D} : \mathbf{DEC}^* &\rightarrow \mathbf{SENV} \\
 \mathcal{C} : \mathbf{COND} &\rightarrow [(ENV \times STATE \times STATE) \rightarrow \mathbb{B}] \\
 \mathcal{E} : \mathbf{EXP} &\rightarrow [(ENV \times STATE \times STATE) \rightarrow_p VALUE]
 \end{aligned}$$

Since locations are not typed and there are elements (like $\underline{n}$) having more than one type, we have to introduce a mechanism to match the static information to environments and states.

For each static environment se let $(ENV \times STATE)_{se}$ consist of all pairs (ρ, σ) s.t. $D(se) = D(\rho)$ and for all x s.t. $se(x) = t$ if $\rho(x) \in D(\sigma)$ then $\sigma(\rho(x)) \in T(t)$. Moreover, let $Prog_{se}$ be the set of *programs* correctly using the typed variables provided by se , that is $Prog_{se} \subseteq [(ENV \times STATE)_{se} \rightarrow_p STATE]$, and for all $p \in Prog_{se}$ if $p(\rho, \sigma) = \sigma'$ then $(\rho, \sigma') \in (ENV \times STATE)_{se}$.

We define the semantic functions by induction, here we just quote some rules, omitting for instance the meaning of operations and predicates that we assume obvious.

- $\mathcal{P}(\mathbf{use} \ decs ; \mathbf{constr} \ cond)(p) = T$ iff
 $p \in Prog_{\mathcal{D}(decs)}$ and $\mathcal{C}(cond)_{\rho, \sigma, p(\rho, \sigma)} = T$ for all $(\rho, \sigma) \in D(p)$.
- $\mathcal{D}(decs) = se$ iff $\vdash decs \triangleright se$

- $\mathcal{T}(\text{Int}) = \mathbb{Z}$, $\mathcal{T}(\text{Real}) = \mathbb{R}$, $\mathcal{T}(\text{Pointer } t) = \text{LOC} \cup \{\underline{n}\}$
- the semantics of logical connectives is standard. Let us see, for instance, the rule for the equality atoms:

$$\mathcal{C}(\text{exp}_1 = \text{exp}_2)_{\rho, \sigma, \sigma'} = \begin{cases} T & \text{if } \mathcal{E}(\text{exp}_1)_{\rho, \sigma, \sigma'} = v = \mathcal{E}(\text{exp}_2)_{\rho, \sigma, \sigma'} \\ F & \text{otherwise} \end{cases}$$
- the rules for the integer and real operations are the obvious ones, relying on the definition of the corresponding mathematical functions; moreover we have

$$\begin{array}{ll} \mathcal{E}(x)_{\rho, \sigma, \sigma'} = \sigma'(\rho(x)) & \mathcal{E}(x\uparrow)_{\rho, \sigma, \sigma'} = \sigma'(\sigma'(\rho(x))) \\ \mathcal{E}(\text{@}x)_{\rho, \sigma, \sigma'} = \sigma(\rho(x)) & \mathcal{E}(\text{@}x\uparrow)_{\rho, \sigma, \sigma'} = \sigma(\sigma(\rho(x))) \end{array}$$

Paradigm Specific Language Taxonomy Consider now a paradigm-specific language PSL. A first distinction we make on its constructs is given by the context needed in order to give them a semantics. Indeed, we have a *data* part, whose semantics is based on set(s) of values, functions and predicates, and a *paradigm-specific* part, whose semantics requires information about the context they are used in. For instance, if we consider HLL, we have that the constructs for B-TYPE, those for REXP and IEXP giving the real and integer operations, and COND are the data part, while pointer dereferencing and imperative variables are the paradigm-specific part, as they need an environment and a state in order to be evaluated.

Let us make a more refined analysis within each category of constructs. The results of this analysis for the paradigm-specific languages in Table 2 will be sketched in Table 3.

Data Constructs

paradigm-specific datatypes $\text{PSL}_{\text{par-spec}}$ are the datatypes that are specific to the language paradigm (i.e., such that some construct of the language cannot be properly used/defined if we drop one of them).

Some examples are time for statecharts, boolean for CCS with value passing and ML, sets of object identities for UML, and pointers, with the constant `nil`, for HLL.

basic data language $\text{PSL}_{\text{basic}}$ is the sublanguage of PSL used to handle the data that are not paradigm-specific, that is those that are not crucial for the language and can be safely substituted; it is a data language.

This part is empty for several paradigm-specific languages (e.g., classical Petri nets), or it is very poor, consisting just of a few predefined datatypes. Some examples are integers and reals for HLL, the part of ML concerning the “datatypes” and the “abstract datatypes”, and the generic values for CCS with value passing.

The combination of $\text{PSL}_{\text{basic}} \oplus \text{PSL}_{\text{par-spec}}$ is the sublanguage of PSL that is a data language.

Paradigm-Specific Constructs

pseudo-data constructs $\text{PSL}_{\text{pseudo}}$ are those language constructs (constituents) used to build correct expressions (type expressions, conditions) that are not directly denoting a data or an operation but need some extra input to yield one.

Some examples are updatable variables in imperative languages (yielding a value for given environment and state), function parameters in functional languages (yielding a value for given environment built by the call), the system function returning the current time (yielding a value given the state), the “happened” construct of statecharts (checking whether an event has happened and yielding a boolean value for given history of the system). In particular, in HLL, we have imperative variables and pointer dereferencing.

kernel constructs PSL_{ker} are those language constructs (constituents) that are not involved in data description, though they can *use* data.

Some examples are statements and subroutines in imperative languages, methods, classes and objects in object-oriented languages, the diagrams of visual languages, as UML and STATECHART. In HLL this part is the main construct of HSPEC.

2 A Method for the Combination

Let us now shortly sketch how we can safely realize the combination of the two languages, that will be made at the level of the abstract syntax, as it is where their structure is more evident. We cannot, of course, give a method for the combination of *any* pair of languages; thus let us start with some assumptions.

2.1 Assumption on DL and PSL

First, we assume that both in the data and in the paradigm-specific languages, there are just one syntactic category corresponding to type expressions, one corresponding to expressions and one corresponding to conditions. This requirement makes clear the correspondence among the syntactic categories of DL with the corresponding categories of PSL.

For each of these categories, the possible productions are in the following forms:

TYPE	::= TYPE-CONST(TYPE*) TYPE-ITER(DEC*, {TYPE EXP COND}*)
EXP	::= OP(EXP*) EXP-ITER(DEC*, {TYPE EXP COND}*) VAR
COND	::= PRED(EXP*) CON(COND*) COND-ITER(DEC*, {TYPE EXP COND}*)
DEC	::= VAR : TYPE;

where TYPE-CONST are the type constructors, OP are operations, PRED are predicates and CON are logical connectives. That is, we have two kinds of constructs for each non-terminal, that are the application of some constructor to its arguments and iterators, binding typed variables (representing elements) in open terms.

PSL	data part		paradigm-specific part	
	PSL _{basic}	PSL _{par+spec}	PSL _{pseudo}	PSL _{ker}
STATECHART	integers, chars, ...	time, <i>states</i> (of the chart), <i>events</i>	variables, happened, is_in_state, since, before, at	visual part, actions
CCS with value passing	generic values	booleans	variables used in input prefixes	combinators ($\llbracket \cdot \rrbracket, +, \dots$)
CCS with parametric channels	generic values	booleans, <i>channels</i>	variables used in input prefixes	combinators ($\llbracket \cdot \rrbracket, +, \dots$)
Petri nets		sets of tokens with operations $\subseteq, \cup, -$ and $\{-\}$		visual part
PASCAL	integers, reals, array, record...	boolean, pointer types, ordered types, enumeration types	constants, variables, parameters of functions and procedures, function calls, pointer dereferencing	declarations except those for types, statements
UML with OCL	OCL basic types	classes (evaluated as sets of object identities), <i>associations</i> , <i>states</i> of statecharts, <i>type of all types</i> , collections (sequences, sets, bags), boolean, iterators, has_type, ... many others	attribute selection, right/left application of association, all instances of a class, operation parameters... many others	UML
Linear time (homogeneous) temporal logic	a unique type, with functions and predicates defining the logical language, logical connectives and rigid quantifications		flexible variables, temporal connectives	
ML	basic types, datatypes, abstract datatypes	boolean, functional and product types	function parameters	function declaration

where the spurious type are emphasized.

Table 3. Taxonomy of Paradigm-Specific Languages

Some examples of iterators are λ (returning functional values), the logical quantifiers, the subsort definition by means of a condition in CASL, and the set-comprehension notation. As usual, the language will consist only of elements without free occurrences of variables.

The static semantics has to provide correctness judgments, using static environment memorizing context information (as the types of the introduced variables).

In general, languages are given by grammars that do not comply to the proposed schema. Therefore, we may have to massage somehow the abstract syntax in order to put it in the correct form, moving some information into the static semantics part.

This is, for instance, the case of HLL; so we rephrase its grammar accordingly to the wanted pattern³.

```

HSPEC      ::= use DEC*; constr COND
DEC         ::= VAR : TYPE;
TYPE        ::= Int | Real | Pointer TYPE
COND        ::= PRED(EXP*) | CON(COND*)
PRED        ::= =|<
CON         ::=  $\wedge$  |  $\vee$  |  $\neg$  |  $\Rightarrow$ 
EXP         ::= OP(EXP*) | VAR | @EXP | EXP $\uparrow$ 
OP          ::= + | * | - | / | nil | ...
    
```

Extra judgements have to be added to the static semantics, like for instance those restricting the application of the pointer type constructor to basic types only.

$$\frac{}{\vdash \text{Int} : BType} \quad \frac{}{\vdash \text{Real} : BType} \quad \frac{\vdash t : BType}{\vdash \text{Pointer } t : PType}$$

Moreover, some rules have to be restricted, because some verifications originally managed by the context-free grammar have to be incorporated.

This is the case, for instance, of the rule for variable declaration, where the correctness of the type expression has to be checked as well

$$\frac{\vdash \text{decs} \triangleright se}{\vdash \text{decs } x : t; \triangleright se[t/x]} \quad x \notin D(se) \text{ and } (\vdash t : BType \text{ or } \vdash t : PType)$$

or pointer dereferencing, where the expression has to be a pointer variable.

$$\frac{se \vdash x : \text{Pointer } t}{se \vdash x \uparrow : t} \quad x \in \text{VAR} \text{ and } \vdash \text{Pointer } t : PType \text{ or } \vdash t : PType$$

There is no need for changing the semantics, as the language generated is still the same. The only difference between this version and the original one is in the way the language is defined, because several “errors” that were intercepted by a finer context free grammar are now captured by the static semantics rules.

Having the static checks of either language given in this way allows us to immediatly apply them to the terms produced by constructs of the other language as well, while if the same requirements are captured using more terminals

³ This is the case for CASL as well, but we are not going into the details of the rephrasing that is straightforward but long to be described.

we should somehow distribute the new cases (productions) among the auxiliary non-terminals, possibly introducing other categories as well.

In several examples the construction schema for **TYPE** (**EXP**, **COND**) requires some extra ingredients (other syntactic categories). In such cases the standard solution is to introduce *spurious* types, with very restricted uses, whose semantics (that is a set) has to be provided as well. For instance, the condition **is_in**(*st*), that appear in the data language of statecharts, uses *st* that is a state of the statechart and, thus, not a data of the language. Technically, we can deal with this, by introducing a spurious type **STATE**, denoting the set of the states of the statechart itself. This technique is also followed in the constraint language of UML (OCL), where classes, associations etc. are introduced as extra types.

The (dynamic) semantics has to be given compositionally following the structure of the abstract syntax, only for correct elements, of course.

2.2 Adding Paradigm-Specific Data

To combine a language **PSL** with a data language (a data component) **DL** means essentially to replace **PSL_{basic}** with **DL**. This substitution is done in two steps, accordingly to the stratification among the constructs of **PSL** induced by our analysis on the use of data in **PSL**.

The first step is the combination of the “new” data from **DL** with the paradigm-specific data of **PSL**, $\mathbf{DL}_{\text{ext}} = \mathbf{DL} \oplus \mathbf{PSL}_{\text{par-spec}}$, while the second one will be discussed in the next session.

There are (at least) two cases, general enough to jointly cover most pragmatic examples, that guarantee a safe combination of **DL** and **PSL_{par-spec}**:

- the datatypes in **PSL_{par-spec}** can be implemented or realized in **DL**; in particular this is the case when they already are included in **DL** (e.g., a boolean type or a time type realized by a **CASL** specification or a **Z** type). This is indeed the technique we adopt for the combination of **HLL** and **CASL**, using the following specifications:

- the first is the axiomatization of the domain of locations; as we only know only that it is a denumerable set, we inductively build it starting from the “first location” and a “successor” operation.

spec LOC = freetype Loc ::= loc₀ | new(Loc)

- using the above specification of locations, we build the data type of pointers for any given type (sort) *s*, that is (up to isomorphism) the disjoint union of locations and the extra value to represent the nil operation.

**spec POINTER[sort *s*] given LOC =
freetype Pointer[*s*] ::= nil | sort Loc**

These “standard” declarations should be considered implicitly added to any program of **DL** used in the combination language. If this approach is possible, it is definitely the most convenient, as the combination **DL_{ext}** is the same as **DL**, in the sense that each data description in it is also a data description in **DL**, and hence tools, theorems and methodologies can be directly reused.

- the types in DL are defined by means of type constructors, that expect as input only other types and hence their semantics is a function over sets of data. In this case we simply have to add to DL some new type constructors, with the related operations and predicates, corresponding to the datatypes in $\text{PSL}_{\text{par-spec}}$ and their integration is given by the fact that the DL type constructors can deal with any set⁴ given in input, so in particular with the semantic interpretation of the paradigm-specific datatypes.

This setting is general enough; it is satisfied for instance by most programming languages, but not by standard algebraic specification languages (where the semantics of type expressions is given together with, and mutually dependent on, the semantics of operations and predicates).

At the end of this phase, we have built a new data language describing a class of structures. Each structure in the class provides the semantics for type expressions, operations and predicates; hence it is some kind of algebra (first-order structure). Now, this “algebra” has to be used as basis for the semantics of the rest of the combination language.

2.3 Adding the Paradigm-Specific Part

We have to adjust the pseudo-data of PSL, as well as the kernel constructs, to work on the data language DL_{ext} .

In general this is possible, because their semantics is described in a sort of generic or polymorphic compositional style, working on any structure (collection of sets, functions and predicates, so basically an algebra) satisfying some requirements. In most cases, the kind of algebras built by DL_{ext} (from now on DL-algebras) is not the same as the kind of algebras expected by the paradigm-specific constructs of PSL (from now on PSL-algebras). Therefore, we have to provide a *conversion function* from DL-algebras into PSL-algebras. This is usually possible if all concepts expressible in DL-algebras are expressible in PSL-algebras as well. For instance, if we use as DL the algebraic specification language CASL, then the DL-algebras are partial algebras. Now, if we want to combine CASL with an imperative language, whose semantics is given in terms of domains, then we have to transform the representation of undefinedness of partial algebras into the representation of undefinedness of domains, that is the bottom element. Since it is straightforward to make such conversion (by standard strict totalization techniques), we are able to combine such languages. But if we try to combine CASL with a paradigm-specific language that has no notion of partiality, then this cannot be done preserving the semantics and the methodology of the paradigm-specific language. In such case, hence, our method cannot be applied.

If we are able to make the conversion from DL-algebras into PSL-algebras, then the compositionality of the semantics allows to delegate the evaluation

⁴ Of course some compatibility among the universes fixed for the semantics of the two languages has to be assumed.

of subcomponents to the (known) semantics of either DL_{ext} or PSL , and the genericity guarantees that the semantic description of the leading operation is able to cope with the results of the subcomputations.

The key points are the adaptation of the extra domains present in the semantics for interpreting the pseudo-data to include the “new” values and the uniform addition of such domains as (silent) arguments to the evaluation rules of data operations from DL_{ext} .

For instance, in our toy example, we have to modify the definition of the set *VALUE* to include the universe of CASL values (hence indirectly modifying *STATE* as well). The evaluation rules for variables and pointers, both before and after execution, are formulated disregarding the actual definition of *VALUES*, so they are sufficiently generic to be able to deal with the new values as well.

2.4 The Resulting Combination

If we have successfully finished our process, we have now a language $DL \oplus PSL$ where a “program” consists of a declarative part, that is an acceptable declarative part in DL_{ext} , and a “program” in PSL without the declarative part (if any).

The semantics is given in three steps:

- the declarative part is used to find a (class of) DL -algebra(s)
- DL -algebras are transformed into PSL -algebras
- the result of the previous step is used as context to define the semantics of the “ PSL -program”, using the rules of the semantics of PSL .

Casl $\oplus$ HLL Let us see the final result of our technique on the toy example we are using to illustrate the methodology.

Syntax The overall grammar consists of all productions of the CASL grammar (in the version rephrased accordingly to the assumptions in Sect. 2.1) plus the following.

HSPEC	::= CASL-SPEC use DEC* ; constr COND
DEC	::= I-VAR : TYPE;
TYPE	::= SORT (a nonterminal of the CASL grammar)
COND	::= all CASL productions for FORMULA, with TERM replaced by EXP and FORMULA by COND
EXP	::= all CASL productions for TERM, with TERM replaced by EXP I-VAR @EXP EXP↑

Notice that in the above grammar there are two kinds of variables: *VAR* is the nonterminal for logical variables of the CASL grammar while *I-VAR* is the nonterminal for imperative (updatable) variables. The new cases of the production for *EXP* concern the pseudo-data constructs.

Static Semantics Let us consider a specification *spec use decs; constr cond*.

In order to verify its static correctness, we first use the CASL judgments on a specification *spec+pointers*, where locations and pointers for each sort declared in *spec* are routinely added.

spec *spec+pointers* =
 LOC and

$$\left. \begin{array}{l} \text{POINTER}[\text{sort } s_1] \text{ and} \\ \vdots \\ \text{POINTER}[\text{sort } s_n] \text{ and} \end{array} \right\} s_1 \dots s_n \text{ are the sorts declared in } \textit{spec}$$

then *spec*

Assuming that *spec+pointers* is statically correct, this first step produces a signature $\Sigma = (S, TF, PF, P, \leq_S)$, giving the sorts, (total and partial) functions and predicates that can be used in the following parts.

Then, we use the rules for creating static environments from HLL static semantics, replacing

$$\frac{}{\vdash \text{Int} : BType} \quad \frac{}{\vdash \text{Real} : BType} \quad \text{by} \quad \frac{}{\vdash s : BType} \text{ } s \text{ declared in } \textit{spec}.$$

Again assuming that *decs* is statically correct, this second step produces a static environment *se* recording which typed imperative variables can be used in the last part.

Finally, we use the CASL judgements replacing **TERM** and **FORMULA** by **EXP** and **COND**, respectively, adding a silent extra parameter, the static environment, to each judgement and adding also the judgements for the pseudo-data constructs:

$$\frac{x \in D(se) \quad se(x) = s \quad s \in S}{(S, TF, PF, P), se \vdash x \triangleright x_s} \quad \frac{x \in D(se) \quad se(x) = s \quad s \in S}{(S, TF, PF, P), se \vdash @x \triangleright (@x)_s}$$

$$\frac{x \in D(se) \quad se(x) = \text{Pointer } s}{(S, TF, PF, P), se \vdash x \uparrow \triangleright (x \uparrow)_s} \vdash \text{Pointer } s : PType$$

It is interesting to note that these judgments are the obvious adaptation of those given for HLL to CASL style, where the correctness of expressions (terms) is described translating them into unambiguous forms, with typing captured by decoration, and requiring that all the forms for one term are equivalent (accordingly to the rules given for subsorting).

Semantics

Let us consider a statically correct specification *spec use decs; constr cond*.

Analogously to the procedure adopted for the static semantics, and using the same notation, we give the semantics of the combination language in three steps.

First the semantics of the CASL specification *spec* determines a class of algebras $\mathcal{M}$ on a signature Σ , that is defined by the standard semantics of CASL for the specification *spec+pointers*. Then, *decs* is evaluated, as in the static semantics, producing a static environment *se*. Finally for each algebra in $\mathcal{M}$ we determine the semantics of the remaining part of the program.

Let us fix a model of *spec+pointers*, that is a Σ -algebra $D \in \mathcal{M}$. In the following let us denote the set of valuations of the logical variables (used by

the quantifiers) by VAL , with V_O as empty valuation, i.e., the totally undefined function.

The semantics functions evaluating expressions and conditions now need as input both logical variables evaluations (as in CASL) and environment and state (as in HLL).

$$\begin{aligned}
VALUE_D &= \bigcup_{s \in \text{Sorts}(\Sigma)} D_s \\
VAL_D &= [\text{VAR} \rightarrow_p VALUE_D] \\
LOC_D &= D_{\text{Loc}} \\
ENV_D &= [\text{I-VAR} \rightarrow_p LOC_D] \\
STATE_D &= [LOC_D \rightarrow_p VALUE_D] \\
\mathcal{P} : \text{HSPEC} &\rightarrow [((ENV_D \times STATE_D) \rightarrow_p STATE_D) \rightarrow \mathbb{B}] \\
\mathcal{D} : \text{DEC}^* &\rightarrow \text{SENV} \\
\mathcal{C} : \text{COND} &\rightarrow [(ENV_D \times VAL_D \times STATE_D \times STATE_D) \rightarrow \mathbb{B}] \\
\mathcal{E} : \text{EXP} &\rightarrow [(ENV_D \times VAL_D \times STATE_D \times STATE_D) \rightarrow_p VALUE_D] \\
\mathcal{T} : \text{TYPE} &\rightarrow \wp(VALUE_D)
\end{aligned}$$

The semantic functions are defined by induction, using the same rules given, respectively, in CASL and HLL semantics. For instance $\mathcal{T}(t) = D_t$ for each type (sort) t as in CASL.

If necessary, the rules of each language are extended with the extra silent parameters needed by the other language. Like for instance in the definition of the semantics of the main production, where the empty logical variable valuation has to be added to be able to evaluate the constraint:

$$\begin{aligned}
\mathcal{P}(\text{spec use decs; constr cond})(p) &= \\
&\begin{cases} T & \text{if } p \in \text{Prog}_{\mathcal{D}(\text{decs})} \text{ and } \mathcal{C}(\text{cond})_{\rho, V_O, \sigma, p(\rho, \sigma)} = T \text{ for all } (\rho, \sigma) \in D(p) \\ F & \text{otherwise} \end{cases}
\end{aligned}$$

Let us see a few examples of rules for expressions, that are those where the interaction between CASL and HLL is more significant. The last two are the modification of standard evaluation in algebraic languages (and CASL in particular) for, respectively, function application and logical variable, while the others are the same given for HLL, with a logical variable evaluation as extra (silent) parameter.

$$\begin{aligned}
\mathcal{E}(x)_{\rho, V, \sigma, \sigma'} &= \sigma'(\rho(x)) & \mathcal{E}(x\uparrow)_{\rho, V, \sigma, \sigma'} &= \sigma'(\sigma'(\rho(x))) \\
\mathcal{E}(\textcircled{x})_{\rho, V, \sigma, \sigma'} &= \sigma(\rho(x)) & \mathcal{E}(\textcircled{x}\uparrow)_{\rho, V, \sigma, \sigma'} &= \sigma(\sigma(\rho)) \\
\mathcal{E}(f(\text{exp}_1, \dots, \text{exp}_n))_{\rho, V, \sigma, \sigma'} &= f^D(\mathcal{E}(\text{exp}_1)_{\rho, V, \sigma, \sigma'}, \dots, \mathcal{E}(\text{exp}_n)_{\rho, V, \sigma, \sigma'}) \\
\mathcal{E}(y)_{\rho, V, \sigma, \sigma'} &= V(y)
\end{aligned}$$

3 Analysing UML

Though the time is not yet ripe for presenting a full application of the method outlined in this paper to UML, the OMG standard object-oriented notation for specifying, visualizing, constructing, and documenting software systems ([22,8]), it is interesting to look at UML for the following reasons:

- what is called UML is an outstanding example of a “language design” based on a factorization principle, very much along the lines we have discussed;
- thus, our analysis applied to UML may help understand more deeply its nature and provide a basis for variations and improvements (for example a lot of work for a better standard is underway);
- UML is a *family* of languages, where the members may differ not only because the semantics of some constructs is not fixed (the so-called *semantic variation points*), but also as part of the constructs, e.g., those for the expressions and the actions (the UML terminology for statements), are not defined by the standard but may be imported from other languages, and UML has mechanisms to extend the language itself by defining new constructs (e.g., stereotypes). This encourages the definition of variants using different data languages. Thus, our method of combination, where most work is restricted to the taxonomy of the paradigm-specific language (UML in this case), is especially convenient, because the taxonomy itself could be reused many times.

Unfortunately, an immediate application of our technique to UML is ruled out not only by the lack of a formal semantics for UML, that has not been yet provided, though several attempts are under way, at least for some parts, but also by the visual nature of UML. Indeed, UML “programs” are not strings but instead bunches of diagrams (variants of graphs) possibly annotated with some text. For this reason, its syntax is not given, as usual, by a BNF grammar; instead it is given by the so called *metamodel*, which is essentially an object-oriented description of the UML models⁵. Consequently the “static semantics” of UML is given by means of constraints on such metamodel.

However, we believe that the adaptations required to apply our techniques to UML may be, though non-trivially, worked-out. Therefore, let us sketch briefly how our taxonomy technique apply to UML.

For what concerns the data constructs, following our terminology, we have that UML is, essentially⁶, a kernel language without data constructs, whereas all data related constructs can be plugged in to obtain a complete language. The OMG has, however, provided a language for the data constructs called OCL (for Object Constraint Language) [22], so that what is usually known under the name “UML” is instead the combination $\text{UML-OCL} = \text{OCL} \oplus \text{UML}$. With respect to this combination the UML part is, in our terminology, the kernel $\text{UML-OCL}_{\text{ker}}$.

Thus, the analysis of UML-OCL from a data-constructs viewpoint is in some sense simplified, because the data constructs are already grouped within the OCL component. The only OCL constructs that are not data related are those allowing to attach a constraint (a condition) to an element of an UML model.

⁵ Using the UML terminology the UML specifications/programs are called *models*.

⁶ We cannot say exactly, because UML allows to define special classes of objects without changeable state and without identity that may be considered a kind of datatypes.

According to our setting, we can thus decompose OCL in the three canonical parts: $\text{OCL} = \text{OCL}_{\text{basic}} \oplus \text{OCL}_{\text{par-spec}} \oplus \text{OCL}_{\text{pseudo}}$

Let us give an idea of what is in each part.

Basic data language $\text{OCL}_{\text{basic}}$ consists of some standard basic predefined types (*Integer*, *Real*), and of a “type declaration mechanism”, precisely the possibility of defining simple enumeration types by listing their elements.

Paradigm-specific datatypes $\text{OCL}_{\text{par-spec}}$ is rather rich; indeed it includes two standard types (*String* and *Boolean*); a declaration mechanism for types (the fact that any classifier appearing in the UML model defines a type with the same name of the class), and many *spurious* datatypes, see 3, (e.g., *OclState* that is the type whose elements are the states of the state machines appearing in the model).

The most interesting paradigm specific types are however the “collection types”, precisely the three collection-like parametric types, *Set*, *Bag*, and *Sequence*, plus the common abstract supertype *Collection*. All such collections types have the usual operations (union, add an element, size, ...) but also some more peculiar ones, as an operation transforming a set/bag into a sequence, which may be nondeterministic.

OCL offers also a rather general *iterator* construct for acting over the values of the collection types, which has several useful particular instantiations (as set-comprehension and a form of universal and existential quantification over collections). Its general form is

$$\text{collection} \rightarrow \text{iterate}\{ x: \text{type}; \text{acc}: \text{type}' = \text{exp} \mid \text{exp}' \}$$

where x and acc are variables that may appear in exp' , and exp is a ground expression. The value denoted by this construct is computed as follows:

x iterates over *collection* and exp' is evaluated for each elem. After each evaluation of exp' , its value is assigned to acc . exp gives the initial value of acc . In this way, the value of acc is built up during the iteration over *collection*; when *collection* is a set or a bag the iterator may be nondeterministic too.

Pseudo-data constructs $\text{OCL}_{\text{pseudo}}$ Among these constructs we have the one allowing “navigation”: given an object O and an association AS , it returns all objects (a collection) associated with O by AS . Another one is **@pre** used only in post-conditions: it is postfix to attributes, operations, methods, and associations, and it denotes that such construct should have been computed before the execution of the operation/method call.

Our analysis supports how to approach a variation of UML-OCL, for example modifying the basic data part accordingly to the technique proposed here. However, since UML standard does not include OCL, that is just an example of a data-component among several possibilities, the problem of plugging data into the UML is different from the general case considered before in this paper. Indeed, we could propose different paradigm specific and/or pseudo-data constructs w.r.t. those present in OCL using an analogous technique to that illustrated here for the basic data and still have an acceptable language of the

UML family. We think that it is interesting, however, to first analyse the paradigm specific and pseudo-data components, understand the needs that they cover, and then perhaps to propose new constructs, or just a revised version, covering these aspects.

4 Conclusions

In the course of a two decades experience in the field of formal specifications, we have witnessed the existence of a variety of languages, where the data part not only play different roles, but sometime is partly undefined or inadequated for the applications.

Last, but not least, we are facing the emergence of UML, a de facto standard in system development, where the data are treated separately and even in a non-standardized form. There are, and there will be, attempts to improve the data part of UML, both for sake of clarity and of further expressiveness; here we only mention the CoFI⁷ initiative, within which some activity is going on in that direction.

We have been naturally led to propose, first for our personal understanding, a non standard analysis of the way the data part is embodied into a language, to be able to modify a language structure in a modular way. Our proposal may serve as a methodological guideline in the design and the upgrade phase of a language.

References

1. E. Astesiano and G. Reggio. Labelled Transition Logic: An Outline. Technical Report DISI-TR-96-20, DISI - Università di Genova, Italy, 1996.
2. E. Battiston, F. De Cindio, and G. Mauri. OBJSA Nets: a Class of High-Level Nets Having Objects as Domains. In G. Rozemberg, editor, *Advances in Petri Nets*, number 340 in Lecture Notes in Computer Science, pages 20-43. Springer Verlag, Berlin, 1988.
3. H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P. Mosses, D. Sannella, and A. Tarlecki. Formal Methods '99 - CASL, The Common Algebraic Specification Language - Semantics. Compact disc published by Springer-Verlag, 1999.
4. J.A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1-66. Addison-Wesley, 1989.
5. R. Bussow, R. Geisler, and M. Klar. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. In E. Astesiano, editor, *Proc. FASE'98*, number 1382 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1998.
6. C. Dimitrovici and U. Hummert. Composition of Algebraic High-Level Nets. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, number 534 in Lecture Notes in Computer Science, pages 52-73. Springer Verlag, Berlin, 1991.

⁷ For more information see <http://www.brics.dk/Projects/CoFI/>.

7. H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An Algebraic Specification Language with two Levels of Semantics. Technical Report 83-01, TUB, Berlin, 1983.
8. M. Fowler and K. Scott. *UML Distilled*. Object Technology Series. Addison-Wesley, 1997.
9. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
10. D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The State-mate Approach*. McGraw Hill, 1998.
11. R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, LFCS-University of Edinburgh, 1986.
12. I.S.O. ISO 8807 Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. ISO, 1989.
13. S. Mauw and G.J. Veltink. An Introduction to PSF_d . In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89, Vol. 2*, number 352 in Lecture Notes in Computer Science, pages 272 – 285. Springer Verlag, Berlin, 1989.
14. S. Mauw and G.J. Veltink. A Process Specification Formalism. *Fundamenta Informaticae*, (XIII):85–139, 1990.
15. R. Milner. *A Calculus of Communicating Systems*. Number 92 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1980.
16. G. Reggio and L. Repetto. CASL-CHART: A Combination of Statecharts and of the Algebraic Specification Language CASL. In this volume.
17. W. Reisig. *Petri Nets: an Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer Verlag, Berlin, 1985.
18. W. Reisig. Petri Nets and Algebraic Specifications. *T.C.S.*, 80:1–34, 1991.
19. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
20. M. Spivey. *The Z Notation*. Prentice-Hall, 1992.
21. The CoFI Task Group on Language Design. Formal Methods '99 - CASL, The Common Algebraic Specification Language - Summary. Available on compact disc published by Springer-Verlag, 1999.
22. UML Revision Task Force. *OMG UML Specification*, 1999. Available at <http://uml.shl.com>.
23. J. Vautherin. Parallel System Specifications with Coloured Petri Nets and Algebraic Data Types. In G. Rozemberg, editor, *Advances in Petri Nets*, number 266 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 1987.
24. M. Weber. Combining Statecharts and Z for the Design of Safety-Critical Control Systems. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, number 1051 in Lecture Notes in Computer Science, pages 307–326. Springer Verlag, Berlin, 1996.

An ASM Semantics for UML Activity Diagrams

Egon Börger¹, Alessandra Cavarra², and Elvinia Riccobene²

¹ Dipartimento di Informatica - Università di Pisa - C.so Italia, 40 - 50125 Pisa

`boerger@di.unipi.it`

(currently visiting Microsoft Research, Redmond)

² Dipartimento di Matematica e Informatica - Università di Catania -

V.le A. Doria, 6 - 95125 Catania

`{cavarra, riccobene}@dmi.unict.it`

Abstract. We provide a rigorous semantics for one of the central diagram types which are used in UML for the description of dynamical system behavior, namely *activity diagrams*. We resolve for these diagrams some of the ambiguities which arise from different interpretations of UML models. Since we phrase our definition in terms of Abstract State Machines, we define at the same time an interesting subclass of ASMs, offering the possibility to exploit the UML tool support for using these special ASMs in the practice of software design. We apply these *Activity Diagram Machines* for a succinct definition of the semantics of OCCAM.

1 Introduction

In [1] it is stated that “UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block”. Indeed the so called *precise semantics of UML* [14] gives an unambiguous textual definition of the syntax for UML notations, nevertheless it leaves the behavioral content of various UML constructs largely open.

In this paper we concentrate upon one of the principal diagram types which are used in UML for the description of dynamical system behavior, namely activity diagrams, and provide a rigorous definition of their dynamics. Thereby we assign to such diagrams a precise meaning, i.e. semantical content in the traditional sense of the word. In doing this we resolve for activity diagrams some of the ambiguities which arise from possible different interpretation of UML models [6,5,7]. Through our definitions we also build a framework for rigorous description and analysis of other possible interpretations of the intuitions which underly UML concepts. We hope that this will contribute to their rational reconstruction and to the rational comparison of different implementations.

Since we phrase our definition in terms of Abstract State Machines [8], we provide at the same time an interesting subclass of ASMs, namely Activity Diagram Machines, whose use in software design is supported by UML tools.

2 UML Notation for Activity Diagrams

An activity diagram is a flowchart which exhibits the flow of control between the sequential or concurrent activities of a computational process. UML activity diagrams commonly contain the following types of nodes (also called *states*) and arcs.

Initial nodes, exactly one per diagram, for the default starting place. **Final nodes** indicate where the control flow terminates. **Action nodes** represent an atomic action to be performed in zero (“insignificant”) time. Atomic actions are considered not to be decomposable, their execution not to be interruptible. Multiple invocations of action nodes might be executed concurrently. **Activity nodes** represent a “nested state structure”, a flow of control made up of other activity and action states. Zooming into an activity state, one finds an entire activity diagram which is entered by entering its (unique) initial state. Activity states are decomposable, the computation associated to them is considered to take some time to complete and to be interruptible by the occurrence of (external or internal) events. Activity nodes can come with entry and exit actions to be performed on entering/leaving the node respectively. **Transitions** specify the path from one action or activity to the next. A transition can be labeled by an *event* (so called triggered transitions) and by a *guard*, indicating that the transition is supposed to fire only if the event occurs and the guard condition is met. In UML events are never associated to (arcs coming out from) action states. **Branch nodes** specify alternative paths and have one incoming transition and two or more outgoing ones. Each outgoing transition has an associated Boolean expression called guard. The guards are supposed to be mutually disjoint and to cover all the possible cases. **Fork/Join bars** split a single flow of control into two or more concurrent flows of control respectively synchronize two or more concurrent flows of control. In UML the coroutinability condition is assumed whereby a) the number of flows that leave a fork must match the number of flows that enter its corresponding join (*balancing property*, see section 3.1), and b) parallel activities may communicate with each other by messages. These (pairs of) composite and concurrent fork/join bars in UML activity diagrams delimit the *concurrent* nodes, to be distinguished from the activity nodes which represent sequential composite nodes. Entering a concurrent node (state) means to activate all of its subdiagrams so that at each moment in each of these subdiagrams at least one state is active, until exiting the corresponding join becomes possible when and only if all subdiagram computations have been terminated.

In this prose description of the principal UML activity diagram concepts we have deliberately left out the so called object flow facility in UML diagrams. This feature allows objects to appear as nodes which are connected by dependencies to activities or to transitions which may create, modify or destroy objects. The ASM definition in the next section for the behavioral meaning of UML activity diagrams captures this object flow facility automatically through the underlying notion of ASM state transformation. A similar remark applies to communication between concurrent agents.

3 ASM Models for UML Activity Diagrams

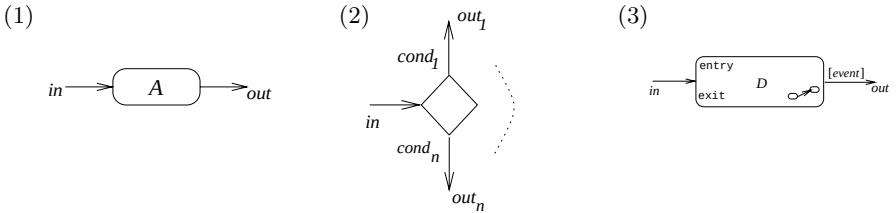
In this section we associate to each building block of the UML notation for activity diagrams an ASM which models the behavior of the UML construct. We first describe the node and subdiagram types occurring in UML activity diagrams and then assign execution rules to them.

3.1 Signature of ASM Activity Diagrams

The graphs of ASM activity diagrams are finite directed graphs whose nodes and arcs belong to abstract sets $NODE$ and ARC coming with auxiliary static functions which yield information on the components (nodes, arcs, subdiagrams).

$NODE$ is partitioned into atomic and composite nodes. Atomic nodes are partitioned into action and branching nodes, composite nodes into (sequential) activity and concurrent nodes.

Action nodes are of form $node(in, A, out, isDynamic, dynArgs, dynMult)$ (1) where the parameter in denotes the (unique) incoming arc, out the (unique) outgoing arc, A an atomic action (to be defined by a set of simultaneous ASM function updates), $isDynamic$ if A may be executed $dynMult$ times in parallel, each time with an argument L_i from a set $dynArgs$ of sequences of objects $\{L_1, \dots, L_n\}$.



Branching nodes are of form $node(in, Cond, Out)$ (2) where in denotes the incoming arc, $Cond$ a finite sequence of disjoint and exhaustive boolean expressions and Out a sequence (of same length as $Cond$) of outgoing arcs.

The *initial node* $\bullet$ and *final nodes* $\odot$ are considered as special action nodes without parameter.

For each type of node parameter we use a (static) function $param$ which applied to nodes yields the parameter. For example $in(node)$ yields the incoming arc of $node$, $action(node)$ the atomic action A associated to an action $node$, etc. We often suppress parameters notationally.

Activity nodes are of form $node(in, entry, exit, D, out, [event], isDynamic, dynArgs, dynMult)$ (3) where in denotes the incoming arc, out the outgoing arc, $entry$, $exit$ the entry/exit actions, D the associated activity (sub)diagram and $[event]$ an [optional] event which serves to determine interrupts (if any) of executions of D ; $isDynamic$, $dynArgs$ and $dynMult$ play the same role as for the action nodes. In accordance with [13] events are admitted to label the outgoing arc only of activity nodes and not of action nodes.

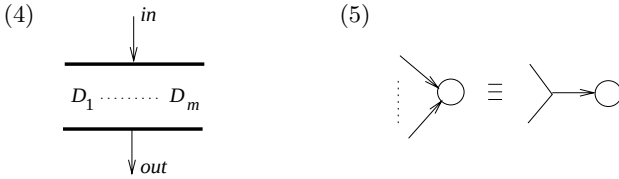
DIAGRAM is an abstract set of activity diagrams, *EVENT* an abstract set of diagram events. A monitored predicate *occurred* defined on *EVENT* indicates that and when an event happens. It is assumed that an occurring event is immediately taken into account and consumed.

Since the computation in an activity node subdiagram can be interrupted at any moment by every event which is associated to a transition coming out from an enclosing diagram, we introduce a static function

$$evQueue : DIAGRAM \rightarrow EVENT^*$$

which yields for each diagram D the sequence of all highest (outermost) occurrences, in diagrams in which D is nested, of all events whose firing can cause the exit of D . If an event e occurs in the *evQueue* of different not nested diagrams, then only that occurrence of e will have effect that is associated to the diagram where the control lies. Apparently it is assumed in UML that at every moment for each active diagram at most one event in *evQueue* can occur.

Concurrent nodes are of form $node(in, D, out)$ (4) where the parameter in denotes the incoming arc, out the outgoing arc, D a sequence $D_1, \dots, D_m$ of subdiagrams (not consisting of only initial or final nodes)¹. In (4) the upper synchronization bar is called *fork*, the lower one *join*.



In addition to the functions which yield node parameters we will use the following auxiliary functions on nodes, arcs and diagrams. The diagram to which an arc belongs is given by the function $diagram : ARC \rightarrow DIAGRAM$. The arc outgoing from the initial node is given by $initArc : DIAGRAM \rightarrow ARC$. The arcs incoming to a final node are provided by $finalArc : DIAGRAM \rightarrow ARC^*$. For an activity node subdiagram D the function $event : DIAGRAM \rightarrow EVENT \cup \{undef\}$ yields the occurrence of an event (if it exists) which labels the transition coming out from that activity node. The arc which is labeled by such an event is provided by the function $arc : EVENT \cup \{undef\} \rightarrow ARC$. $out(D)$ yields the outgoing arc of the activity node with subdiagram D . $exit(D)$ yields the exit parameter defined on the activity node with subdiagram D . If D is the main diagram, $exit(D)$ returns its exit action. The predicate $exited(D)$ is true if the exit action associated to the diagram D has been performed. $FinalNode(n, D)$ is true if n is a final node of D .

The subdiagram associated to an activity node whose outgoing arc is labeled by the occurrence of an event is provided by $diagram : EVENT \rightarrow DIAGRAM$.

In connection with labeled transitions, we use the function *guard*, defined on *ARC*, which yields the guard condition labeling an arc. On “not labeled” arcs this guard has the constant value *true* (and is therefore notationally suppressed).

¹ The initial (resp. final) node of the subdiagrams D_i is considered to be represented by the fork (resp. join) bar.

The nesting structure of an ASM activity diagram is encoded in the following static functions:

- $upDgm : DIAGRAM \longrightarrow DIAGRAM \cup \{undef\}$, assigning to a diagram D_1 its immediately enclosing diagram (if any).
- $chainDgm : DIAGRAM \times DIAGRAM \longrightarrow DIAGRAM^* \cup \{undef\}$

$$chainDgm(D_1, D_2) = [T_1, \dots, T_n] \quad \text{where } n > 1 \wedge \\ T_1 = D_1 \wedge T_n = D_2 \wedge \forall i = 1 \dots n - 1, \quad upDgm(T_i) = T_{i+1}$$

$chainDgm$ yields an empty sequence on each pair of diagrams D_1 and D_2 where D_1 is not nested in D_2 . We write $chainDgm(D_1, \widehat{D_2})$ to indicate the right open sequence $chainDgm(D_1, \widehat{D_2}) = [T_1, \dots, T_n[$, if it exists.

We denote by $\subset$ the transitive closure of $upDgm$, i.e.

$$D \subset D' \text{ iff } upDgm(D) = D' \vee \exists D'' (upDgm(D) = D'' \wedge D'' \subset D')$$

Definition 1 (ASM Activity Diagrams). *An activity diagram is defined recursively following its sequential depth, i.e. the maximum level of nesting of occurring activity nodes, and its concurrency depth, i.e. the maximum level of nesting of occurring concurrent nodes.*

- A sequential ASM activity diagram of sequential depth 0 is a finite directed graph containing only atomic nodes and exactly one initial node.
- A sequential ASM activity diagram of sequential depth $n + 1$ is a finite directed graph containing besides atomic nodes only activity nodes whose associated subdiagram is an ASM activity diagram of sequential depth at most n .
- Every sequential ASM activity diagram (i.e. of finite sequential depth) is an ASM activity diagram of concurrency depth 0.
- An ASM activity diagram of concurrency depth $n + 1$ is a finite directed graph D of atomic and composite nodes such that for each concurrent node which occurs in D or in a diagram of the D -subdiagram chain, its (immediate) subdiagrams are ASM activity diagrams of concurrency depth at most n .

Any ASM activity diagram of arbitrary concurrency depth is called an ASM activity diagram. ASM activity diagrams of positive concurrency depth are called concurrent ASM activity diagrams.

Remark 1. UML activity diagrams inherit through the metamodel in [14] all the features of state diagrams including *synchronization nodes* to synchronize concurrent subactivities. One transition connects the output of a fork in one flow to the input of the synch node, and another transition connects the output of the synch node to the input of a join in the other flow [12]. Synchronization nodes can be incorporated here by adapting the definitions given below for concurrent node synchronization.

3.2 Rules of ASM Activity Diagrams

In this subsection we define the ASM rules which express the semantics of the control flow of sequential and concurrent activities as transformation of the configuration of the graphs of ASM activity diagrams. We use *multi-agent ASMs* [8] to model the concurrent subactivities in a UML activity diagram.

ASMs are transition systems, their states are multi-sorted first-order structures, i.e. sets with relations and functions, where for technical convenience relations are considered as characteristic boolean-valued functions. The transition relation is specified by rules describing the modification of the functions from one state to the next, namely in the form of guarded updates (“rules”)

if *Condition* **then** *Updates*

where *Updates* is a set of function updates $f(t_1, \dots, t_n) := t$, which are simultaneously executed when *Condition* is true. A multi-agent ASM is given by a set of (sequential) agents, each executing a program consisting of ASM transition rules. Distributed runs of multi-agent ASMs are defined in [8].

Let *AGENT* be the abstract set of agents *a* which move through their associated diagram *diagram(active(a))* executing the activity required at the currently *active* arc of this diagram, i.e. the arc where the agent’s control lies. Thus

active : *AGENT* $\rightarrow$ *ARC*

is a dynamic function whose updates follow the control flow of the given activity diagram. The agents execute UML activity diagrams, i.e. they all use the ASM rules defined below. As a consequence we below use the 0-ary function *Self* which is interpreted by each agent *a* as *a*. We often notationally suppress *Self*, writing *active* for *active(Self)* and *currDgm* for *diagram(active(Self))*.

Agents can be activated and deactivated during the execution. The current mode of an agent is recorded by a dynamic function

mode : *AGENT* $\rightarrow$ {*running*, *waiting*, *suspended*, *undef*}.

An agent can be *running* in normal mode, *waiting* for the end of concurrent subcomputations of his children—in these two cases we say that it is active—, *suspended* because exited from a sequence of nested diagrams when its computation has been stopped due to some event firing, or deleted (of mode *undef*) when it has regularly finished its own computation.

We formalize the data flow of activity diagrams by an *env* function representing the state of each agent.

When an agent is created to perform a concurrent subcomputation (defined by one of the subdiagrams leaving from a *fork* synchronization bar of a concurrent node), he is linked to the *parent* agent by the dynamic function

parent : *AGENT* $\rightarrow$ *AGENT* $\cup$ {*undef*}.

We assume that this function yields *undef* for the main agent who is not part of any concurrent flow. The active subagents of an agent *a* are collected in the set *SubAgent(a)* = {*a'* $\in$ *AGENT* | *parent(a') = a*}.

When an activity has to be performed several times in parallel, we “clone” the current running agent as often as required by the activity multiplicity. In order to distinguish the cloned agents from the original one, we introduce the predicate *isClone* on *AGENT*.

For the initial state it is required that there is a unique agent, in *running* mode with *active* arc positioned on the initial arc of the associated diagram D_{Main} . In case the main diagram has an associated entry action we assume that this action has been executed as part of the initialization.²

Typically a running agent, to determine the next action, has to look at the target of its currently active arc and to check that the associated guard is true and that no interrupt occurs. This is expressed by the following condition which appears as guard of most of the rules.

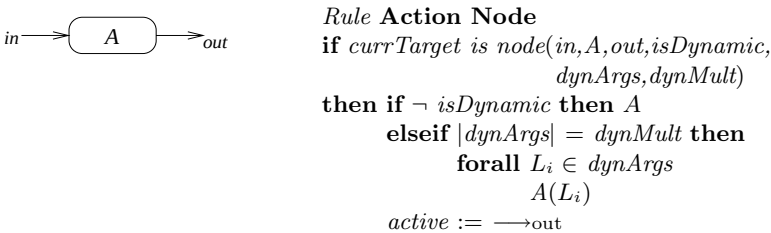
$$\begin{aligned} currTarget \text{ is node} &\equiv mode(Self) = running \wedge noEventFor(Self) \\ &\wedge active(Self) = in(node) \wedge guard(active(Self)) \end{aligned}$$

where $noEventFor(Self) \Leftrightarrow \forall e \in evQueue(currDgm) : \neg occurred(e)$.

The **semantics of atomic nodes** is straightforward. By interpreting the UML concept of “atomic action” by “ASM rule”, we provide a rigorous semantical meaning to this concept which turned out to be general enough for contemporary system design [2].

If the control is on the arc $in(node)$ and the associated guard is satisfied (in case of triggerless guarded actions or of transitions triggered by events which are classified as triggering internal transitions), then A is performed if the action node is not dynamic. Otherwise, if the number of the dynamic arguments $dynArgs$ corresponds to the dynamic multiplicity $dynMult$, A is executed (by $Self$) concurrently on each argument list. The semantics of ASMs takes care of the consistency problem for the possibly conflicting concurrent updates of env , a problem which is hardly considered in [14]. In both cases, the control passes to the arc outgoing from the node.

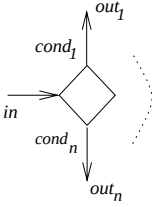
According to [14], if $dynArgs$ is evaluated to the empty set, then no action is performed. If the $dynArgs$ “evaluates to a set with fewer or more elements than the number allowed by the dynamic multiplicity ... the behavior is not defined” [14]. This is reflected by the following rule Action Node.



For action nodes without dynamic arguments standard ASMs guarantee A to be an atomic (indivisible) action. In the presence of dynamic arguments a concept of parameterized ASMs with atomic interpretation is needed as provided in [4] for structuring large machines.

² The initialization could also be described by a rule which fires only in the initial state to perform the entry action of the main diagram D_{Main} and to set the control of the unique running agent on the arc outgoing from the initial node.

If the control is on the arc incoming to a branching node, the associated guard is satisfied and no interrupting event does occur, then the conditions are evaluated and the control passes to the arc associated to the true condition.



Rule Branching Node

```

if currTarget is node(in, (condi)i≤n, (outi)i≤n)
then
    if cond1 then active :=  $\longrightarrow$ out1
    ⋮
    elseif condn then active :=  $\longrightarrow$ outn

```

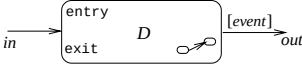
The way UML describes final nodes is ambiguous. One question is whether exit actions should be executed upon reaching final nodes (“normal” exit) or only when events occur which interrupt the subdiagram computation (“interrupt” exit). This is related to the question whether an arc is allowed to leave out of the subdiagram of an activity node(*in*, *entry*, *exit*, *D*, *out*, *event*) without passing through the *out*-arc of this node. The Normal Activity Exit rule defined below suggests one way of resolving this ambiguity, but others are possible.

Semantics of an activity node. When an activity node is entered, its activity subdiagram is executed (once or multiple times in parallel, depending on the node dynamics) starting with the entry action and following the semantics of activity diagrams. The activity node is not exited unless a final node of the subdiagram is reached (normal exit) or trigger events occur on a transition coming out from it (interrupt exit).

If the control is on the incoming arc of an activity node, the associated guard is satisfied and no interrupting event does occur, then the control passes to the initial arc of the nested diagram and the entry action³ is performed if the node is not dynamic. Otherwise, if the number of the dynamic arguments *dynArgs* corresponds to the dynamic multiplicity *dynMult*, *D* is executed in parallel on each argument list of the set *dynArgs*. The running agent *Self* is cloned *dynArgs* -1 many times. For each agent (a) the environment is extended by the corresponding dynamic list, (b) the control passes to the initial arc of the nested diagram and (c) the entry action is performed. (See the macro *activate* at the end of this section.) In [14] there is no mention what happens in case *dynArgs* is evaluated to the empty set. In our model no activity is performed and the agent waits on the incoming arc until a possible triggered event will put its control on the outgoing arc. If the value *dynArgs* differs from *dynMult* “the behavior is not defined” [14]. The rule Enter Activity does not fire in this case.

If the control is on a final arc of the subdiagram of an activity node, two cases may occur, depending on whether the arc *out* outgoing from the activity node is labeled by an event or not. In the second case, the exit action must be performed. At the final arc of the main diagram the agent gets mode *undef* to stop the computation; otherwise, the control passes to the outgoing arc. All cloned agents end here their (concurrent sub)activities. (See the macro *delete* at

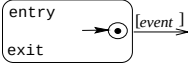
³ If not defined, *entry* stands for *skip*.

**Rule Enter Activity**

```

if currTarget is node(in, entry, exit, D, out, [event],
      isDynamic, dynArgs, dynMult)
then if  $\neg$  isDynamic
  then active  $:=$  initArc(D)
    entry
  else if  $|dynArgs| = dynMult$ 
    then let  $dynArgs = \{L_0, L_1, \dots, L_n\}$ 
      extend AGENT with  $a_1 \dots a_n$ 
        do forall  $1 \leq i \leq n$ 
          activate( $a_i, D, L_i$ )
          isClone( $a_i$ )  $:=$  true
        env  $:=$  env  $\cup$   $L_0$ 
        active  $:=$  initArc(D)
        entry

```

**Rule Normal Activity Exit**

```

if currTarget is node &
      FinalNode(node, currDgm) &
      event(currDgm) = undef
then
  exit(currDgm)
  if currDgm  $= D_{Main}$ 
    then mode(Self)  $:=$  undef
  else if  $\neg$  isClone(Self)
    then active  $:=$  out(currDgm)
    else delete(Self)

```

the end of this section.) In the first case exit is possible only upon firing of the event.

If an event *ev* occurs that is relevant for the current diagram, then first all exits from nested diagrams are performed, in the sequential order which is given by the subdiagram structure—this is achieved by iterating the ASM submachine *exitDiagrams* until its termination—and then the control passes to the arc labeled by *ev*. All cloned agents end here their own (concurrent sub)activities (rule **Event Handling**).

Rule Event Handling

```

if occurred(ev) & ev  $\in$  evQueue(currDgm)
then seq InitializeExitDiagrams
  Iterate(exitDiagrams(ev))
  if  $\neg$  isClone(Self)
    then active  $:=$  arc(ev)
    else delete(Self)

```

To perform the exit from nested diagrams in the order of the subdiagram structure one has to distinguish the following cases:

1. The agent is inside a concurrent flow (i.e. $\text{parent}(\text{Self}) \neq \text{undef}$). The sequential execution order from children to parents for exiting subdiagrams is reflected by the check that all subagents have performed all their exit actions and that the agent's current diagram is the next to be exited. If the event ev is significant only for this agent (i.e. it has no effect on the sibling agents participating in the concurrent computation), then all the exit actions from the current diagram to the diagram $D = \text{diagram}(ev)$ associated to the event ev are performed. Otherwise, if the event has effect on all the sibling agents running in parallel, then all of them independently first perform all the exit actions from their own current diagram up to but excluding their common ancestor diagram $D' = \text{diagram}(\text{active}(\text{parent}(\text{Self})))$, and then set their mode to *suspended* whereby they stop their computation.
2. The agent is not inside a concurrent flow (i.e. $\text{parent}(\text{Self}) = \text{undef}$). If the agent is not parent of concurrent subagents and its current diagram is the next to be exited, then it performs in the correct order all the exit actions from its current diagram to the diagram $D = \text{diagram}(ev)$ associated to ev . Otherwise, if the agent is parent of concurrent subagents, then it checks whether all subagents have concluded the exit actions of the diagrams under their control. In this case, the parent agent sequentially (a) performs in sequence all the exit actions from its current diagram to the diagram $D = \text{diagram}(ev)$, and (b) gets mode *running*.

To make this work we initialize the exiting submachine *exitDiagrams* by:

InitializeExitDiagrams $\equiv \forall D : \text{exited}(D) := \text{false}$.

The exiting submachine itself is defined as follows:

```

exitDiagrams( $ev$ )  $\equiv$  if  $\text{parent}(\text{Self}) \neq \text{undef} \ \&$ 
     $\forall a \in \text{SubAgent}(\text{Self}) : \text{mode}(a) = \text{suspended} \ \&$ 
     $\text{exited}(\text{currDgm}) = \text{false}$ 
then if  $\text{upDgm}(\text{diagram}(ev)) \subset D'$ 
    then  $\text{exitFromTo}(\text{currDgm}, D)$ 
    if  $\text{upDgm}(\text{diagram}(ev)) \supseteq D'$ 
    then seq  $\text{exitFromTo}(\text{currDgm}, \widehat{D'})$ 
     $\text{deactivate}(\text{Self})$ 
if  $\text{parent}(\text{Self}) = \text{undef} \ \& \ \text{exited}(\text{currDgm}) = \text{false}$ 
then if  $\text{mode}(\text{Self}) = \text{running}$ 
    then  $\text{exitFromTo}(\text{currDgm}, D)$ 
    if  $\text{mode}(\text{Self}) = \text{waiting} \ \&$ 
     $\forall a \in \text{SubAgent}(\text{Self}) : \text{mode}(a) = \text{suspended}$ 
    then seq  $\text{exitFromTo}(\text{currDgm}, D)$ 
     $\text{mode}(\text{Self}) := \text{running}$ 
where  $D = \text{diagram}(ev)$ 
     $D' = \text{diagram}(\text{active}(\text{parent}(\text{Self})))$ 

```

The macro $\text{exitFromTo}(D_1, D_2)$ performs sequentially the exit actions from the diagram D_1 to the diagram D_2 , following in this loop the subdiagram structure:

```

 $\text{exitFromTo}(D_1, D_2) \equiv$  loop through  $T \in \text{chainDgm}(D_1, D_2)$ 
     $\text{exit}(T)$ 
     $\text{exited}(T) := \text{true}$ 

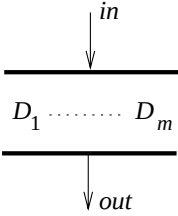
```


The macro $deactivate(a) \equiv mode(a) := suspended$ stops an agent's (sub)computation temporarily or permanently.

The definition of sequential composition and of iteration of ASMs provided in [4] guarantees that the machine *exitDiagrams* starts in the state in which an event triggers the application of the Event Handling rule and terminates its execution before the next event may take effect, thus preventing nesting of calls of the *exitDiagrams* machine.

Semantics of concurrent nodes. When the control is on an arc entering a fork synchronization bar, then the flow is split into two or more flows of control. The currently running agent creates the necessary new agents setting their mode to *running*. Each new subagent a_i inherits the program for executing activity diagrams, its control is started on the first arc of its current diagram D_i and its entry action (if present) is performed (see the macro *activate* below). The parent agent gets mode waiting and his control is moved to the arc outgoing from the join synchronization bar (see rule **Fork** below).

When all the subagents running in a parallel flow reach their join synchronization bar, their parallel flows must be joined. The parent agent, who is positioned in waiting mode on the arc outgoing from the join bar, checks whether all his subagents have already finished their work. If yes, the parent agent gets mode running and deletes all the subagents (see rule **Join** below).



Rule Fork

if $currTarget$ is node($in, D_1, \dots, D_m, out$)
then

extend AGENT with $a_1 \dots a_m$
 do forall $1 \leq i \leq m$
 $activate(a_i, D_i)$
 $parent(a_i) := Self$
 $mode(Self) := waiting$
 $active := out$

Rule Join

if $active = out(node(in, D_1, \dots, D_m, out)) \&$
 $mode(Self) = waiting \&$
 $\forall a_i \in SubAgent(Self) :$
 $mode(a_i) = running \&$
 $active(a_i) = finalArc(D_i)$

then

do forall $a_i \in SubAgent(Self)$
 $delete(a_i)$
 $mode(Self) := running$

$\begin{aligned} \text{activate}(a, D, [S]) &\equiv \text{mode}(a) := \text{running} \\ &\quad \text{Mod}(a) := \text{Mod}(\text{Self}) \\ &\quad \text{env}(a) := \text{env}(\text{Self}) \cup S \\ &\quad \text{active}(a) := \text{initArc}(D) \\ &\quad \text{entry}(D) \end{aligned}$	$\begin{aligned} \text{delete}(a) &\equiv \text{mode}(a) := \text{undef} \\ &\quad \text{Mod}(a) := \text{undef} \\ &\quad \text{parent}(a) := \text{undef} \\ &\quad \text{isClone}(a) := \text{false} \\ &\quad \text{env}(a) := \text{undef} \\ &\quad \text{active}(a) := \text{undef} \end{aligned}$

4 Discussion

The activity diagram metamodel in [14] is a class diagram and thus provides only a structural, static definition without describing the control-flow semantics. Essentially it comes as a particular case of the state diagram metamodel although some state diagram features are meaningless for activities. For example *stub states* can never occur within an activity diagram due to the requirement [14] that an activity is always entered through its initial node and exited through a final node or by triggering of an event.

Some features which appear in [14] but are not mentioned here are automatically covered by our definition. Namely *call states* are special *action states* and therefore require no separate formalization, similarly for synchronization nodes, which are variants of concurrent nodes, and for actions or activities which label transitions reflected by action and activity nodes targeted by the transition. The UML *object and communication flow states* are automatically captured in our model by the underlying notion of ASM state.

We have resolved the ambiguity between admitting triggerless transitions within activity diagrams (as in [1]) and the requirement that a transition guard must be evaluated only after triggering the transition event, namely by assuming transitions to be optionally labeled by events and by interpreting empty transition guards as true. We have completed the definition of actions or activities where the list of *dynArgs* is empty. Our model provides a clear semantics of the object and communication flow within parallel and concurrent activities, although we did not commit to any specific reporting mechanism of subagents to their parents since [14] leaves this question open. On the basis of the ASM composition principles defined in [4] our model also resolves the issue of atomicity of exit actions, a problem which seems not to be addressed in [1,14].

5 Activity Diagrams for the Semantics of Occam

In this section we apply the ASM semantics of UML activity diagrams to give a rigorous but succinct formulation of the semantics of the programming language Occam [3], where the combination of rigor and succinctness is achieved by associating to the graphical UML notation a semantical ASM definition. The description starts from the usual graphical layout of a program as flowchart—we define below such a graph generation from programs—whereby the standard sequential and syntax directed part of control is already incorporated into the

underlying graph structure (flattening the structured program) and thus semantically defined by the ASM rules for activity diagrams with atomic and concurrent nodes. It only remains to define the semantical meaning of the atomic actions—basic Occam instructions—which are executed by the agents when passing through the action nodes of the diagram. Figure 1 thus provides the complete definition of the semantics of Occam, where the atomic actions and the guards are specific for Occam and defined as follows. We use the function e to express the environment (i.e. the association of values to local variables) of each agent a and the function $eval$ for expression evaluation under a given environment. $s[v/t]$ denotes the result of substituting v in s by t (see Appendix for flowcharts generation from OCCAM programs).

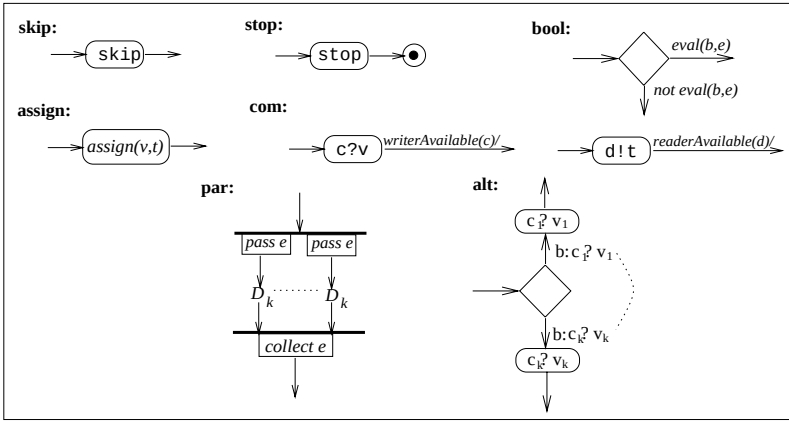


Fig. 1. ASM Activity Diagrams for the Semantics of Occam

$$\begin{array}{ll}
 \text{writerAvailable}(c) \equiv & \text{readerAvailable}(d) \equiv \\
 \exists \text{ writer} \in \text{AGENT} \ \exists n \in \text{NODE}: & \exists \text{ reader} \in \text{AGENT} \ \exists n \in \text{NODE}: \\
 \text{active}(\text{writer}) = \text{in}(n) \ \& & \text{active}(\text{reader}) = \text{in}(n) \ \& \\
 \text{action}(n) = d!t \ \& & \text{action}(n) = c?v \ \& \\
 \text{eval}(c, e(a)) = \text{eval}(d, e(\text{writer})) & \text{eval}(d, e(a)) = \text{eval}(c, e(\text{reader}))
 \end{array}$$

$$\begin{array}{ll}
 \text{assign}(v, t) \equiv e := e[v/\text{eval}(t, e)] & c?v \equiv e(a) := e(a)[v/\text{eval}(t, e(\text{writer\_for\_c}))] \\
 \text{pass } e \equiv e(a) := e(\text{parent}(a)) & d!t \equiv \text{skip} \\
 \text{collect } e \equiv e(\text{Self}) := \cup_{1 \leq i \leq k} e(a_i) & b:c?v \equiv \text{writerAvailable}(c) \ \& \ \text{eval}(b, e(a))
 \end{array}$$

In the Fork rule we add $\text{pass } e$ to the updates of $\text{activate}(a, D)$, in the Join rule $\text{collect } e$ to the updates of Self . Occam syntax conditions guarantee that our definition of reader and writer for a channel is consistent, since for each channel at each moment at most one pair of reader/writer exists which are available for each other.

Remark 2. The description in Figure 1 uses no composite activity nodes because we have assumed that in the initially given flowchart the structured

parts of the underlying Occam program has been flattened. One can use activity nodes to describe such structural pieces of code directly, without flattening.

Remark 3. The description of the ALT construct in Figure 1 really provides the deterministic Transputer implementation of this construct, due to the underlying UML assumption that the guards of branching nodes have to be evaluated in sequential order and to be disjoint and complete. One can reflect the non deterministic Occam interpretation of this construct by interpreting the guards g_i as driven by monitored choice functions which directly reflect the non determinism, but outside the run.

From the above definition of the semantics of Occam one can derive its implementation to Transputer code by a series of stepwise refinements of ASMs, see [3] where the correctness of such a transformation has been proved.

6 Conclusion

We have used ASMs to a transparent yet rigorous definition of the semantics of UML activity diagrams. Where the UML texts do not provide enough information to decide between different possible meanings, we have presented a formalization which seems to be in the main stream of the ideas surrounding UML. Exploiting the abstract nature of ASMs it is easy to adapt these definitions to changing requirements. We have applied Activity Diagram machines to provide a rigorous succinct definition of the semantics of OCCAM.

We intend to use our notion of Activity Diagram Machines as a preparatory step for an ASM definition of UML state machines, declared in [10,9] to provide the official meaning of their implementation in Statemate.

References

1. G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.
2. E. Börger, *High Level System Design and Analysis using Abstract State Machines*; in D. Hutter, W. Stephan, P. Traverso, M. Ullmann (eds): *Current Trends in Applied Formal Methods (FM-Trends 98)*, Springer LNCS 1641, pp. 1–43, 1999.
3. Börger E., Durdanovic I., *Correctness of Compiling Occam to Transputer Code*; in: *The Computer Journal*, Vol. 39, No.1, pp.52-92, 1996.
4. E. Börger, J. Schmid, *Composition and Submachine Concepts for Sequential ASMs*. Gurevich Festschrift, Proc. CSL'2000 (to appear).
5. A. Evans, J-M. Bruel, R. France, K. Lano, B. Rumpe, *Making UML Precise*, OOPSLA'98 Workshop on "Formalizing UML. Why and How?" October 1998.
6. A. Evans, R. France, K. Lano, B. Rumpe, *The UML as a formal modeling notation*, UML98 - Beyond the notation, Springer LNCS, 1998.
7. R.B. France, A.S. Evans, K.C. Lano, and B. Rumpe, *Developing the UML as a formal modeling notation*; in *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, 1998.
8. Y. Gurevich, *Evolving Algebras 1993: Lipari Guide*; in E. Börger (Ed.): *Specification and Validation Methods*, Oxford University Press, 1995.

9. D. Harel, A. Naamad, *The STATEMATE Semantics of Statecharts*, ACM Trans.Soft.Eng. method 5(4), 1996, 293-333.
10. D. Harel, M. Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, 1998.
11. Rational Software Corporation, Unified Modeling Language (UML), version 1.3, <http://www.rational.com>, 1999.
12. J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.
13. *UML Notation Guide*, 1999. (Published as part of [11]).
14. *UML 1.3 Semantics*, 1999. (Published as part of [11]).

7 Appendix

We here give a recursive definition for the generation of flowcharts from OCCAM programs. In this definition we make use of the following auxiliary functions:

$new(Domain)$ yields a fresh element from $Domain$

$makeLink(node, node')$ yields $node \longrightarrow node'$ $connect(\longrightarrow, node)$ yields $\longrightarrow node$
 $makeLink(node, label)$ yields $node \xrightarrow{label}$ $connect(node, \longrightarrow)$ yields $node \longrightarrow$
 $makeLink(label, node)$ yields $\xrightarrow{label} node$ $connect(node, \longrightarrow, g)$ yields $node \xrightarrow{g}$

Definition 2. We define the function $mkGraph(P, in, out) = in \longrightarrow \boxed{P} \longrightarrow out$ by recursion on the structure of the program P .

Basis (Atomic program statements):

- $mkGraph(atomic, in, out) = in \longrightarrow \boxed{atomic} \longrightarrow out$ $atomic \in \{assign(v, t), skip, stop\}$
 - $mkGraph(c?v, in, out) = in \longrightarrow \boxed{c?v} \xrightarrow{readerAvailable(c)/} out$
 - $mkGraph(d!t, in, out) = in \longrightarrow \boxed{d!t} \xrightarrow{writerAvailable(d)/} out$

Induction Step

- 1- $mkGraph(P_1 \text{ SEQ } P_2, in, out) =$
 $mkGraph(P_1, in, mid)$
 $mkGraph(P_2, mid, out)$
 where $mid = new(LINK)$
- 2- $mkGraph(IF b THEN P ELSE Q, in, out) =$
 $let \beta = new(BRANCHNODE)$
 $yes = makeLink(\beta, b)$
 $no = makeLink(\neg\beta, b)$
 $connect(in, \beta)$
 $mkGraph(P, yes, out)$
 $mkGraph(Q, no, out)$
- 3- $mkGraph(WHILE b DO P, in, out) =$
 $let \beta = new(BRANCHNODE)$
 $yes = makeLink(\beta, b)$
 $return = makeLink(' ', b)$
 $connect(in, \beta)$
 $connect(\beta, out, \neg b)$
 $mkGraph(P, yes, return)$
- 4- $mkGraph(PAR(P_1, \dots, P_k), in, out) =$
 $let \alpha = new(FORKNODE)$
 $\alpha_i = makeLink(\alpha, ' ', i = 1, \dots, k)$
 $\beta = new(JOINNODE)$
 $\beta_i = new(ARC)$
 $connect(in, \alpha)$
 $mkGraph(P_i, \alpha_i, \beta_i)$
 $connect(\beta_i, \beta)$
 $connect(\beta, out)$

```

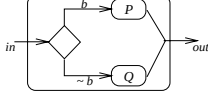
5- mkGraph(ALT( $b_1 : c_1?v_1, \dots, b_k : c_k?v_k$ ), in,  $out_1, \dots, out_k$ ) =
  let  $\beta$  = new(BRANCHNODE)
     $l_i$  = makeLink( $\beta, b_i : c_i?v_i$ ),  $i = 1, \dots, k$ 
  connect(in,  $\beta$ )
  mkGraph( $b_i : c_i?v_i, l_i, out_i$ ),  $i = 1, \dots, k$ 

```

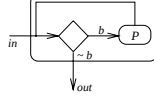
1-



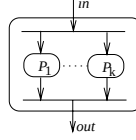
2-



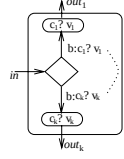
3-



4-



5-



Approximate Bisimilarity

Mingsheng Ying^{1*} and Martin Wirsing²

¹ State Key Laboratory of Intelligent Technology and Systems,
Department of Computer Science and Technology,
Tsinghua University, Beijing 100084, China
`yingmsh@mail.tsinghua.edu.cn`

² Institut für Informatik,
Ludwig-Maximilians-Universität München,
Oettingenstraße 67, 80538 München, Germany
`wirsing@informatik.uni-muenchen.de`

Abstract We introduce a notion of approximate bisimilarity in order to be able to reason about the approximate equivalence of processes. Approximate bisimilarity is based on the notion of bisimulation index for labelled transition systems. We establish some basic properties of bisimulation indexes and give a Hennessy-Milner logical characterization of approximate bisimilarity. As an application we show how to describe approximate correctness of real time systems by giving an example in real time ACP.

Keywords: Process Algebra, Bisimulation, Real Time ACP.

1 Introduction

One of the central notions in process calculus is bisimulation [7, 8]. Roughly speaking, bisimulation expresses the equivalence of processes whose external actions are identical. Requiring exactly the same actions is quite rigorous. There are many situations and applications where two processes fail to satisfy this requirement, but the two processes are still more or less bisimilar in the following sense: whenever a process makes an action the other one can make an action different from but very similar to the action the first process made. In order to cope with such situations we propose a concept of approximate bisimilarity in this paper which is especially appropriate for applications where actions are equipped with a non-discrete metric as it is the case, e.g., for real time systems.

To define approximate bisimilarity, we introduce the concept of bisimulation indexes in this paper. We presume a metric on actions which describes a certain

* The first author was supported by the National Foundation for Distinguished Young Scholars (Grant No: 69725004), Research and Development Project of High-Technology (Grant No: 863-306-ZT06-04-3) and Foundation of Natural Sciences (Grant No: 69823001) of China and Fok Ying-Tung Education Foundation, and this work was carried out when the first author was at Institut für Informatik, Ludwig-Maximilians-Universität München as an Alexander von Humboldt research fellow.

distance (and on the opposite side, similarity) between actions. Bisimulation indexes are induced from this metric. Based on the concept of bisimulation indexes, we are able to introduce a notion of approximate bisimilarity, namely, lambda-bisimilarity. The notion of lambda-bisimilarity provides us with a continuous spectrum of equivalences which allow us to associate a degree of bisimilarity to pairs of processes and therefore to analyze the similarity of processes. Moreover, λ -bisimilarity is very suitable to describe approximate correctness of processes. We show this by generalizing some of the classical properties of bisimulation and by an example in real time ACP. In particular, we give a Hennessy-Milner logical characterization of approximate bisimilarity. As an application we study the behaviours of clocks and show that in contrast to classical bisimilarity approximate bisimilarity allows us to relate precise clocks with unreliable clocks.

This paper is organized as follows: in Sect. 2, we introduce the concept of bisimulation indexes in labelled transition systems and give some of their basic properties. In Sect. 3 we establish a Hennessy-Milner logical characterization of bisimulation indexes. To illustrate how bisimulation indexes can be applied to describe certain approximate behaviours of real time systems, a simple example in ACP is presented in Sect. 4. Sect. 5 is the concluding section in which we point out some related works and some problems for further works.

2 Bisimulation Indexes in Transition Systems

Recall that a metric space is a pair (X, ρ) in which X is a non-empty set and ρ is a mapping from $X \times X$ into $[0, \infty]$ such that (1) $\rho(x, y) = 0$ iff $x = y$, (2) $\rho(x, y) = \rho(y, x)$, and (3) $\rho(x, z) \leq \rho(x, y) + \rho(y, z)$ for any $x, y, z \in X$. If (1) is weakened by (1)' $\rho(x, x) = 0$ for each $x \in X$, then ρ is called a pseudo-metric; and if (3) is strengthened by (3)' $\rho(x, z) \leq \max\{\rho(x, y), \rho(y, z)\}$ for any $x, y, z \in X$, then ρ is called an ultra-metric or it is said to be non-Archimedean.

Definition 1. Let $\sigma = (S, T, \{\xrightarrow{t} : t \in T\})$ be a labelled transition system, ρ a metric on T and $R \subseteq S \times S$. We write

$$b_R(s, r'; t) = \inf\{\rho(t, u) : u \in T \wedge \exists s' \in S. s \xrightarrow{u} s' \wedge r' R s'\} \quad \text{and}$$

$$b_R(r, s) = \sup\{b_R(s, r'; t) : r' \in S \wedge t \in T \wedge r \xrightarrow{t} r'\}$$

for any $r, r', s \in S$ and $t \in T$ (where $\sup \emptyset, \inf \emptyset$ are defined to be 0 and ∞ , respectively). Then

$$b_R = \sup\{\max\{b_R(s_1, s_2), b_{R^{-1}}(s_2, s_1)\} : s_1 R s_2\}$$

is called the bisimulation index of R . Sometimes, we write b_R^σ for b_R to indicate the labelled transition system σ which we are considering.

The bisimulation index is a measure of the similarity of a relation R with bisimulation. Informally, if two processes r, s are in relation R (i.e., $r R s$ holds) and $r \xrightarrow{t} r'$, then $b_R(s, r'; t)$ denotes the infimum of distances between

transitions t and other transitions u , where u is a transition which can be used to complete the diagram of $s \xrightarrow{u} s'$ and $r' R s'$. Then $b_R(r, s)$ is the supremum of all such distances of transitions starting at r or s . The bisimulation index measures the distance to bisimulation for the whole relation R . From the definition of b_R , it is not difficult to see that the above definition is a numerical counterpart of the standard definition of bisimulations [8, Def. 4.1]. We should point out that the smaller the value of b_R is, the better R corresponds to a bisimulation, that is, in R an action performed by an agent will be simulated by a closer action performed by the other agent.

The bisimulation index has the following useful properties.

Proposition 1. 1. If R is a bisimulation, then $b_R = 0$. Especially, $b_{\text{Id}_S} = 0$.

2. $b_{R^{-1}} = b_R$.

3. $b_{R_1 \circ R_2} \leq b_{R_1} + b_{R_2}$.

Moreover, if ρ is an ultra-metric, then $b_{R_1 \circ R_2} \leq \max\{b_{R_1}, b_{R_2}\}$.

4. $b_{\bigcup_{i \in I} R_i} \leq \sup_{i \in I} b_{R_i}$.

The above proposition is a numerical version of Prop. 4.1 in [8]. It shows in particular that the supremum of individual bisimulation indexes is an upper limit of the bisimulation index of a union of relations. Moreover, the bisimulation index of a bisimulation is 0. The following example shows that the converse does not hold.

Example 1. Let T be the real line with the usual (Euclidean) metric, $S = \{x, y\}$, $x \xrightarrow{t} x$ for each rational number t , $y \xrightarrow{t'} y$ for each irrational number t' and $R = \{(x, y)\}$. Then we know that $b_R = 0$ but R is not a bisimulation. Thus the converse of Prop. 1(1) does not hold in general.

To give a partial converse of Prop. 1(1), let us introduce the notion of δ -closedness. Let $R \subseteq S \times S$ and $x \in S$. Then $R[x] = \{y \in S : x R y\}$. For any $x, y \in S$, $T(x, y) = \{t \in T : x \xrightarrow{t} y\}$; and for any $Y \subseteq S$, $T(x, Y) = \bigcup_{y \in Y} T(x, y)$. If $x, y \in S$ and there are $z \in S$ and $t \in T$ such that $z R x$ and $z \xrightarrow{t} y$, then $\langle x, y \rangle$ is called an R -push; if for any R -push $\langle x, y \rangle$, $T(x, R[y])$ is closed (with respect to ρ), then R is said to be *lower* δ -closed; and if both R and R^{-1} are lower δ -closed, then R is δ -closed. For example, $R = \{(x, y)\}$ in Ex. 1 is not lower δ -closed and thus R is not δ -closed.

Proposition 2. If R is δ -closed and $b_R = 0$, then R is a bisimulation.

A λ -bisimilarity is a relation whose bisimulation index is smaller than or equal to λ .

Definition 2. Let $\sigma = (S, T, \{\xrightarrow{t} : t \in T\})$ be a labelled transition system, $R \subseteq S \times S$ and $\lambda \in [0, \infty]$. If $b_R \leq \lambda$, then R is called a λ -bisimulation.

Clearly, any R is an ∞ -bisimulation; if $\lambda_1 \leq \lambda_2$ and R is a λ_1 -bisimulation, then R is also a λ_2 -bisimulation; and if R is a λ_i -bisimulation ($i \in I$), then R is an $(\inf_{i \in I} \lambda_i)$ -bisimulation. As a simple corollary of Prop. 1, we have

- Corollary 1.** 1. If R is a bisimulation, then R is a 0-bisimulation.
 2. R is a λ -bisimulation iff so is R^{-1} .
 3. If R_i is a λ_i -bisimulation ($i = 1, 2$), then $R_1 \circ R_2$ is a $(\lambda_1 + \lambda_2)$ -bisimulation. Moreover, if ρ is an ultra-metric, and R_1, R_2 are all λ -bisimulations, so is $R_1 \circ R_2$.
 4. If R_i is a λ -bisimulation ($i \in I$), so is $\bigcup_{i \in I} R_i$.

Definition 3. For any labelled transition system σ and any $\lambda \in [0, \infty]$, we define λ -bisimilarity $\sim_\lambda^\sigma$ as

$$\sim_\lambda^\sigma = \bigcup \{ \lambda\text{-bisimulation} \}.$$

For simplicity, $\sim_\lambda^\sigma$ is often abbreviated to $\sim_\lambda$ if σ is recognizable from the context.

The concept of bisimilarities with parameters λ supplies us with a continuous spectrum of relations which are similar to bisimulation at different degrees.

- Proposition 3.** 1. $\sim \subseteq \sim_0$ and if ρ is discrete, i.e., for some $\lambda > 0$, $\rho(t, u) \geq \lambda$ for any $t, u \in T$ with $t \neq u$, then $\sim = \sim_0$. $\sim_\infty = R \times R$. If $\lambda_1 \leq \lambda_2$, then $\sim_{\lambda_1} \subseteq \sim_{\lambda_2}$.
 2. For any $\lambda \in [0, \infty]$, $\sim_\lambda$ is a λ -bisimulation and it is reflexive and symmetric. $\sim_{\lambda_1} \circ \sim_{\lambda_2} \subseteq \sim_{\lambda_1 + \lambda_2}$. $\sim_0$ is an equivalence relation.
 If ρ is an ultra-metric, then $\sim_\lambda$ is an equivalence relation.

This proposition is immediate from Cor. 1 and Def. 3. We call $\sim_\lambda$ approximate bisimilarity of degree λ . Then $\sim_0$ is the best approximate bisimilarity which in many (non-discrete) applications may be more appropriate than classical bisimilarity. Note that Prop. 3 shows in particular the converse of Prop. 1 for discrete metrics.

Proposition 4. $s_1 \sim_\lambda s_2$ iff $b_{\sim_\lambda}(s_1, s_2) \leq \lambda$ and $b_{\sim_\lambda}(s_2, s_1) \leq \lambda$.

This proposition provides a recursive characterization of λ -bisimilarity $\sim_\lambda$ in terms of bisimulation indexes, but it appears in a way which is not as plain as Prop. 4.4 in [8]. To conclude this section we now propose the definition of bisimulation index up to $\sim_\lambda$. This notion induces a proof technique which is very useful for establishing bisimilarities with parameters.

Definition 4. 1. If $b_R(s, r'; t)$ in Def. 1 is replaced by

$$b_{R, \lambda}(s, r'; t) = \inf \{ \rho(t, u) : u \in T \wedge \exists s' \in S. s \xrightarrow{u} s' \wedge r' \sim_\lambda \circ R \circ \sim_\lambda s' \}$$

then we may define $b_{R, \lambda}$ in the same way as b_R .

$b_{R, \lambda}$ is called the bisimulation index of R up to $\sim_\lambda$.

2. If $b_{R, \lambda} \leq \mu$, then R is called a μ -bisimulation up to $\sim_\lambda$.

Proposition 5. 1. $b_{\sim_0 \circ R \circ \sim_0} \leq b_{R, 0}$; and

2. If ρ is an ultra-metric, then $b_{\sim_\lambda \circ R \circ \sim_\lambda} \leq \max\{\lambda, b_{R, \lambda}\}$.

Proposition 6. 1. If R is a μ -bisimulation up to $\sim_0$, then $R \subseteq \sim_\mu$.
 2. If ρ is an ultra-metric and R is a μ -bisimulation up to $\sim_\lambda$,
 then $R \subseteq \sim_{\max\{\lambda, \mu\}}$.

This proposition is an immediate corollary of Prop. 5.

3 Hennessy-Milner Logical Characterization of Bisimulation Indexes

In this section, we aim at describing bisimulation indexes in the style of Hennessy-Milner logic. So, this section is actually a generalization of Sect. 10.2 and 10.3 in [8]. To this end, we first establish a stratification of λ -bisimilarity which is analogous to Milner's work as in [8, Sect. 10.2] for bisimilarity.

Definition 5. Let $\sigma = (S, T, \{\overset{t}{\rightarrow} : t \in T\})$ be a transition system and $\lambda \in [0, \infty)$. Then the functional $\sharp_\sigma^\lambda : 2^{S \times S} \rightarrow 2^{S \times S}$ ($\sharp^\lambda$, for short) is defined as follows: for any $R \subseteq S \times S$, $\sharp^\lambda(R)$ is the set of all (s_1, s_2) such that for any $\theta > \lambda$ and for any $\tilde{t} \in T^* = \bigcup_{n=0}^\infty T^n$,

- i. whenever $s_1 \xrightarrow{\tilde{t}} s'_1$ then for some $\tilde{u} \in T$ and $s'_2 \in S$, $s_2 \xrightarrow{\tilde{u}} s'_2$, $s'_1 R s'_2$ and $\rho(\tilde{t}, \tilde{u}) < \theta$; and
- ii. whenever $s_2 \xrightarrow{\tilde{u}} s'_2$ then for some $\tilde{t} \in T$ and $s'_1 \in S$, $s_1 \xrightarrow{\tilde{t}} s'_1$, $s'_1 R s'_2$ and $\rho(\tilde{t}, \tilde{u}) < \theta$;

where for any $\tilde{t} = t_1 \dots t_m$ and $\tilde{u} = u_1 \dots u_n$ in T^* , we define

$$\rho(\tilde{t}, \tilde{u}) = \begin{cases} \max_{i=1}^m \rho(t_i, u_i), & m = n \\ \infty, & \text{otherwise} \end{cases}$$

Proposition 7. 1. $\sharp^\lambda$ is monotonic.
 2. R is a λ -bisimulation iff $R \subseteq \sharp^\lambda(R)$.
 3. $\sim_\lambda$ is the greatest fix-point of $\sharp^\lambda$.

Proof. (1) Clear from the definition of $\sharp^\lambda$.

(2) R is a λ -bisimulation iff $b_R \leq \lambda$ iff for any $(s_1, s_2) \in R$, $b_R(s_1, s_2) \leq \lambda$ and $b_{R^{-1}}(s_2, s_1) \leq \lambda$. By a simple calculation it is easy to show that $b_R(s_1, s_2)$ and $b_{R^{-1}}(s_2, s_1) \leq \lambda$ iff $(s_1, s_2) \in \sharp^\lambda(R)$.

(3) With Cor. 1(3) we know that $\sim_\lambda$ is the greatest λ -bisimulation. Hence, it holds that $\sim_\lambda \subseteq \sharp^\lambda(\sim_\lambda)$ and $\sharp^\lambda(\sim_\lambda) \subseteq \sharp^\lambda(\sharp^\lambda(\sim_\lambda))$ from (1) and (2). Furthermore, $\sharp^\lambda(\sim_\lambda)$ is also a λ -bisimulation, $\sharp^\lambda(\sim_\lambda) \subseteq \sim_\lambda$, and $\sim_\lambda$ is a fix-point of $\sharp^\lambda$. Since any fix-point of $\sharp^\lambda$ must be a λ -bisimulation, it is included in $\sim_\lambda$ and $\sim_\lambda$ is the greatest fix-point of $\sharp^\lambda$.

Definition 6. Let $\lambda \in [0, \infty)$. Then $\sim_\lambda^\kappa$ ($\kappa \in \text{On}$, the class of ordinals) are defined inductively as follows:

- i. $\sim_\lambda^0 = S \times S$;
- ii. $\sim_\lambda^{\kappa+1} = \sharp_\sigma^\lambda(\sim_\lambda^\kappa)$; and
- iii. $\sim_\lambda^\kappa = \bigcap_{\mu < \kappa} \sim_\mu^\kappa$ if $\kappa \in \text{On}_{\text{II}}$, the class of limit ordinals.

Proposition 8. 1. $\mu \leq \delta$ implies $\sim_\delta \subseteq \sim_\mu$.
 2. $\sim_\lambda = \bigcap_{\kappa \in \text{On}} \sim_\lambda^\kappa$.

Proof. A standard argument concerning largest fixed points.

Now, we introduce a modal logical language similar to that in [8, Sect. 10.3].

Definition 7. Let $\sigma = (S, T, \{\overset{t}{\rightarrow} : t \in T\})$ be a transition system and $\lambda \in [0, \infty)$. Then the modal logical language $\mathcal{L}_{\sigma, \lambda}$ is the smallest class of symbol strings satisfying the following conditions:

- i. if $F \in \mathcal{L}_{\sigma, \lambda}$, $\tilde{t} \in T^*$ and $\theta > \lambda$ then $\langle \tilde{t}, \theta \rangle F \in \mathcal{L}_{\sigma, \lambda}$;
- ii. if $F \in \mathcal{L}_{\sigma, \lambda}$ then $\neg F \in \mathcal{L}_{\sigma, \lambda}$; and
- iii. if $F_i \in \mathcal{L}_{\sigma, \lambda}$ for every $i \in I$ then $\bigwedge_{i \in I} F_i \in \mathcal{L}_{\sigma, \lambda}$. In particular, true $\stackrel{\text{def}}{=} \bigwedge_{i \in I} F_i$.

It should be noted that in modality $\langle \tilde{t}, \theta \rangle$ we have two parameters: $\tilde{t}$ is a string of actions from T , and θ is a real number greater than λ . This is different from the logic in [8]. $\mathcal{L}_{\sigma, \lambda}$ is often abbreviated to $\mathcal{L}_\lambda$ if σ is known from the context.

Definition 8. Let $\sigma = (S, T, \{\overset{t}{\rightarrow} : t \in T\})$ be a transition system and ρ a metric on T , and let $\lambda \in [0, \infty)$. Then the satisfaction relation $\models$ between S and $\mathcal{L}_\lambda$ with respect to ρ is defined inductively as follows:

- 1. $s \models \langle \tilde{t}, \theta \rangle F$ if there exist $\tilde{u} \in T^*$ and $s' \in S$ such that $s \xrightarrow{\tilde{u}} s'$, $\rho(\tilde{t}, \tilde{u}) < \theta$ and $s' \models F$;
- 2. $s \models \neg F$ if $s \models F$ does not hold; and
- 3. $s \models \bigwedge_{i \in I} F_i$ if $s \models F_i$ for every $i \in I$.

We can also provide a stratification for the language $\mathcal{L}_\lambda$. First we define the depth of a formula.

Definition 9. 1. For any $F \in \mathcal{L}_\lambda$, the depth $d(F) \in \text{On}$ of F is defined as follows:

- i. $d(\langle \tilde{t}, \theta \rangle F) = d(F) + 1$;
 - ii. $d(\neg F) = d(F)$; and
 - iii. $d(\bigwedge_{i \in I} F_i) = \sup_{i \in I} d(F_i)$.
2. $\mathcal{L}_\lambda^\kappa = \{F \in \mathcal{L}_\lambda : d(F) \leq \kappa\}$ for any $\kappa \in \text{On}$.

Then we are able to present our main results in this section. The following theorem gives a logical characterization of λ -bisimilarity: two processes are λ -bisimilar iff they satisfy the same formulas of the Hennessy-Milner logic.

Theorem 1. *Let ρ be an ultra-metric or $\lambda = 0$. Then*

1. *For any $\kappa \in \text{On}$, $s_1 \sim_\lambda^\kappa s_2$ iff for any $F \in \mathcal{L}_\lambda^\kappa$, $s_1 \models F$ iff $s_2 \models F$.*
2. *$s_1 \sim_\lambda s_2$ iff for any $F \in \mathcal{L}_\lambda$, $s_1 \models F$ iff $s_2 \models F$.*

Proof. (2) is immediate from (1) and Prop 8(2), so it suffices to prove (1). We proceed by transfinite induction on κ .

If $\kappa = 0$, then $\mathcal{L}_\lambda^\kappa = \{\text{true}, \text{false}\}$ (up to logical equivalence) and the conclusion is obviously correct.

Suppose that $\kappa = \mu + 1$ and the conclusion holds for μ . First, we use induction on the structure of F to show that if $s_1 \sim_\lambda^\kappa s_2$ then $s_1 \models F$ iff $s_2 \models F$ for any $F \in \mathcal{L}_\lambda^\kappa$.

Case 1. $F = \langle \tilde{t}, \theta \rangle G$. If $s_1 \models F$, then there are $\tilde{u} \in T^*$ and $s'_1 \in S$ such that $s_1 \xrightarrow{\tilde{u}} s'_1$, $\rho(\tilde{t}, \tilde{u}) < \theta$ and $s'_1 \models G$. Since $F \in \mathcal{L}_\lambda^\kappa$, $\theta > \lambda$, and $s_1 \sim_\lambda^\kappa s_2$ leads to that there are $\tilde{v} \in T^*$ and $s'_2 \in S$ such that $s_2 \xrightarrow{\tilde{v}} s'_2$, $s'_1 \sim_\lambda^\mu s'_2$, and $\rho(\tilde{u}, \tilde{v}) < \theta$ when ρ is an ultra-metric or $\rho(\tilde{u}, \tilde{v}) < \theta - \rho(\tilde{t}, \tilde{u})$ when $\lambda = 0$. Then from $G \in \mathcal{L}_\lambda^\mu$, $s'_1 \models G$ and the induction hypothesis for μ we obtain $s'_2 \models G$. In addition, if ρ is an ultra-metric then $\rho(\tilde{t}, \tilde{v}) \leq \max\{\rho(\tilde{t}, \tilde{u}), \rho(\tilde{u}, \tilde{v})\} < \theta$, and if $\lambda = 0$ then $\rho(\tilde{t}, \tilde{v}) \leq \rho(\tilde{t}, \tilde{u}) + \rho(\tilde{u}, \tilde{v}) < \theta$. So, $s_2 \models \langle \tilde{t}, \theta \rangle G = F$.

Case 2. $F = \neg G$ or $\bigwedge_{i \in I} G_i$. Immediate from the induction hypothesis for G or G_i ($i \in I$).

Secondly, we show that $s_1 \sim_\lambda^\kappa s_2$ if for any $F \in \mathcal{L}_\lambda^\kappa$, $s_1 \models F$ iff $s_2 \models F$. If not so, i.e., $s_1 \sim_\lambda^\kappa s_2$ does not hold, then there are $\theta > \lambda$ and $\tilde{t} \in T^*$ such that $s_1 \xrightarrow{\tilde{t}} s'_1$ and for all $\tilde{u} \in T^*$ and $s'_2 \in S$, $s_2 \xrightarrow{\tilde{u}} s'_2$ and $\rho(\tilde{t}, \tilde{u}) < \theta$ implies that $s'_1 \sim_\lambda^\mu s'_2$ does not hold (or the symmetric). Now, for any $\tilde{u} \in T^*$ and $s'_2 \in S$ with $s_2 \xrightarrow{\tilde{u}} s'_2$ and $\rho(\tilde{t}, \tilde{u}) < \theta$, it does not hold that $s'_1 \sim_\lambda^\mu s'_2$, and with the induction hypothesis we know that there is $G(\tilde{u}, s'_2) \in \mathcal{L}_\lambda^\mu$ such that $s'_1 \models G(\tilde{u}, s'_2)$ but $s'_2 \models G(\tilde{u}, s'_2)$ does not hold. We put

$$F = \langle \tilde{t}, \theta \rangle \bigwedge \{G(\tilde{u}, s'_2) : \tilde{u} \in T^* \wedge s'_2 \in S \wedge s_2 \xrightarrow{\tilde{u}} s'_2 \wedge \rho(\tilde{t}, \tilde{u}) < \theta\}.$$

Then for any $\tilde{u} \in T^*$ and $s'_2 \in S$ with $s_2 \xrightarrow{\tilde{u}} s'_2$ and $\rho(\tilde{t}, \tilde{u}) < \theta$, $s'_1 \models G(\tilde{u}, s'_2)$, and furthermore $s'_1 \models \bigwedge \{G(\tilde{u}, s'_2) : \tilde{u} \in T^* \wedge s'_2 \in S \wedge s_2 \xrightarrow{\tilde{u}} s'_2 \wedge \rho(\tilde{t}, \tilde{u}) < \theta\}$. In addition, $s_1 \xrightarrow{\tilde{t}} s'_1$ and $\rho(\tilde{t}, \tilde{t}) = 0 < \theta$, so $s_1 \models F$. On the other hand, $s_2 \models F$ does not hold, this contradicts that $s_1 \models F$ iff $s_2 \models F$. In fact, if $s_2 \models F$, then there are $\tilde{v} \in T^*$ and $r_2 \in S$ such that $s_2 \xrightarrow{\tilde{v}} r_2$ and $\rho(\tilde{t}, \tilde{v}) < \theta$ and $r_2 \models \bigwedge \{G(\tilde{u}, s'_2) : \tilde{u} \in T^* \wedge s'_2 \in S \wedge s_2 \xrightarrow{\tilde{u}} s'_2 \wedge \rho(\tilde{t}, \tilde{u}) < \theta\}$. However, $r_2 \models G(\tilde{v}, r_2)$ does not hold, and furthermore also $r_2 \models \bigwedge \{G(\tilde{u}, s'_2) : \tilde{u} \in T^* \wedge s'_2 \in S \wedge s_2 \xrightarrow{\tilde{u}} s'_2 \wedge \rho(\tilde{t}, \tilde{u}) < \theta\}$ does not hold, a contradiction.

Let $\kappa \in \text{On}_{\text{II}}$. If $s_1 \sim_\lambda^\kappa s_2$, then we also may use induction on the structure of F to show that $s_1 \models F$ iff $s_2 \models F$ for each $F \in \mathcal{L}_\lambda^\kappa$. If $F = \langle \tilde{t}, \theta \rangle G$, then

$d(F) = d(G) + 1 \leq \kappa$, and $d(F) < \kappa$ because $\kappa \in \text{On}_{\text{II}}$. With the induction hypothesis for $d(F) < \kappa$, we know that the conclusion holds. If $F = \neg G$ or $\bigwedge_{i \in I} G_i$ then the conclusion comes directly from the induction hypothesis for G or G_i ($i \in I$). Conversely, if for any $F \in \mathcal{L}_{\lambda}^{\kappa}$, $s_1 \models F$ iff $s_2 \models F$, then for every $\mu < \kappa$, and for every $F \in \mathcal{L}_{\lambda}^{\mu}$, $s_1 \models F$ iff $s_2 \models F$ because $\mathcal{L}_{\lambda}^{\mu} \subseteq \mathcal{L}_{\lambda}^{\kappa}$, and from the induction hypothesis we obtain $s_1 \sim_{\lambda}^{\mu} s_2$. Therefore, $s_1 \sim_{\lambda}^{\kappa} s_2$.

4 An Example in Real Time ACP

In this section we present an example for using bisimulation indexes in the analysis of real time systems. Real time processes will be described in terms of real time ACP.

4.1 Basic Real Time ACP

We recall some basic terminologies and notations of real time ACP from [3]. To present our example, we only need to use a fragment, $\text{BPA}\delta\rho\text{I}$, of real time ACP. Let A be a set and $\delta \notin A$ and $A_{\delta} = A \cup \{\delta\}$. Elements of A are called atomic actions, and δ stands for inaction. We use the set R^+ of all non-negative real numbers as the time domain. We add to a time stamp to atomic actions. Then the set of basic actions is $AT = \{a(t) : a \in A_{\delta} \wedge t \in R^+\}$. Each atomic action as well as δ is parameterized by a non-negative real number, for example, $a(t)$ means performing action a at time t . Furthermore, we assume an infinite set $\mathfrak{R}$ of process constants and an infinite set $TVar$ of time variables valued in R^+ .

Then the syntax of $\text{BPA}\delta\rho\text{I}$ is given as follows: the set ∂_{ρ} of process expressions in $\text{BPA}\delta\rho\text{I}$ is the smallest set of symbol strings fulfilling the following conditions:

1. AT , $\mathfrak{R}$, and $\{a(v) : a \in A_{\delta} \wedge v \in TVar\} \subseteq \partial_{\rho}$;
2. if $P, Q \in \partial_{\rho}$, then $P + Q, P.Q \in \partial_{\rho}$;
3. if $t \in R^+$ and $P \in \partial_{\rho}$, then $t \gg P \in \partial_{\rho}$; and
4. if $v \in V$, $V \subseteq R^+$ and $P \in \partial_{\rho}$, then $\int_{v \in V} P \in \partial_{\rho}$.

From the above definition of ∂_{ρ} , we know that there are four primary process constructs in $\text{BPA}\delta\rho\text{I}$: $+$ stands for alternative composition (choice, as in CCS) and $.$ for sequential composition. $\gg$ is the time shift and $t \gg P$ denotes the process P starting at time t , and this means that all actions of P that have to be performed at or before time T are turned into deadlocks because their execution times have already passed. $\int_{v \in V}$ is a binder of time variable v , that is, v ceases to be a free variable in $\int_{v \in V} P$ (if it happened to be one in P), and intuitively $\int_{v \in V} P$ is just the alternative composition of the alternatives $P[t/v]$ for $t \in V$. A process expression without free time variables is called a process and the set of all processes in $\text{BPA}\delta\rho\text{I}$ is denoted by $\wp_{\rho}$. As a recursion construct, a defining equation $C \stackrel{\text{def}}{=} P$ with $P \in \wp_{\rho}$ is supposed for each process constant $C \in \mathfrak{R}$.

The operational semantics of $\text{BPA}\delta\rho\text{I}$ is also given as a transition system $((\wp_\rho \cup \{\checkmark\}) \times R^+, AT \cup \{\text{idle}\}, \{\xrightarrow{\alpha} : \alpha \in AT \cup \{\text{idle}\}\})$ (see Table 1). For any $P \in \wp_\rho$ and $t \in R^+$, the pair $\langle P, t \rangle$ denotes a state at time t where the process P has still to be executed. $\checkmark$ stands for termination, and for each $t \in R^+$, $\langle \checkmark, t \rangle$ means that a process comes to termination at time t . The definition of transition relations in Table 1 implies the following properties:

1. $\langle P, r \rangle \xrightarrow{a(s)} \langle Q, t \rangle$ implies $r < s = t$. This means that state $\langle P, r \rangle$ becomes $\langle Q, t \rangle$ after performing action a at time t ;
2. $\langle P, r \rangle \xrightarrow{\text{idle}} \langle Q, s \rangle$ implies $r < s$ and $P = Q$. This means that process P idles from time r to time s . $\langle P, r \rangle \xrightarrow{\text{idle}} \langle P, s \rangle$ is always abbreviated as $\langle P, r \rangle \rightarrow \langle P, s \rangle$;
3. $\langle P, r \rangle \xrightarrow{a(s)} \langle \checkmark, t \rangle$ implies $r < s = t$. This means that process P at time r can perform action a at time t and then it terminates; and
4. it is impossible that $\langle \checkmark, r \rangle \xrightarrow{\alpha} X$ for any $\alpha \in AT \cup \{\text{idle}\}$ and $X \in (\wp_\rho \cup \{\checkmark\}) \times R^+$.

The transition relations $\xrightarrow{\alpha}$ with $\alpha \in AT \cup \{\text{idle}\}$ are defined by the rules in Table 1.

4.2 Approximate Bisimilarity in Real Time ACP

In [3], the concept of bisimulations in $\text{BPA}\delta\rho\text{I}$ was introduced in the usual way. Let S be a binary relation on $(\wp_\rho \cup \{\checkmark\}) \times R^+$. If $\langle X, r \rangle S \langle \checkmark, s \rangle$ or $\langle \checkmark, s \rangle S \langle X, r \rangle$ implies $X = \checkmark$, then S is said to be normal. A normal binary relation S on $(\wp_\rho \cup \{\checkmark\}) \times R^+$ is called a bisimulation if and only if for any $P, Q \in \wp_\rho$, $t \in R^+$ and $\alpha \in AT \cup \{\text{idle}\}$,

- i. whenever $\langle P, t \rangle \xrightarrow{\alpha} \langle P', r \rangle$ then, for some $Q' \in \wp_\rho$, $\langle Q, t \rangle \xrightarrow{\alpha} \langle Q', r \rangle$ and $\langle P', r \rangle S \langle Q', r \rangle$; and
- ii. whenever $\langle Q, t \rangle \xrightarrow{\alpha} \langle Q', r \rangle$ then, for some $P' \in \wp_\rho$, $\langle P, t \rangle \xrightarrow{\alpha} \langle P', r \rangle$ and $\langle P', r \rangle S \langle Q', r \rangle$.

For any $P, Q \in \wp_\rho$, P and Q are bisimilar, denoted as $P \sim Q$, if there is a bisimulation S with $\langle P, 0 \rangle S \langle Q, 0 \rangle$.

It is easy to see from the above definition that two bisimilar processes must perform the same action exactly at the same time instant. However, we may also be interested in some weaker equivalences in which a small error of execution time of action is admissible, and such equivalences are even more realistic in some situations. So, in the sequel we are going to introduce bisimulation indexes in $\text{BPA}\delta\rho\text{I}$. First, we define a natural metric on $AT \cup \{\text{idle}\}$:

$$\begin{aligned} \rho(a(r), b(s)) &= \begin{cases} |r - s|, & a = b \\ \infty, & \text{otherwise} \end{cases}, \\ \rho(a(r), \text{idle}) &= \rho(\text{idle}, a(r)) = \infty, \\ \rho(\text{idle}, \text{idle}) &= 0. \end{aligned}$$

Atom

$$\frac{}{\langle a(r), t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle} \quad (t < r) \quad \frac{}{\langle a(r), t \rangle \rightarrow \langle a(r), s \rangle} \quad (t < s < r)$$

$$\frac{}{\langle \delta(r), t \rangle \rightarrow \langle \delta(r), s \rangle} \quad (t < s < r)$$

Alternative

$$\frac{\langle P, t \rangle \xrightarrow{a(r)} \langle P', r \rangle}{\langle P + Q, t \rangle \xrightarrow{a(r)} \langle P', r \rangle} \quad \frac{\langle Q, t \rangle \xrightarrow{a(r)} \langle Q', r \rangle}{\langle P + Q, t \rangle \xrightarrow{a(r)} \langle Q', r \rangle} \quad \frac{\langle P, t \rangle \rightarrow \langle P, r \rangle}{\langle P + Q, t \rangle \rightarrow \langle P + Q, r \rangle}$$

$$\frac{\langle Q, t \rangle \rightarrow \langle Q, r \rangle}{\langle P + Q, t \rangle \rightarrow \langle P + Q, r \rangle} \quad \frac{\langle P, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}{\langle P + Q, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle} \quad \frac{\langle Q, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}{\langle P + Q, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}$$

Sequential

$$\frac{\langle P, t \rangle \xrightarrow{a(r)} \langle P', r \rangle}{\langle P.Q, t \rangle \xrightarrow{a(r)} \langle P'.Q, r \rangle} \quad \frac{\langle P, t \rangle \rightarrow \langle P, r \rangle}{\langle P.Q, t \rangle \rightarrow \langle P.Q, r \rangle} \quad \frac{\langle P, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}{\langle P.Q, t \rangle \xrightarrow{a(r)} \langle Q, r \rangle}$$

Shift

$$\frac{\langle P, t \rangle \xrightarrow{a(r)} \langle P', r \rangle}{\langle s \gg P, t \rangle \xrightarrow{a(r)} \langle P', r \rangle} \quad (r > s) \quad \frac{}{\langle s \gg P, t \rangle \rightarrow \langle s \gg P, r \rangle} \quad (t < r < s)$$

$$\frac{\langle P, t \rangle \rightarrow \langle P, r \rangle}{\langle s \gg P, t \rangle \rightarrow \langle s \gg P, r \rangle} \quad \frac{\langle P, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}{\langle s \gg P, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle} \quad (r > s)$$

Integration (for all rules: $u \in V$)

$$\frac{\langle P[u/v], t \rangle \xrightarrow{a(r)} \langle P', r \rangle}{\langle \int_{v \in V} P, t \rangle \xrightarrow{a(r)} \langle P', r \rangle} \quad \frac{\langle P[u/v], t \rangle \rightarrow \langle P[u/v], r \rangle}{\langle \int_{v \in V} P, t \rangle \rightarrow \langle \int_{v \in V} P, r \rangle} \quad \frac{\langle P[u/v], t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}{\langle \int_{v \in V} P, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}$$

Constant (for all rules: $C \stackrel{\text{def}}{=} P$)

$$\frac{\langle P, t \rangle \xrightarrow{a(r)} \langle P', r \rangle}{\langle C, t \rangle \xrightarrow{a(r)} \langle P', r \rangle} \quad \frac{\langle P, t \rangle \rightarrow \langle P, r \rangle}{\langle C, t \rangle \rightarrow \langle C, r \rangle} \quad \frac{\langle P, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}{\langle C, t \rangle \xrightarrow{a(r)} \langle \sqrt{}, r \rangle}$$

Table1. Rules for the transition relation $\xrightarrow{\alpha}$

It is straightforward to show that ρ is a metric on $AT \cup \{\text{idle}\}$, and then ρ induces directly a bisimulation index and the concept of λ -bisimulations in the transition system $((\wp_\rho \cup \{\sqrt{}\}) \times R^+, AT \cup \{\text{idle}\}, \{\overset{\alpha}{\rightarrow} : \alpha \in AT \cup \{\text{idle}\}\})$. If $P, Q \in \wp_\rho$ and there is a normal binary relation S on $(\wp_\rho \cup \{\sqrt{}\}) \times R^+$ such that $\langle P, 0 \rangle S \langle Q, 0 \rangle$ and S is a λ -bisimulation, then P and Q are said to be λ -bisimilar and we write $P \sim_\lambda Q$.

Lemma 1. *A normal binary relation S on $(\wp_\rho \cup \{\sqrt{}\}) \times R^+$ is a λ -bisimulation if and only if $\langle P, r \rangle S \langle Q, s \rangle$ implies, for all $a \in A_\delta$ and $t \in R^+$,*

- i. whenever $\langle P, r \rangle \rightarrow \langle P, r' \rangle$ then, for some s' , $\langle Q, s \rangle \rightarrow \langle Q, s' \rangle$ and $\langle P, r' \rangle S \langle Q, s' \rangle$;
- ii. whenever $\langle Q, s \rangle \rightarrow \langle Q, s' \rangle$ then, for some r' , $\langle P, r \rangle \rightarrow \langle P, r' \rangle$ and $\langle P, r' \rangle S \langle Q, s' \rangle$;
- iii. whenever $\langle P, r \rangle \xrightarrow{a(t)} \langle P', t \rangle$ and $\lambda' > \lambda$ then, for some Q' and u , $\langle Q, s \rangle \xrightarrow{a(u)} \langle Q', u \rangle$, $|t - u| < \lambda'$ and $\langle P', t \rangle S \langle Q', u \rangle$;
- iv. whenever $\langle Q, s \rangle \xrightarrow{a(t)} \langle Q', t \rangle$ and $\lambda' > \lambda$ then, for some P' and u , $\langle P, r \rangle \xrightarrow{a(u)} \langle P', u \rangle$, $|t - u| < \lambda'$ and $\langle P', u \rangle S \langle Q', t \rangle$;
- v. whenever $\langle P, r \rangle \xrightarrow{a(t)} \langle \surd, t \rangle$ and $\lambda' > \lambda$ then, for some u , $\langle Q, s \rangle \xrightarrow{a(u)} \langle \surd, u \rangle$, $|t - u| < \lambda'$ and $\langle \surd, t \rangle S \langle \surd, u \rangle$; and
- vi. whenever $\langle Q, s \rangle \xrightarrow{a(t)} \langle \surd, t \rangle$ and $\lambda' > \lambda$ then, for some u , $\langle P, r \rangle \xrightarrow{a(u)} \langle \surd, u \rangle$, $|t - u| < \lambda'$ and $\langle \surd, u \rangle S \langle \surd, t \rangle$.

Proof. Immediate from Def. 1, 2 and 3.

Example 2. The following two processes

$$\int_{v \in [0,1]} a(v)$$

and

$$\int_{v \in [0,1)} a(v)$$

are 0-bisimilar. This example shows that the condition $|t - u| < \lambda'$ for all $\lambda' > \lambda$ cannot be simplified to $|t - u| \leq \lambda$ if the process definition contains integration constructs.

Now, we give an example in $\text{BPA}\delta\rho\text{I}$ in which we can see that bisimulation indexes are satisfactory tools to describe a certain relationship among real time systems.

Example 3. Three clocks.

As an application of the integration construct, Baeten and Bergstra described three different but more or less similar clocks in terms of in $\text{BPA}\delta\rho\text{I}$ (see [3, Ex. 5.6]). The first clock is absolutely precise and it is defined as

$$C_1(t) \stackrel{\text{def}}{=} \text{tick}(t).C_1(t+1) .$$

If one starts the clock in state $\langle C_1(t), 0 \rangle$ it will start ticking at time t and continue to do so each time unit with absolute precision. The second clock allows a tolerance for the ticks of 0.02 time units:

$$C_2(t) \stackrel{\text{def}}{=} \int_{v \in [t-0.01, t+0.01]} \text{tick}(v).C_2(t+1) ;$$

whereas the third clock cumulates the errors:

$$C_3(t) \stackrel{\text{def}}{=} \int_{v \in [t-0.01, t+0.01]} \text{tick}(v).C_2(v+1) .$$

With bisimulation we are unable to distinguish $C_3(t)$ from $C_2(t)$ with respect to C_1 because both of them are not bisimilar to $C_1(t)$. However, bisimulation index is more discriminative, and we may prove that $C_1(t)$ and $C_2(t)$ are approximate bisimilar with a possible error (of 0.01) whereas $C_1(t)$ and $C_3(t)$ are not approximate bisimilar for any given error. More exactly, we have

- a. $C_1(t) \sim C_2(t)$ does not hold, but $C_1(t) \sim_{0.01} C_2(t)$ if $t > 0$; and
- b. for any $\lambda < \infty$, $C_1(t) \sim_\lambda C_3(t)$ does not hold.

We also can give a logical description of the relationship of $C_1(t)$, $C_2(t)$ and $C_3(t)$ with the modal logic proposed in Sect. 3. For example, the following modal logic formula discriminates $C_1(t)$ and $C_3(t)$

$$G_n = \langle \text{tick}(t+0.01), \theta \rangle \langle \text{tick}(t+1.02), \theta \rangle \dots \langle \text{tick}(t+1.01n+0.01), \theta \rangle \text{true} ,$$

where $\lambda \in [0, \infty)$, $n = \lceil 100\lambda \rceil + 1$, $[x]$ stands for the integer part of real number x , $\theta = \lambda + 0.01$ and $\text{true} \stackrel{\text{def}}{=} \bigwedge_{i \in \phi} F_i$. It may be proven that $G_n \in \mathcal{L}_\lambda$, and $\langle C_3(t), 0 \rangle \models G_n$ but it does not hold that $\langle C_1(t), 0 \rangle \models G_n$ (and consequently, G_n does not hold for $\langle C_2(t), 0 \rangle$, too).

5 Related Works and Conclusion

As shown in the previous sections approximate bisimulation is a direct generalization of bisimulation which possesses many of the properties of bisimulation such as a Hennessy-Milner logical characterization. Moreover, approximate bisimulation provides a whole range of equivalences for studying the degree of bisimilarity of two processes. This makes approximate bisimilarity to be a good basis for reasoning about the operational behaviour of processes in a continuous (or more generally, non-discrete) setting. In particular, approximate bisimilarity provides us with a formal tool for describing approximate correctness of programs where some quantifiable, bounded error of the implementation is tolerated. For example, the "tolerant" clock C_2 is an approximately correct implementation of the "precise" clock C_1 (see Section 4).

The notion of bisimulation index is the key to approximate bisimilarity. The bisimulation index is a measure of the similarity of a relation R with bisimulation. In the paper we have shown some basic properties of bisimulation indexes. Because of limit of space, we omit some deeper theoretic results on bisimulation indexes and more examples here. For these aspects, we refer to a longer version [12] of this paper in which preservation of bisimulation indexes under various operations of transition systems is shown and some fundamental properties of strong and weak bisimulation indexes in CCS are elaborated. In [12] we

present also two examples in timed CCS [11] to demonstrate further that bisimulation indexes are suitable to characterize the relationship between specifications of real time systems and their approximate implementations.

The mathematical structures that we deal with in this paper are labelled transition systems whose sets of labels are equipped with metrics. The metric approach was first introduced by Arnold and Nivat in the framework of infinite trees to give semantics of nondeterministic recursive programs [1, 2]. De Bakker and Zucker proposed a denotational semantics of processes using metrics [5]. Afterwards, metric semantic models for a large variety of program constructs have been developed by the Amsterdam Concurrency Group, see for examples [4] and [10]. In order to give a metric comparative semantics of a kernel fragment of real time ACP, F. van Breugel [9] used metric labelled transition systems in which both the set of labels and the set of states are equipped with metrics. It is worth noting that metric labelled transition systems were successfully used to describe a compact operational semantics of programming languages with dense choices and the main results in [6] were considerably generalized. For an excellent exposition on this topic, see [9].

As mentioned above, in this paper only the set of labels in a labelled transition system is equipped with a metric; by contrast in van Breugel's approach [9] both the set of labels and the set of states in a labelled transition systems are equipped with metrics. More essentially, our motivation is different from van Breugel's. van Breugel aimed at providing metric comparative semantics for programming languages with dense choice. But the purpose of this paper is to introduce an approximate version of bisimulations which may be used to characterize approximate correctness of concurrent programs. It seems that many results in this paper can be generalized into the setting of van Breugel's metric labelled transition systems by adjusting slightly the definition of bisimulation indexes.

References

- [1] A. Arnold and M. Nivat. Metric Interpretations of Infinite Trees and Semantics of Nondeterministic Recursive Programs. *Theoretical Computer Science*, 11:181–205, 1980.
- [2] A. Arnold and M. Nivat. The Metric Space of Infinite Trees, Algebraic and Topological Properties. *Fundamenta Informaticae*, 4:445–476, 1980.
- [3] J. C. M. Baeten and J. Bergstra. Real-Time Process Algebra. *Formal Aspects of Computing*, 3:142–188, 1991.
- [4] J. W. de Bakker and J. J. M. M. Rutten, editors. *Ten Years of Concurrency Semantics, Selected Papers of the Amsterdam Concurrency Group*. World Scientific, Singapore, 1992.
- [5] J. W. de Bakker and J. I. Zucker. Processes and the Denotational Semantics of Concurrency. *Information and Control*, 54:70–120, 1982.
- [6] J. N. Kok and J. J. M. M. Rutten. Contractions in Comparing Concurrency Semantics. *Theoretical Computer Science*, 76:179–222, 1990.
- [7] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin, 1980.

- [8] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
- [9] F. van Breugel. *Comparative Metric Semantics of Programming Languages: Non-determinism and Recursion*. Birkhäuser, Boston, 1998.
- [10] R. J. van Glabbeek and J. J. M. M. Rutten. The Processes of de Bakker and Zucker Represent Bisimulation Equivalences Classes. In J. W. de Bakker, editor, *25 Jaar Semantik*, Liber Amicorum, pages 243–246. CWI, Amsterdam, 1989.
- [11] Y. Wang. Real-Time Behaviour of Asynchronous Agents. In J. C. M. Baeten and J. W. Klop, editors, *Proc. CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer, Berlin, 1990.
- [12] M. S. Ying and M. Wirsing. Approximate Bisimilarity and Its Applications. Technical report 9906, Institut für Informatik, Ludwig-Maximilians-Universität München, 1999.

Time and Probability in Process Algebra^{*}

Suzana Andova

Department of Mathematics and Computing Science
Eindhoven University of Technology The Netherlands,
`suzana@win.tue.nl`

Abstract. In the paper we present an ACP-like process algebra which can be used to model both probabilistic and time behaviour of parallel systems. This process algebra is obtained by extension of untimed probabilistic process algebra with constructors that allow the explicit specification of timing aspects. In this paper we concentrate on giving axioms and deduction rules for these constructors. We give two probabilistic process algebras with discrete time. The first one only manipulates with processes that may be initialized within the current time slice or may delay a finite and fixed number of time slices. Later, we add processes whose execution can be postponed for an arbitrary number of time slices.

1 Introduction

This paper proposes a compositional algebraic approach to the specification of probabilistic discrete-time systems.

In the past years, the need for reasoning about probabilistic behaviour of software and hardware systems has triggered much interest in the area of formal methods ([21,15,20,16,11,1,2]). The general approach taken has been to extend existing models and techniques which have proved successful in the non-probabilistic setting with a notion of probability.

On the other hand, the correctness of some systems hinges on the timing aspects, or the timing constraints themselves of a system are the object of study. In such cases untimed models are of little help. Thus, various time extensions of the standard formal methods have been proposed ([18,16,17,4,6,7,23,8]).

Simultaneously introducing time and probability provides a new aspect to the specification and verification of software and hardware systems. It allows performance properties related to various forms of time delay to be specified and verified, and also reliability analysis, such as, average time during which the system operates correctly.

In this paper we present an ACP-like process algebra which can be used to model both probabilistic and time behaviour of systems. It is obtained as a discrete-time extension of the untimed probabilistic process algebra in [1,2], which combines probabilities and non-determinism. Time is introduced by cutting it up into a countably infinite number of time slices and clock ticks take

^{*} Research is supported by PROMACS project, SION 612-10-000 of the Netherlands Organisation for Scientific Research (NWO)

place between these time slices. Moreover, we do not assume that an absolute clock exists, but the moment when an action occurs is measured with respect to the previous action. This gives the relative aspect of timing. Thus, we end with a discrete relative time version of probabilistic process algebra. In the paper actually we present three different process algebras built up in a modular way. First, we extend the Basic process algebra with probabilistic choice ($[1,2]$) with time constants and time operators. There are constants presenting undelayable actions, actions that can execute only in the time slice they are initialized in. To make the passage of time explicit, the time unit delay operator is introduced. It postpones the execution of a process to the next time slice. In this process algebra only processes that are initialized within the current time slice or processes that delay for a finite and fixed number of time slices can be specified. In order to cope with more complex systems it is necessary to extend this algebra by processes that can be initialized in the current or any future time slice. Doing so, we obtain the Probabilistic basic process algebra with delayable actions.

Next we introduce parallel composition and communication. Even in the untimed version this turned out to be a non-trivial problem. Due to the presence of non-determinism, the introduction of probabilistic choice and next, parallel composition may be done in various ways. The model of the parallel composition we have worked with in untimed theory (see also [13]) needs some extra operator ($\parallel$) in order to obtain a finite axiomatization of parallel composition. To obtain a finite axiomatization and clear deduction rules in the proposed time extension we add again extra operators and a normalizing function which helps to define the probability distribution function over parallel processes.

Besides the axiom system we investigate the operational semantics of probabilistic time processes based on the alternating model and probabilistic bisimulation equivalence as proposed by Larsen and Skou in [21]. First, the operational semantics for the basic process algebra is presented and later it is extended to the operational semantics of the process algebra with a notion of parallel composition and communication. For both we prove that the axiomatization is complete for probabilistic bisimulation equivalence.

Related work has been done in [16] where the author presents a probabilistic and time extension of CCS. He relates the algebra with the TPCTL temporal logic used for verification of properties of a system. We use a different model of parallel composition in our formalism, which in our opinion is less deterministic and we propose a pure algebraic method that will be used for specification as well as for verification of concurrent systems. The example that we give, the PAR protocol, has an algebraic specification. The verification part consists of an algebraic calculation that transforms the protocol specification into a process which can be treated as a Markov chain. By this, the verification is reduced to a performance analysis of that Markov chain.

2 Basic Process Algebra

In this section we present the signatures and the axiom systems of two basic process algebras. The attribute *basic* indicates that this axiomatizations do not have any notion of concurrency. An extension with parallel composition is given in Section 3. According to the notation used in [23] (which we often refer to) in the names of the process algebras presented here the addition $-ID$ should be used. It indicates that the immediate deadlock is not included in the algebras. However, since no confusion can arise in the presented probabilistic process algebras, in our notation we will omit this addition.

2.1 Probabilistic Process Algebra with Undelayable Actions

Axiom System First, we define the signature, that is the constants and operators from which the process terms of $\text{prBPA}_{\text{drt}}^-$ are built. Probabilistic basic process algebra with discrete relative time without delayable actions, $\text{prBPA}_{\text{drt}}^-$, for a certain set of atomic actions A has the signature Σ_{prB-} containing:

- constants: $\underline{a}$ for each $a \in A$ (undelayable atomic actions)
a special constant $\underline{\delta}, \delta \notin A$ (undelayable deadlock)
- the binary operators: $+$ (non-deterministic choice operator)
 $\cdot$ (sequential composition operator)
 $\oplus_\pi$ for each $\pi \in \langle 0, 1 \rangle$ (probabilistic choice)
- the unary operators: σ_{rel} (unit delay operator)
 ν_{rel} (now operator).

The non-deterministic choice operator and the sequential composition operator are standard operators from ACP ([9]). The unit delay operator postpones the execution of a process to the next time slice. The now operator denotes the part of a process that can do an action within the first time slice. The probabilistic choice operator expresses that the choice between two processes is determined by a probabilistic distribution. Process $p \oplus_\pi q$ behaves as p with probability π and as q with probability $1 - \pi$. This operator can easily be extended to an n -ary operator ([1]).

The axiom system has the axioms from $\text{BPA}_{\text{drt}}^- - ID$ ([23]) given in Table 1, with a modified idempotency law, the axioms from prBPA ([1,2]) for the probabilistic choice operator in Table 2 and the new axioms which express the relations between the time operators and the probabilistic choice operator (Table 3).

The axiom for distribution, *PAC5* shows that the probabilistic choice is resolved before the non-deterministic choice. This reasoning comes from our approach that probabilistic choice is determined by the internal behaviour of the system and it is made before any action is executed. On the other side, the non-deterministic choice is made exactly at the moment when the first action occurs. (See also the definition of the partial choice operator in [5].)

Axiom *PDRT* needs also some explanation. For probabilistic choice as well as for non-deterministic choice we use the concept of weak time factorization which expresses that time passing does not determine a choice. If both processes

$x + y = y + x$	A1	$\sigma_{rel}(x) + \sigma_{rel}(y) = \sigma_{rel}(x + y)$	<i>DRT1</i>
$(x + y) + z = x + (y + z)$	A2	$\sigma_{rel}(x) \cdot y = \sigma_{rel}(x \cdot y)$	<i>DRT2</i>
$\underline{a} + \underline{a} = \underline{a}$	AA3	$\sigma_{rel}(x) + \underline{\delta} = \sigma_{rel}(x)$	<i>DRT5</i>
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4	$\nu_{rel}(\underline{a}) = \underline{a}$	<i>DCS1</i>
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$\nu_{rel}(x + y) = \nu_{rel}(x) + \nu_{rel}(y)$	<i>DCS2</i>
$\underline{a} + \underline{\delta} = \underline{a}$	A6	$\nu_{rel}(x \cdot y) = \nu_{rel}(x) \cdot y$	<i>DCS3</i>
$\underline{\delta} \cdot x = \underline{\delta}$	A7	$\nu_{rel}(\sigma_{rel}(x)) = \underline{\delta}$	<i>DCS4</i>

Table 1. Axioms of $BPA_{drt}^- - ID$.

$x \Downarrow_{\pi} y = y \Downarrow_{1-\pi} x$	<i>PAC1</i>	$\sigma_{rel}(x \Downarrow_{\pi} y) = \sigma_{rel}(x) \Downarrow_{\pi} \sigma_{rel}(y)$	<i>PDRT</i>
$x \Downarrow_{\pi} (y \Downarrow_{\rho} z) = (x \Downarrow_{\pi+\rho-\pi\rho} y) \Downarrow_{\pi+\rho-\pi\rho} z$	<i>PAC2</i>	$\nu_{rel}(x \Downarrow_{\pi} y) = \nu_{rel}(x) \Downarrow_{\pi} \nu_{rel}(y)$	<i>PDCS</i>
$x \Downarrow_{\pi} x = x$	<i>PAC3</i>		
$(x \Downarrow_{\pi} y) \cdot z = x \cdot z \Downarrow_{\pi} y \cdot z$	<i>PAC4</i>		
$(x \Downarrow_{\pi} y) + z = (x + z) \Downarrow_{\pi} (y + z)$	<i>PAC5</i>		

Table 3. Probabilistic choice operator and time operators.**Table 2.** Probabilistic choice operator.

in probabilistic choice can idle to the next time slice then the moment when the choice is made does not affect the outcome of the choice. Let us consider the following processes: $\sigma_{rel}(\underline{a}) \Downarrow_{\pi} \sigma_{rel}(\underline{b})$ and $\sigma_{rel}(\underline{a} \Downarrow_{\pi} \underline{b})$. In the first process the probabilistic choice is resolved within the current time slice. After that a time tick occurs and then, in the second time slice, action a is executed with probability π and action b is executed with probability $1 - \pi$. The second process idles one time unit and then the probabilistic choice is resolved. Again, in the second time slice action a appears with probability π and action b with probability $1 - \pi$. (At this point our approach differs from [16].)

2.2 Probabilistic Process Algebra with Delayable Actions

Up to now we have only dealt with atomic actions that must be executed in the current time slice. If communication between such atomic actions does not occur in the current time slice the system ends in deadlock, if there is no other alternative of course. In specification of parallel systems usually communication is initiated by one of two processes that communicate. For example in the PAR protocol given in Sec. 4 in a communicating system sender - channel, the channel should be always ready to accept a message sent by the sender. Thus the communication is controlled and initiated by the sender and the “send” process can clearly be specified by undelayable atomic actions (Sec. 2.1). But we need to find a way to specify the passive behaviour of the channel, that is, a process that is “able to accept a message from the sender whenever it arrives”!

Axiom System In this section we extend the signature by new atomic actions which are denoted as $a, b, c, \dots$. The action a means intuitively “execute action a in the current or any future time slice and terminate after the execution” (axiom $a = \underline{a} + \sigma_{rel}(a)$). We call these atomic actions delayable. In addition to these new processes we introduce the constant δ which stands for livelock, that is, time can pass but no further action is possible (axiom $\delta = \underline{\delta} + \sigma_{rel}(\delta)$).

Thus we obtain the Probabilistic basic process algebra with discrete relative time and delayable actions, $\text{prBPA}_{\text{drt}}^+$, with the signature Σ_{prB^+} containing the constants and operators from Σ_{prB^-} and the new constants: a for each $a \in A$ (delayable atomic actions) and a special constant $\delta, \delta \notin A$ (livelock). The set of closed terms over signature Σ_{prB^+} is denoted by $\mathcal{SP}$. (In the operational semantics these terms are called probabilistic processes.)

The axioms for the new constants are given in Table 4 ($a \in A_\delta$). One can note that only one of them is given as an equality and the others are conditional axioms. We have introduced here delayable atomic actions only like in [6], but not the unbounded delayable time operator or the iterated delay operator like it has been done in [7,23]. We do so in order to obtain a simpler process algebra since combining the probabilistic choice operator and this operator leads to a very complicated axiomatization. (This is actually a consequence of the way we combine the probabilistic choice and the non-deterministic choice.) Of course, we have to restrict the RSP(USD) rule ([23]) and thus we have four restricted variants of this rule.

$a = \underline{a} + \sigma_{rel}(a)$		DA1
$y = \underline{a} + \sigma_{rel}(y)$	$\Rightarrow y = a$	DA2
$y = \underline{a} \cdot x + \sigma_{rel}(y)$	$\Rightarrow y = a \cdot x$	DA3
$z = z + z, y = \underline{a} + \nu_{rel}(z) + \sigma_{rel}(y), u = \nu_{rel}(z) + \sigma_{rel}(u)$	$\Rightarrow y = a + u$	DA4
$z = z + z, y = \underline{a} \cdot x + \nu_{rel}(z) + \sigma_{rel}(y), u = \nu_{rel}(z) + \sigma_{rel}(u)$	$\Rightarrow y = a \cdot x + u$	DA5

Table 4. Axioms for delayable actions

Basic Terms We define the set of basic terms $\mathcal{B}$ inductively, with the help of an intermediate set $\mathcal{B}_+ \subseteq \mathcal{B}$. In $\mathcal{B} \setminus \mathcal{B}_+$ the outermost operator is a probabilistic choice operator. Elements of $\mathcal{B}_+$ are all constants and terms that have as the outermost operator a non-deterministic choice operator, a sequential composition or a delay operator.

1. $a \in A_\delta \Rightarrow \underline{a}, a \in \mathcal{B}_+$;
2. $a \in A, t \in \mathcal{B} \Rightarrow \underline{a} \cdot t, a \cdot t \in \mathcal{B}_+$;
3. $t, s \in \mathcal{B}_+ \Rightarrow t + s, \sigma_{rel}(t) \in \mathcal{B}_+$;
4. $t, s \in \mathcal{B} \Rightarrow t \dot{+}_\pi s \in \mathcal{B}$ for $\pi \in \langle 0, 1 \rangle$.

Theorem 1 ((Elimination theorem)). *Let p be a closed $\text{prBPA}_{\text{drt}}^+$ term. Then there is a basic term q such that $\text{prBPA}_{\text{drt}}^+ \vdash p = q$.* $\square$

Some axioms have a condition of the form: $z = z + z$. In the theory this equality holds for all terms which have as a basic term a term from $\mathcal{B}_+$ (called trivial probabilistic processes). In the model the property $z \Leftrightarrow z + z$ is fulfilled by all processes which cannot do probabilistic transitions to different equivalence classes (Lemma 5). Without this condition, due to the distribution law (axiom *PAC5*), contra-intuitive process terms are obtained. For example, if $z \equiv \underline{a} \dot{\vdash}_\pi \underline{b}$ ($z \neq z + z$) and if y is a process such that $y = z + \sigma_{rel}(y)$, then we obtain

$$\begin{aligned} y &= (\underline{a} + \sigma_{rel}(y)) \dot{\vdash}_\pi (\underline{b} + \sigma_{rel}(y)) \\ &= (\underline{a} + \sigma_{rel}(\underline{a} + \sigma_{rel}(y))) \dot{\vdash}_{\pi^2} (\underline{a} + \sigma_{rel}(\underline{b} + \sigma_{rel}(y))) \dot{\vdash}_{\pi(1-\pi)} \\ &\quad (\underline{b} + \sigma_{rel}(\underline{a} + \sigma_{rel}(y))) \dot{\vdash}_{\pi(1-\pi)} (\underline{b} + \sigma_{rel}(\underline{b} + \sigma_{rel}(y))) \text{ and } y \neq a \dot{\vdash}_\pi b. \end{aligned}$$

Structured Operational Semantics We presented two process algebras. One of them is an extension of the other one. Now we give the operational semantics of the larger axiom system $\text{prBPA}_{\text{drt}}^+$ which is an operational extension of the operational semantics of $\text{prBPA}_{\text{drt}}^-$.

The operational semantics consists of four types of deduction rules, rules for probabilistic transition: $\leadsto$ (which are unlabelled), rules for action transitions: $\xrightarrow{a}$ (which are labelled with atomic actions $a \in A$), rules for time transition: $\xrightarrow{\sigma}$ and rules for the $\mathbb{D}$ predicate. The probabilistic transition, $p \leadsto x$, expresses that with some non-zero probability process p behaves as process x . The value of this probability is determined by the probability distribution function defined by Definition 2. The meanings of $\xrightarrow{a}$ and $\xrightarrow{\sigma}$ are the same as in the non-probabilistic discrete-time versions of ACP, $x \xrightarrow{a} y$ means that x can perform action a and afterwards it continues like y ; $x \xrightarrow{\sigma} y$ means that x can perform a time step and then in the next time slice in behaves like y .

As we have stated already, the time passing cannot resolve a probabilistic choice. In the algebra we have axiom *PrDRT1* which makes terms $\sigma_{rel}(\underline{a} \dot{\vdash}_{0.5} \underline{b})$ and $\sigma_{rel}(\underline{a}) \dot{\vdash}_{0.5} \sigma_{rel}(\underline{b})$ to be considered equal. In the operational semantics the interpretations of these two processes have to be bisimilar. This requirement can be met in at least two ways. The first option is that these two terms have two different operational interpretations. The most appropriate interpretation of these processes are given in Figure 1. In order to make them bisimilar we have to change the definition of bisimulation relation in a way that it would relate these processes, which means a complex and non-intuitive definition. Also a lot of deduction rules have to be added in order to cover all sequences which only differ in the order of $\leadsto$ and $\xrightarrow{\sigma}$'s. This approach is not appropriate for technical reasons as well. Namely, without any constraints about the order of $\leadsto$ and $\xrightarrow{\sigma}$ transitions the formulation of many propositions, and moreover their proofs, are difficult and unclear. Thus, we come up with another idea to interpret these two terms exactly the same. In this way we obtain simple deduction rules of the operational semantics that guarantee some useful properties about the order in which $\leadsto$ and $\xrightarrow{\sigma}$ transitions appear.

Like in the untimed probabilistic process algebras in [2,1] the operational semantics of $\text{prBPA}_{\text{drt}}^+$ is based on the alternating model ([16]). Every probabilistic

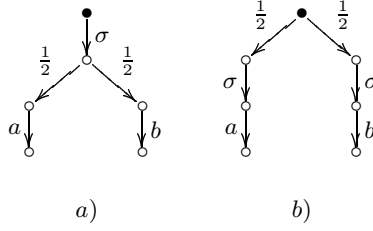


Fig. 1. Transition graphs of two processes

transition is followed by action transitions possibly preceded by σ -transitions. In order to achieve alternation of transitions we introduce new constants and new processes, called non-deterministic processes. These processes can only perform an action transition or a σ -transition but not a probabilistic transition. The set of non-deterministic processes is denoted by $\mathcal{DP}$ and its elements by $\check{x}$.

The signature Σ_{prBos}^+ of the term deduction system of $\text{prBPA}_{\text{drt}}^+$ contains the constants and operators from Σ_{prB+} and more, the new constants: $\underline{\check{a}}$ for each $a \in A$ (non-deterministic undelayable atomic actions), $\underline{\check{\delta}}$ (non-deterministic undelayable deadlock), $\check{a}$ for each $a \in A$ (non-deterministic delayable atomic actions), $\check{\delta}$ (non-deterministic livelock). Informally, these constants can be considered as $\underline{\check{a}}$, $\underline{\check{\delta}}$, $\check{a}$ and $\check{\delta}$, respectively, from non-probabilistic discrete-time process algebras. $\mathcal{PR}$ denotes the set of all probabilistic and non-deterministic processes, that is, $\mathcal{SP} \cup \mathcal{DP}$.

The semantics of $\text{prBPA}_{\text{drt}}^+$ is given by the deduction rules of $\text{prBPA}_{\text{drt}}^-$ shown in Table 5 ($a \in A$), the deduction rules for delayable actions given in Table 6 ($a \in A$), the probability distribution function defined by Definition 2 and the probabilistic bisimulation relation defined by Definition 3.

The $\mathbb{D}$ predicate is not essential in the operational semantics and it can be obtained from the other relations (transitions). Actually, we can prove that if $p \in \mathcal{SP}$, then $\mathbb{D}(p)$ iff $\exists x, y : p \rightsquigarrow x \ \& \ x \xrightarrow{\sigma} y$ and if $x \in \mathcal{DP}$, then $\mathbb{D}(x)$ iff $\exists y : x \xrightarrow{\sigma} y$. We add this predicate to the semantics in order to get simple rules for the parallel composition operator in Section 3. From the deduction rules we can easily prove that: if p is an $\mathcal{SP}$ term and $p \rightsquigarrow x$, then $x \in \mathcal{DP}$; if p is an $\mathcal{SP}$ term, then $p \not\rightsquigarrow$; if x is a $\mathcal{DP}$ term and $x \xrightarrow{\sigma} y$, then $y \in \mathcal{DP}$; if x is a $\mathcal{DP}$ term and $x \xrightarrow{a} p$ for some $a \in A$, then $p \in \mathcal{SP}$.

Definition 2. A probability distribution function $\mu : \mathcal{PR} \times \mathcal{PR} \rightarrow [0, 1]$ is defined inductively as follows:

$$\begin{array}{ll}
 \mu(x, \check{x}) &= 1, \text{ for } x \in \{\underline{\check{a}}, \underline{\check{\delta}}, a, \delta\}, \\
 \mu(p \cdot q, x' \cdot q) &= \mu(p, x'), \\
 \mu(p + q, x' + x'') &= \mu(p, x')\mu(q, x''), \\
 \mu(p \boxplus_{\pi} q, x) &= \pi\mu(p, x) + (1 - \pi)\mu(q, x), \\
 \mu(\sigma_{\text{rel}}(p), \sigma_{\text{rel}}(x)) &= \mu(p, x), \\
 \mu(\nu_{\text{rel}}(p), \nu_{\text{rel}}(x)) &= \mu(p, x), \\
 \mu(p, x) &= 0 \text{ otherwise.}
 \end{array}$$

$\underline{a} \rightsquigarrow \check{\underline{a}}$	$\underline{\delta} \rightsquigarrow \check{\underline{\delta}}$	$\frac{p \rightsquigarrow x}{p \cdot q \rightsquigarrow x \cdot q}$	
$\frac{p \rightsquigarrow x, q \rightsquigarrow y}{p + q \rightsquigarrow x + y}$	$\frac{p \rightsquigarrow x}{p \vdash_{\pi} q \rightsquigarrow x, q \vdash_{\pi} p \rightsquigarrow x}$	$\frac{p \rightsquigarrow x}{\sigma_{rel}(p) \rightsquigarrow \sigma_{rel}(x), \nu_{rel}(p) \rightsquigarrow \nu_{rel}(x)}$	
$\frac{\check{\underline{a}} \xrightarrow{a} \checkmark}{\sigma_{rel}(x) \xrightarrow{\sigma} x}$	$\frac{x \xrightarrow{a} \checkmark}{x \cdot y \xrightarrow{a} y}$	$\frac{x \xrightarrow{a} \checkmark}{\nu_{rel}(x) \xrightarrow{a} \checkmark}$	$\frac{x \xrightarrow{a} \checkmark}{x + y \xrightarrow{a} \checkmark, y + x \xrightarrow{a} \checkmark}$
	$\frac{x \xrightarrow{a} p}{x \cdot y \xrightarrow{a} p \cdot y}$	$\frac{x \xrightarrow{a} p}{\nu_{rel}(x) \xrightarrow{a} p}$	$\frac{x \xrightarrow{a} p}{x + y \xrightarrow{a} p, y + x \xrightarrow{a} p}$
$\frac{x \not\rightsquigarrow}{\sigma_{rel}(x) \xrightarrow{\sigma} x}$	$\frac{x \xrightarrow{\sigma} x', y \not\rightsquigarrow}{x + y \xrightarrow{\sigma} x', y + x \xrightarrow{\sigma} x'}$	$\frac{x \xrightarrow{\sigma} x', y \xrightarrow{\sigma} y'}{x + y \xrightarrow{\sigma} x' + y'}$	$\frac{x \xrightarrow{\sigma} x'}{x \cdot y \xrightarrow{\sigma} x' \cdot y}$
$\mathbb{D}(\sigma_{rel}(x))$	$\frac{\mathbb{D}(x)}{\mathbb{D}(x \cdot y)}$	$\frac{\mathbb{D}(x)}{\mathbb{D}(x + y), \mathbb{D}(y + x)}$	$\frac{\mathbb{D}(x)}{\mathbb{D}(x \vdash_{\pi} y), \mathbb{D}(y \vdash_{\pi} x)}$

Table 5. Deduction rules of $\text{prBPA}_{\text{drt}}^-$.

$a \rightsquigarrow \check{a}$	$\delta \rightsquigarrow \check{\delta}$	$\check{a} \xrightarrow{a} \checkmark$	$\check{a} \xrightarrow{\sigma} \check{a}$	$\check{\delta} \xrightarrow{\sigma} \check{\delta}$	$\mathbb{D}(a)$	$\mathbb{D}(\check{a})$	$\mathbb{D}(\delta)$	$\mathbb{D}(\check{\delta})$

Table 6. Deduction rules for delayable actions.

The probability distribution function, $\mu(p, x)$, gives the total probability with which p behaves as x . For example, using the definition we obtain that $\mu(a \vdash_{0.5} a, \check{a}) = 0.5\mu(a, \check{a}) + 0.5\mu(a, \check{a}) = 1$. We find it easier to calculate the total probability by such a function, than to make it a part of the deduction rules ([21,16]). Because in the construction of the term model we use the Larsen-Skou probabilistic bisimulation relation ([21]), we need to extend the probability distribution function to the power set of $\mathcal{PR}$. Thus, $\mu(p, M) = \sum_{x \in M} \mu(p, x)$ for each $M \subseteq \mathcal{PR}$, $M \neq \emptyset$, and $\mu(p, \emptyset) = 0$ for each $p \in \mathcal{PR}$.

Definition 3. Let R be an equivalence relation on the set of processes $\mathcal{PR}$. R is a probabilistic bisimulation if:

1. If pRq and $p \rightsquigarrow s$ then there is a term t such that $q \rightsquigarrow t$ and sRt ;
2. If sRt and $s \xrightarrow{\sigma} p$, then there is a term q such that $t \xrightarrow{\sigma} q$ and pRq ;
3. If sRt and $s \xrightarrow{a} p$, then there is a term q such that $t \xrightarrow{a} q$ and pRq ;
4. If sRt and $s \xrightarrow{a} \checkmark$, then $t \xrightarrow{a} \checkmark$;
5. If sRt and $\mathbb{D}(s)$, then $\mathbb{D}(t)$;
6. If pRq , then $\mu(p, M) = \mu(q, M)$ for each $M \in \mathcal{PR}/R$.

We say that p is probabilistically bisimilar to q , denoted $p \rightleftharpoons q$, if there is a probabilistic bisimulation R such that pRq .

Theorem 4 ((Congruence)). $\Leftrightarrow$ is a congruence relation on $\text{prBPA}_{\text{drt}}^+$. $\square$

In [3] we prove that the axioms of $\text{prBPA}_{\text{drt}}^+$ are sound for $\Leftrightarrow$. An axiom is sound if there is a bisimulation relation that matches the left and the right sides of the axiom or the left and the right sides of the conclusion if the axiom is conditional. In order to prove the soundness of the axioms for delayable actions (DA1 – DA5) some technicalities are needed and they are presented by the following lemmas.

Lemma 5. If $z \in \mathcal{SP}$ and $z \Leftrightarrow z + z$, $z \rightsquigarrow x$ and $z \rightsquigarrow y$, then $x \Leftrightarrow y$. $\square$

Lemma 6. Let y be an $\mathcal{SP}$ process and $y \Leftrightarrow \nu_{\text{rel}}(z) + \sigma_{\text{rel}}(y)$ for some z such that $z \Leftrightarrow z + z$. If $y \rightsquigarrow x$ and $y \rightsquigarrow y$, then $x \Leftrightarrow y$. $\square$

Theorem 7 ((Soundness)). If $p, q \in \mathcal{SP}$ and $\text{prBPA}_{\text{drt}}^+ \vdash x = y$, then $x \Leftrightarrow y$. $\square$

In [3] completeness of $\text{prBPA}_{\text{drt}}^+$ for the presented operational semantics is proved. Basically, the proof follows the line of the completeness proofs in [10] and [23] except in the parts where probabilities play a prominent role. In the proof of these parts the induction on the structure of basic terms is used as well as the Cancellation theorem given below.

Theorem 8 ((Cancellation)). If p, q and r are $\mathcal{PR}$ terms and $\pi \in \langle 0, 1 \rangle$ such that $p \dot{+}_{\pi} q \Leftrightarrow p \dot{+}_{\pi} r$, then $q \Leftrightarrow r$. $\square$

Theorem 9 ((Completeness)). If p and q are closed $\text{prBPA}_{\text{drt}}^+$ terms, then $p \Leftrightarrow q \Rightarrow \text{prBPA}_{\text{drt}}^+ \vdash p = q$. $\square$

3 Extension with Merge and Communication

Basic process algebra cannot be used to design concurrent systems. For that reason in this section we present an extension of the basic process algebra from Section 2.2. with a notion of parallel composition and communication. The obtained extension is called the Probabilistic algebra of communicating processes with discrete relative time, in short denoted by $\text{ACP}_{\pi, \text{drt}}^+$.

Axiom System The signature $\Sigma_{\text{prC}+}$ consists of the operators from $\text{prBPA}_{\text{drt}}^+$ and operators: $\parallel$ (merge), $\ll$ (left merge), $|$ (communication merge) and encapsulation ∂_H with $H \subseteq A$, which are standard operators from ACP, $\mathbb{I}$ (merge with memory), which is also an operator in ACP_{π}^+ ([2]) and a new operator $\overline{\sigma}$ (renormalization operator).

In the interleaving model which is essential to ACP-like process algebras, parallel composition is modeled by non-deterministic choice, in the sense that

the choice of the process which executes the next action is considered to be non-deterministic. Thus, if two probabilistic processes are run in parallel, such as in the process $P \equiv (p \dot{+}_\pi q) \parallel (s \dot{+}_\rho t)$ (for simplicity we assume that they do not communicate and that p, q, s and t are trivial probabilistic processes), each of them can be chosen non-deterministically to execute the next action. If for example, the process $p \dot{+}_\pi q$ executes the next action, then the probabilistic choice between p and q is already resolved. The merge with memory operator postpones the non-deterministic choice between the left and the right process in the parallel composition till the probabilistic choice between p and q becomes resolved. Thus, process P continues like $p \cdot (s \dot{+}_\rho t)$ with probability π and like $q \cdot (s \dot{+}_\rho t)$ with probability $1 - \pi$. We emphasize that the probabilistic choice between s and t is not necessarily resolved. This is the main difference in our approach with the already mentioned work in [16]. Because in the derivation process $s \dot{+}_\rho t$ gets lost but it has to come after p or q the $\parallel$ operator memorizes (in the fourth argument) $s \dot{+}_\rho t$. The role of the renormalization operator $\overline{\sigma}$ becomes clear from axiom $PM4$. For example, let us consider the following parallel composition expressed by the $\parallel$ operator: $Q \equiv (\underline{a} + \sigma_{rel}(\underline{b}), \underline{a} + \sigma_{rel}(\underline{b})) \parallel (\sigma_{rel}(\underline{d}), \underline{c} \dot{+}_{0.5} \sigma_{rel}(\underline{d}))$ and let us assume that the first process performs the next action. Then, if $\underline{a}$ is the outcome of the non-deterministic choice, then after execution of a process Q continues like $\underline{c} \dot{+}_{0.5} \sigma_{rel}(\underline{d})$ due to axioms $PM4$ and $PM5$. But if the outcome of the non-deterministic choice is $\sigma_{rel}(\underline{b})$, then process Q idles one time unit and then action b occurs and Q continues like the part of $\underline{c} \dot{+}_{0.5} \sigma_{rel}(\underline{d})$ which is time consistent. That is, action c cannot occur after the time tick (it would cause a time inconsistency) and thus, only action d occurs with renormalized probability 1. Axioms $PR1 - 5$ show that this operator “filters” the part of a process which idles for one time unit and at the same time it does renormalization of the probabilities.

$ACP_{\pi, \text{drt}}^+$ is parametrized by a communication function $\gamma : A_\delta \times A_\delta \rightarrow A_\delta$ ([9]). The axiom for the operators are given in Table 7, 8 and 9 with $a, b \in A_\delta$ and $\pi \in \langle 0, 1 \rangle$.

Theorem 10 ((Elimination theorem)). *If p is a closed $ACP_{\pi, \text{drt}}^+$ term, then there is a closed $\text{prBPA}_{\text{drt}}^+$ term q such that $ACP_{\pi, \text{drt}}^+ \vdash p = q$.* $\square$

Structured Operational Semantics The term model of $ACP_{\pi, \text{drt}}^+$ is an extension of the term model of $\text{prBPA}_{\text{drt}}^+$, that is, it is obtained by an extension of the signature, the set of deduction rules and the probability distribution function given in Section 2.2. We consider the signature $\Sigma_{\text{prC}_{os}^+}$ which contains the constants and the operators from both $\Sigma_{\text{prB}_{os}^+}$ and Σ_{prC^+} .

The deduction rules for the new operators and the $\mathbb{D}$ predicate in $ACP_{\pi, \text{drt}}^+$ are given in Table 10 where $a, b, c \in A$ and $H \subseteq A$. The probability distribution function for the term model of $ACP_{\pi, \text{drt}}^+$ is defined by Definition 2 and Definition 11. The probabilistic bisimulation relation is defined by Definition 3.

The operator $\overline{\sigma}$ and the $\mathbb{D}$ predicate are used in the deduction rules for σ -transition of the $\parallel$ and the $\underline{\parallel}$ operators. If x can do a σ -transition to x' but p

$\underline{a} \mid \underline{b}$	$= \underline{\gamma(a, b)}$	<i>CF</i>	$\overline{\sigma}(\nu_{rel}(x))$	$= \underline{\delta}$	<i>PR1</i>
			$\overline{\sigma}(\nu_{rel}(x) + \sigma_{rel}(y))$	$= y$	<i>PR2</i>
			$\overline{\sigma}(\nu_{rel}(x) \sqcup_{\pi} y)$	$= \overline{\sigma}(y)$	<i>PR3</i>
$\underline{a} \parallel x$	$= \underline{a} \cdot x$	<i>CM2</i>	$\overline{\sigma}((\nu_{rel}(x) + \sigma_{rel}(y)) \sqcup_{\pi} z)$	$= \overline{\sigma}(\sigma_{rel}(y) \sqcup_{\pi} z)$	<i>PR4</i>
$\underline{a} \cdot x \parallel y$	$= \underline{a} \cdot (x \parallel y)$	<i>CM3</i>	$\overline{\sigma}(\sigma_{rel}(x))$	$= x$	<i>PR5</i>
$(x + y) \parallel z$	$= x \parallel z + y \parallel z$	<i>CM4</i>	$\partial_H(\underline{a})$	$= \underline{a}$	$a \notin H$ <i>D1</i>
$(x \sqcup_{\pi} y) \parallel z$	$= x \parallel z \sqcup_{\pi} y \parallel z$	<i>PCM1</i>	$\partial_H(\underline{a})$	$= \underline{\delta}$	$a \in H$ <i>D2</i>
$\underline{a} \mid \underline{b} \cdot x$	$= (\underline{a} \mid \underline{b}) \cdot x$	<i>CM5</i>	$\partial_H(\sigma_{rel}(x))$	$= \sigma_{rel}(\partial_H(x))$	<i>D3</i>
$\underline{a} \cdot x \mid \underline{b}$	$= (\underline{a} \mid \underline{b}) \cdot x$	<i>CM6</i>	$\partial_H(x + y)$	$= \partial_H(x) + \partial_H(y)$	<i>D4</i>
$\underline{a} \cdot x \mid \underline{b} \cdot y$	$= (\underline{a} \mid \underline{b}) \cdot (x \parallel y)$	<i>CM7</i>	$\partial_H(x \cdot y)$	$= \partial_H(x) \cdot \partial_H(y)$	<i>D5</i>
$\sigma_{rel}(x) \mid \nu_{rel}(y)$	$= \underline{\delta}$	<i>CM8</i>	$\partial_H(x \sqcup_{\pi} y)$	$= \partial_H(x) \sqcup_{\pi} \partial_H(y)$	<i>PD6</i>
$\nu_{rel}(x) \mid \sigma_{rel}(y)$	$= \underline{\delta}$	<i>CM9</i>	$x \parallel y$	$= (x, x) \parallel (y, y)$	<i>PM1</i>
$\sigma_{rel}(x) \mid \sigma_{rel}(y)$	$= \sigma_{rel}(x \mid y)$	<i>CM10</i>	$(x \sqcup_{\pi} u, z) \parallel (y, w)$	$= (x, z) \parallel (y, w) \sqcup_{\pi} (u, z) \parallel (y, w)$	<i>PM2</i>
$(x \sqcup_{\pi} y) \mid z$	$= x \mid z \sqcup_{\pi} y \mid z$	<i>PCM6</i>	$(x, z) \parallel (y \sqcup_{\pi} v, w)$	$= (x, z) \parallel (y, w) \sqcup_{\pi} (x, z) \parallel (v, w)$	<i>PM3</i>
$x \mid (y \sqcup_{\pi} z)$	$= x \mid y \sqcup_{\pi} x \mid z$	<i>PCM7</i>			

Table 7. Additional axioms for $ACP_{\pi, drt}^+$ - I.

$\sigma_{rel}(x) \parallel \nu_{rel}(y)$	$= \underline{\delta}$	<i>A18</i>	$z = z + z \Rightarrow$
$\sigma_{rel}(x) \parallel (\nu_{rel}(y) + \sigma_{rel}(z))$	$= \sigma_{rel}(x \parallel z)$	<i>A19</i>	$(x + y) \mid z = x \mid z + y \mid z$ <i>CM11</i>
$\sigma_{rel}(x) \parallel (\nu_{rel}(y) \sqcup_{\pi} z)$	$= \sigma_{rel}(x) \parallel z$	<i>A20</i>	$z = z + z \Rightarrow$
$\sigma_{rel}(x) \parallel ((\nu_{rel}(y) + \sigma_{rel}(z)) \sqcup_{\pi} w)$	$= \sigma_{rel}(x) \parallel (\sigma_{rel}(z) \sqcup_{\pi} w)$	<i>A21</i>	$z \mid (x + y) = z \mid x + z \mid y$ <i>CM12</i>
$\sigma_{rel}(x) \parallel \sigma_{rel}(y)$	$= \sigma_{rel}(x \parallel y)$	<i>A22</i>	

Table 8. Additional axioms for $ACP_{\pi, drt}^+$ - II.

cannot, then neither does $x \parallel p$. But if p can do a σ -transition and there is a sub-process of p which cannot idle one time unit, then in the next time unit process x' is followed by process $\overline{\sigma}(p)$, the sub-process of p which can idle one time unit (see also Lemma 13). Moreover, in the definition of the probability distribution function the value $\mu(\overline{\sigma}(p), y)$ is renormalized by a normalizing factor which depends on the probability that p behaves as a “now” process (a process that with 0 probability performs a σ -transition). The normalizing factor is defined by an auxiliary function ν .

We need again to distinguish probabilistic from non-deterministic processes. Probabilistic processes are processes that can perform only probabilistic transitions and non-deterministic processes are processes that can perform an action transition or a σ -transition. $\mathcal{SP}$ is the set of all probabilistic processes, $\mathcal{DP}$ is the set of all non-deterministic processes and $\mathcal{PR} = \mathcal{SP} \cup \mathcal{DP}$. It is obvious that $\mathcal{PR} \subset \mathcal{PR}$, $\mathcal{SP} \subset \mathcal{SP}$ and $\mathcal{DP} \subset \mathcal{DP}$.

Definition 11. Let $p \in \mathcal{SP}$ and $P = \{u : u \in \mathcal{DP} \ \& \ p \rightsquigarrow u\}$. The normalizing factor is a function: $\nu : \mathcal{SP} \rightarrow [0, 1]$ where $\nu(p) = \mu(p, \{u : u \in P \ \& \ \neg \mathbb{D}(u)\})$.

$(\nu_{rel}(x) + \sigma_{rel}(u), z) \parallel (\nu_{rel}(y) + \sigma_{rel}(v), w) = (\nu_{rel}(x), z) \parallel (\nu_{rel}(y), w) + \sigma_{rel}((u, \bar{\sigma}(z)) \parallel (v, \bar{\sigma}(w)))$	<i>PM4</i>
$x = x + x, y = y + y \Rightarrow (\nu_{rel}(x), z) \parallel (\nu_{rel}(y), w) = \nu_{rel}(x) \parallel w + \nu_{rel}(y) \parallel z + \nu_{rel}(x) \mid \nu_{rel}(y)$	<i>PM5</i>
$x = x + x, u = u + u, y = y + y \Rightarrow (\nu_{rel}(x) + \sigma_{rel}(u), z) \parallel (\nu_{rel}(y), w) = (\nu_{rel}(x), z) \parallel (\nu_{rel}(y), w)$	<i>PM6</i>
$x = x + x, y = y + y, v = v + v \Rightarrow (\nu_{rel}(x), z) \parallel (\nu_{rel}(y) + \sigma_{rel}(v), w) = (\nu_{rel}(x), z) \parallel (\nu_{rel}(y), w)$	<i>PM7</i>

Table 9. New axioms for the merge with memory operator.

$\frac{p \rightsquigarrow x}{p \parallel q \rightsquigarrow x \parallel q}$	$\frac{p \rightsquigarrow x, q \rightsquigarrow y}{p \mid q \rightsquigarrow x \mid y}$	$\frac{p \rightsquigarrow x}{\partial_H(p) \rightsquigarrow \partial_H(x)}$
$\frac{p \rightsquigarrow x, q \rightsquigarrow y}{p \parallel q \rightsquigarrow (x, p) \parallel (y, q), (p, z) \parallel (q, w) \rightsquigarrow (x, z) \parallel (y, w)}$	$\frac{p \rightsquigarrow x, x \xrightarrow{\sigma} y}{\bar{\sigma}(p) \rightsquigarrow y}$	$\frac{\neg \mathbb{D}(p)}{\bar{\sigma}(p) \rightsquigarrow \check{\delta}}$
$\frac{x \xrightarrow{a} p}{(x, z) \parallel (y, w) \xrightarrow{a} p \parallel w, (y, w) \parallel (x, z) \xrightarrow{a} w \parallel p}$	$\frac{x \xrightarrow{a} \checkmark}{(x, z) \parallel (y, w) \xrightarrow{a} w, (y, w) \parallel (x, z) \xrightarrow{a} w}$	
$\frac{x \xrightarrow{a} p, y \xrightarrow{b} q, \gamma(a, b) = c}{(x, z) \parallel (y, w) \xrightarrow{c} p \parallel q, x \mid y \xrightarrow{c} p \parallel q}$	$\frac{x \xrightarrow{a} \checkmark, y \xrightarrow{b} q, \gamma(a, b) = c}{(x, z) \parallel (y, w) \xrightarrow{c} q, x \mid y \xrightarrow{c} q}$	$\frac{x \xrightarrow{a} p}{x \parallel y \xrightarrow{a} p \parallel y}$
$\frac{x \xrightarrow{a} p, y \xrightarrow{b} \checkmark, \gamma(a, b) = c}{(x, z) \parallel (y, w) \xrightarrow{c} p, x \mid y \xrightarrow{c} p}$	$\frac{x \xrightarrow{a} \checkmark, y \xrightarrow{b} \checkmark, \gamma(a, b) = c}{(x, z) \parallel (y, w) \xrightarrow{c} \checkmark, x \mid y \xrightarrow{c} \checkmark}$	$\frac{x \xrightarrow{a} \checkmark}{x \parallel y \xrightarrow{a} y}$
$\frac{x \xrightarrow{a} p, a \notin H}{\partial_H(x) \xrightarrow{a} \partial_H(p)}$	$\frac{x \xrightarrow{a} \checkmark, a \notin H}{\partial_H(x) \xrightarrow{a} \checkmark}$	
$\frac{x \xrightarrow{\sigma} u, y \xrightarrow{\sigma} v}{(x, p) \parallel (y, q) \xrightarrow{\sigma} (u, \bar{\sigma}(p)) \parallel (v, \bar{\sigma}(q))}$	$\frac{x \xrightarrow{\sigma} x', \mathbb{D}(p)}{x \parallel p \xrightarrow{\sigma} x' \parallel \bar{\sigma}(p)}$	$\frac{x \xrightarrow{\sigma} x', y \xrightarrow{\sigma} y'}{x \mid y \xrightarrow{\sigma} x' \mid y'}$
$\frac{\mathbb{D}(x), \mathbb{D}(y)}{\mathbb{D}(x \parallel y), \mathbb{D}((x, z) \parallel (y, w))}$	$\frac{\mathbb{D}(x), \mathbb{D}(y)}{\mathbb{D}(x \parallel y), \mathbb{D}(x \mid y)}$	$\frac{\mathbb{D}(x)}{\mathbb{D}(\partial_H(x))}$
		$\frac{p \rightsquigarrow x, x \xrightarrow{\sigma} y, \mathbb{D}(y)}{\mathbb{D}(\bar{\sigma}(p))}$

Table 10. Operational semantics of $\text{ACP}_{\pi, \text{drt}}^+$.

The probability distribution function $\mu : \text{PR} \times \text{PR} \rightarrow [0, 1]$ is defined with the equalities given in Definition 2 and the following:

$$\begin{aligned}
\mu(p \parallel q, (x, p) \parallel (y, q)) &= \mu(p, x) \mu(q, y), \\
\mu(p \parallel q, x \parallel y) &= \mu(p, x), \\
\mu(p \mid q, x \mid y) &= \mu(p, x) \mu(q, y), \\
\mu(\partial_H(p), \partial_H(x)) &= \mu(p, x), \\
\mu((p, z) \parallel (q, w), (x, z) \parallel (y, w)) &= \mu(p, x) \mu(q, y), \\
\mu(\bar{\sigma}(p), \check{\delta}) &= 1, & \text{if } \neg \mathbb{D}(p) \\
\mu(\bar{\sigma}(p), y) &= \frac{1}{1 - \nu(p)} \mu(p, \{x : p \rightsquigarrow x \ \& \ x \xrightarrow{\sigma} y\}) & \text{if } \mathbb{D}(p)
\end{aligned}$$

In [3] the complete proofs of the following properties can be found.

Theorem 12 ((Congruence)). $\Leftrightarrow$ is a congruence relation on $\text{ACP}_{\pi, \text{drt}}^+$. $\square$

Lemma 13. If $p, q, r \in \text{SP}$ and $\pi \in \langle 0, 1 \rangle$ then: $\overline{\sigma}(\nu_{\text{rel}}(q) + \sigma_{\text{rel}}(r)) \Leftrightarrow r$, $\overline{\sigma}(\nu_{\text{rel}}(q) \uplus_{\pi} r) \Leftrightarrow \overline{\sigma}(r)$ and $\overline{\sigma}((\nu_{\text{rel}}(p) + \sigma_{\text{rel}}(q)) \uplus_{\pi} r) \Leftrightarrow \overline{\sigma}(\sigma_{\text{rel}}(q) \uplus_{\pi} r)$. $\square$

Theorem 14 ((Soundness)). If $p, q \in \text{SP}$ and $\text{ACP}_{\pi, \text{drt}}^+ \vdash p = q$, then $p \Leftrightarrow q$. $\square$

Theorem 15 ((Completeness)). If p and q are $\text{ACP}_{\pi, \text{drt}}^+$ closed terms and $p \Leftrightarrow q$, then $\text{ACP}_{\pi, \text{drt}}^+ \vdash p = q$. $\square$

4 PAR Protocol

In this section we consider the PAR protocol (Positive Acknowledgement with Retransmission protocol) as it is described in [22]. The same protocol is specified in the untimed probabilistic process algebra in [2]. But since in that process algebra time aspects of the protocol can not be expressed in a quantitative manner a timer process and the priority operator are used in the specification and the verification of the protocol. With the ability to express quantitatively the time characteristics of the protocol, the priority operator is not needed for the verification in $\text{ACP}_{\pi, \text{drt}}^+$.

Next we give a brief description of the PAR protocol (see [22,2] for more details). The PAR protocol is a relatively simple one-way protocol. It is a data-link level protocol intended to be used over unreliable transmission channels which may corrupt or lose data. The protocol is modeled as four processes, one sender process S , one receiver R and two communication channels K and L . The sender receives data from the upper level at port 1 and sends them to the receiver via a communication channel K through port 3. After that it waits for an acknowledgement from the receiver R (at port 5) before a new datum is transmitted. If an acknowledgement does not occur within some period of time T (time-out period), the sender resends the old datum. The receiver receives data from the channel K at port 4 and if the data are undamaged it delivers them to the upper level (at port 2) and sends an acknowledgement to the sender through the channel L (at port 6). A control bit is used in order to avoid multiple writing of a message at the output port. Of crucial importance here is the duration of the time-out period, which should be longer than the sum of the delays through the channels (d_K and d_L) and message processing time by the receiver (d_R).

Let D be a finite set of data. The set of atomic actions A contains read, send and communication actions and k and l actions which present loss of a message and acknowledgement, respectively. We use the standard read/send communication function given by $r_k(x) \mid s_k(x) = c_k(x)$ for communication port k and message x . Unreliability of the channel K is specified by the probabilistic choice operator: correct transmission of a message with probability π , corruption of a message with probability α and loss of a message with probability $1 - \pi - \alpha$ (in

a similar way this is specified for the channel L). The specifications of the four processes (based on the one in [12]) are given by the recursive equations:

$$\begin{aligned}
\text{Sender : } S &= S_0 \\
S_b &= \sum_{f \in D} r_1(f) \cdot S_b^f & (b = 0, 1, f \in D) \\
S_b^f &= \underline{\underline{s_3(fb)}} \cdot WS_b^f \\
WS_b^f &= \sum_{t=0}^T \sigma_{rel}^t(\underline{\underline{r_5(ack)}} \cdot \sigma_{rel}(S_{1-b})) + \sigma_{rel}^T(S_b^f) + \sum_{t=0}^T \sigma_{rel}^t(\underline{\underline{r_5(\perp)}} \cdot \sigma_{rel}(S_b^f)) \\
\text{Receiver : } R &= R_0 \\
R_b &= \sum_{f \in D} r_4(fb) \cdot SH_b^f + \sum_{f \in D} r_4(f(1-b)) \cdot SR_b + r_4(\perp) \cdot R_b \\
SH_b^f &= \underline{\underline{s_2(f)}} \cdot \sigma_{rel}^{d_R}(SR_{1-b}) \\
SR_b &= \underline{\underline{s_6(ack)}} \cdot \sigma_{rel}(R_b)
\end{aligned}$$

Channels :

$$\begin{aligned}
K &= \sum_{f \in D, b \in \{0,1\}} r_3(fb) \cdot \sigma_{rel}^{d_K}(K_f) & L &= r_6(ack) \cdot \sigma_{rel}^{d_L}(L_a) \\
K_f &= (\underline{\underline{s_4(fb)}} \uplus_{\pi} \underline{\underline{s_4(\perp)}} \uplus_{\pi} \underline{\underline{k}}) \cdot \sigma_{rel}(K) & L_a &= (\underline{\underline{s_5(ack)}} \uplus_{\eta} \underline{\underline{s_5(\perp)}} \uplus_{\zeta} \underline{\underline{l}}) \cdot \sigma_{rel}(L)
\end{aligned}$$

The protocol behaviour is obtained by the composition of the four processes:

$$PAR = t_I \circ \partial_H(S \parallel K \parallel L \parallel R),$$

where $H = \{r_i(x), s_i(x) | i \in \{3, 4, 5, 6, 7\}, x \in (D \times \{0, 1\}) \cup \{ack, \perp\}\}$ is the set of encapsulated atomic actions and t_I is the pre-abstraction operator, that renames all internal actions from the set

$I = \{c_i(x) | i \in \{3, 4, 5, 6, 7\}, x \in (D \times \{0, 1\}) \cup \{ack, \perp\}\} \cup \{k, l\}$ into t . In $ACP_{\pi, drt}^+$ we derive the following recursive specification for process $PAR = P_0$:

$$\begin{aligned}
P_0 &= \sum_{f \in D} r_1(f) \cdot P_1 \\
P_1 &= \underline{\underline{t}} \cdot \sigma_{rel}^{d_K}(P_2) & P_5 &= \underline{\underline{t}} \cdot \sigma_{rel}^{d_L}(P_6) \\
P_2 &= \underline{\underline{t}} \cdot P_3 \uplus_{\pi} \underline{\underline{t}} \cdot P_4 & P_6 &= \underline{\underline{t}} \cdot \sigma_{rel}(P) \uplus_{\eta} \underline{\underline{t}} \cdot \underline{\underline{t}} \cdot \sigma_{rel}^{d_K}(P_7) \uplus_{\zeta} \underline{\underline{t}} \cdot \sigma_{rel}^{T-d_K-d_R-d_L}(P_8) \\
P_3 &= \underline{\underline{t}} \cdot \underline{\underline{s_2(f)}} \cdot \sigma_{rel}^{d_R}(P_5) & P_7 &= \underline{\underline{t}} \cdot \underline{\underline{t}} \cdot \sigma_{rel}^{d_L}(P_6) \uplus_{\pi} \underline{\underline{t}} \cdot \sigma_{rel}^{T-d_K}(P_8) \\
P_4 &= \sigma_{rel}^{T-d_K}(P_1) & P_8 &= \underline{\underline{t}} \cdot \sigma_{rel}^{d_K}(P_7)
\end{aligned}$$

From the specification the reader can note that after reading a datum from the environment no other non-deterministic choice occurs, that is, the internal behaviour of the protocol is pure probabilistic. Thus, if we abstract from the content of the messages sent from the environment this specification can be considered as a Markov chain. Further using Markov chain analysis we can prove liveness of the protocol by showing that state P_0 is a recurrent state, the probability to re-enter P_0 after finite number of time steps is 1. In terms of the protocol this means that a datum received by the environment at port 1 is delivered within finite period of time at port 2 after which the protocol is ready to receive a new datum.

5 Conclusion and Further Work

In this paper we presented an ACP-like process algebra suitable for specifying probabilistic timed concurrent systems. In our previous work we have extended ACP process algebra with probabilities in a manner that non-determinism is considered as one of the operators as well. In particular the non-determinism is used in formalization of the interleaving approach to parallel composition. This process algebra can be used to derive a process which can be considered as a Markov chain. As a result the question can be answered: “How many times in average a message should be resent from a sender to a receiver to get received if the communication goes through an unreliable channel with given performances?” But still this says nothing about the probability (or any bounds) that a message is received within 5 seconds. Our research shows that the process algebra proposed in the paper is a good starting point to build a method that answers such questions algebraically.

The example of the PAR protocol shows that this formalism is also suitable for verification of some simple concurrent systems. Since non-determinism has been resolved during the derivation, we have obtained a process which can be viewed as a Markov chain. Still in more complex systems, due to the presence of non-determinism, in order to do a performance analysis of the protocol specification we have to find a way to resolve it. It turns out that $ACP_{\pi, \text{drt}}^+$ can successfully be extended by operators and principles, as the priority operator, the abstraction operator and fairness principles, that can cope with this problem. Here we are not talking about rules with a very complex axiomatization, but ones that can be brought out by a slight modification of their counterparts in ACP. Some details in the formalization of these operators and rules are still open questions. It is one of our main goal in the further research.

Acknowledgments. Here I would like to express my gratitude to Jos Baeten and Kees Middelburg without whose help these results would not have been obtained.

References

1. S. Andova, *Process algebra with interleaving probabilistic parallel composition*, CSR 99-04, Eindhoven University of Technology, 1999.
2. S. Andova, *Process algebra with probabilistic choice (extended abstract)*, Proc. ARTS'99, Bamberg, Germany, J.-P. Katoen, ed., LNCS 1601, Springer-Verlag, pp. 111-129, 1999. Full version report CSR 99-12, Eindhoven University of Technology, 1999.
3. S. Andova, *Discrete relative time in probabilistic process algebra*, Eindhoven University of Technology, CSR 00-**, 2000.
4. J.C.M. Baeten, J.A. Bergstra, *Real time process algebra*, Formal Aspects of Computing, 3(2):142-188, 1991.
5. J.C.M. Baeten, J.A. Bergstra, *Process algebra with partial choice*, Proc. CONCUR'94, Uppsala, B. Jonsson & J. Parrow, eds., LNCS 836, Springer Verlag, 465-480, 1994.
6. J.C.M. Baeten, J.A. Bergstra, *Discrete time process algebra*, Formal Aspects of Computing, 8(2):188-208, 1996.
7. J.C.M. Baeten, J. A. Bergstra, *Discrete time process algebra: Absolute time, relative time and parametric time*, Fundamenta Informaticæ, 29(1,2):51-76, 1997.

8. J.C.M. Baeten, C. A. Middelburg, *Process algebra with timing: Real time and discrete time*, Eindhoven University of Technology, CSR 99-11, 1999. To appear in J.A. Bergstra, A. Ponse and S.A. Smolka, ed., *Handbook of Process Algebra*, Elsevier, 2000.
9. J.C.M. Baeten, W.P. Weijland, *Process algebra*, Cambridge University Press, 1990.
10. J.C.M. Baeten, C. Verhoef, *Concrete process algebra*, *Handbook of Logic in Computer Science*, volume 4: "Semantic Modelling", Oxford University Press, 1995.
11. J.C.M. Baeten, J.A. Bergstra, S.A. Smolka, *Axiomatizing probabilistic processes: ACP with generative probabilities*, *Information and Computation* 121(2): 234-255, September 1995.
12. D. Bošnački, *Discrete time specification of the PAR protocol*, (unpublished paper).
13. P.R. D'Argenio, H. Hermanns, J.-P. Katoen *On generative parallel composition*, Preliminary Proc. of PROBMIV'98, Indianapolis, USA, C. Baier & M. Huth & M. Kwiatkowska & M. Ryan ed., 105-121, 1998.
14. A. Giacalone, C.-C. Jou, S. A. Smolka, *Algebraic reasoning for probabilistic concurrent systems*, Proc. Working Conference on Programming Concepts and Methods, IFIP TC 2, Sea of Galilee, Israel, M. Broy & C.B. Jones ed., 443-458, 1990.
15. R. J. van Glabbeek, S. A. Smolka, B. Steffen, C. M. N. Tofts, *Reactive, generative and stratified models of probabilistic processes*, Proc. of 5th Annual IEEE Symp. on Logic in Computer Science, Philadelphia, PA, 130-141, 1990.
16. H. Hansson, *Time and probability in formal design of distributed systems*, Ph.D. thesis, DoCS 91/27, University of Uppsala, 1991.
17. H. Hennessy, T. Regan, *A process algebra for timed systems*, *Information and Computation*, 117(2):221-239, 1995.
18. A. Jeffrey, *Discrete Timed CSP*, Tech. Report PMG Memo 78, Chalmers University of Technology, Dep. of Computer Sciences, 1991.
19. B. Jonsson, K.G. Larsen, *Specification and refinement of probabilistic processes*, Proc. of 6th Annual IEEE Symp. on Logic in Computer Science, Amsterdam, 1991.
20. C.-C. Jou, S. A. Smolka *Equivalences, congruences and complete axiomatizations for probabilistic processes*, Proc. CONCUR '90, LNCS 458, Springer Verlag, Berlin, 367-383, 1990.
21. K.G. Larsen, A. Skou, *Bisimulation through probabilistic testing*, Proc. of 16th ACM Symp. on Principles of Programming Languages, Austin, TX, 1989.
22. F.W. Vaandrager, *Two simple protocols*, In: *Applications of Process Algebra*, Cambridge University Press, J.C.M. Baeten ed., pp.23-44, 1990.
23. J. J. Vereijken, *Discrete-time process algebra*, Ph.D. thesis, IPA 97/06, Eindhoven University of Technology, Computing Science Department, 1997.

A Modal Logic for KLAIM

Rocco De Nicola and Michele Loreti

Dipartimento di Sistemi e Informatica, Università di Firenze
{denicola,loreti}@dsi.unifi.it

Abstract. KLAIM is an experimental programming language that supports a programming paradigm where both processes and data can be moved across different computing environments. The language relies on the use of explicit localities, and on allocation environments that associate logical localities to physical sites. This paper presents a temporal logic for specifying properties of Klaim programs. The logic is inspired by Hennessy-Milner Logic (HML) and the ν -calculus, but has novel features that permit dealing with state properties to describe the effect of actions over the different sites. The logic is equipped with a consistent and complete proof system that enables one to prove properties of mobile systems.

Keywords: Mobile Code Languages, Temporal Logics of Programs, Coordination Models.

1 Introduction

The increasing use of wide area networks, especially the World Wide Web, is calling for new programming paradigms and for new programming languages that model interactions among clients and servers by means of *mobile agents*; these are programs that are transported and executed on different hosts. KLAIM (*a Kernel Language for Agents Interaction and Mobility*) [7] is one of such.

KLAIM consists of core Linda [3,4] with multiple located tuple spaces and of a set of process operators, borrowed from Milner's CCS [6]. The underlying communication model is based on shared data space and is, thus, asynchronous.

In KLAIM, tuple spaces and processes are distributed over different localities, which are considered as first-class data. The classical Linda operations, indexed with the locations of the tuple space they operate on, allow programmers to distribute/retrieve data and processes over/from different nodes directly. Thus, programmers can directly manage the physical distribution of processes, the allocation policies, and the agents' mobility.

For KLAIM's programs, like for other class of programs, it is crucial to establish correctness, deadlock freeness, liveness and to control access rights. Since the language is based on process algebras, a natural candidate for such tasks is a temporal logic based on HML, the logic proposed by Hennessy and Milner to specify and verify properties of CCS agents [5].

However, one soon realizes that HML would be insufficient. For achieving our task we need both state formulae (to test for the presence of specific tuples at given localities) and richer actions (to specify the performed actions and their source and target).

In this paper we shall introduce a variant of HML with recursion, the syntax of our logic is then the following:

$$\phi ::= \mathbf{tt} \mid t@ \sigma \mid \langle A \rangle \phi \mid \kappa \mid \nu \kappa. \phi \mid \phi \vee \phi \mid \neg \phi$$

where the state properties are specified by the basic operator $t@ \sigma$, and the classical indexed *diamond* operator ($\langle a \rangle$) is replaced by an action operator that contains sets of (abstract version of) the rich transition labels that are generated by the following grammar:

$$a ::= \mathcal{O}(s_1, t, s_2) \mid \mathcal{I}(s_1, t, s_2) \mid \mathcal{R}(s_1, t, s_2) \mid \mathcal{E}(s_1, P, s_2) \mid \mathcal{N}(s_1, -, s_2).$$

In the syntax above the label indicates source and destination of information movement (s_1 and s_2), the information transmitted (et and P) and the kind of movement ($\mathcal{O}, \mathcal{I}, \dots$).

Via abstract actions we can specify sets of labels that are characterized by common aspects, as source or destination of information movement, structure of the information transmitted and kind of movement.

We will show, via two simple examples, that the proposed logic is sufficiently expressive for describing interesting properties of mobile systems.

To support verification of such properties we will introduce also a proof system based on tableau. The proof system is inspired by [2], the additional difficulties and the novelties of our contribution are due to the fact that Cleaveland's system does not consider value passing and restricts attention to systems with a finite state space.

The rest of the paper is organized as follows. Section 2 contains the new labeled semantics for KLAIM. Section 3 contains syntax and semantics of the proposed logics together with its proofs system and a sketch of the soundness and completeness proof. Section 4 contains the KLAIM program for a distributed information system manager and the logical specification of some of its key properties. Section 5 shows a new equivalence that is in full agreement with the new one induced by the proposed logics.

Due to space limitation most of the proofs are omitted; they can be retrieved at [8]

2 KLAIM: Syntax and Semantics

KLAIM (Kernel Language for Agent Interaction and Mobility) is a language designed for programming applications over wide area networks. KLAIM is based on the notion of *locality* and relies on a Linda-like communication model.

Linda [1,3,4] is a coordination language with asynchronous communication and shared memory. Messages are structured data named *tuples*. The shared space is named *Tuple Space*. Tuples are accessed by pattern matching.

A KLAIM system is a set of *nodes* that we call *physical names* or *sites*. We use $\mathcal{S}$ to denote the set of *sites* and $s, s_1, s_2, \dots$ to denote its element.

Programs refer to sites using *localities*, or *logical name*. We use Loc to denote the set of localities and $l, l_1, l_2, \dots$ to denote its elements. We also assume existence of a locality $\mathbf{self} \in Loc$. We will use ℓ to denote elements of $\mathcal{S} \cup Loc$. The operations over tuple spaces take as argument the name of a node where the target tuple space resides and a tuple.

Every node has a computational component, a set of processes running in parallel, a tuple space and an associated environment ρ that binds localities to sites. We also assume that in the node s the environment ρ is such that $\rho(\mathbf{self}) = s$; i.e. the locality $\mathbf{self}$ refers to the node where a processes is running.

The set Net of KLAIM net is defined in Table 1. A node is defined by three parameters: the physical name s , the environment ρ and the process P . A net N can be obtained from the parallel composition of nodes.

$N ::= s ::_{\rho} P$	(node)
$\mid N_1 \parallel N_2$	(net composition)

Table 1. Nets syntax

For defining the syntax of processes, we introduce the following syntactic categories. We use Exp for the set expressions and Ψ for the set of parameterized *processes identifiers*, respectively ranged over by e and A . We use $VLoc$, Var and $VProc$ as the sets of locality, value and process variables, they are ranged over by u, x and X respectively. Moreover, $\tilde{u}$ will indicate sequences of locality variables and $\{\tilde{u}\}$ the set of locality variables in $\tilde{u}$. A similar notation we will also be used for other kinds of sequences.

$P ::= \mathbf{nil}$	(null process)
$\mid act.P$	(action prefixing)
$\mid \mathbf{out}(et)$	(evaluated tuple)
$\mid P_1 \mid P_2$	(parallel composition)
$\mid X$	(process variable)
$\mid A\langle\tilde{P}, \tilde{\ell}, \tilde{e}\rangle$	(process invocation)
$act ::= \mathbf{out}(t)@l \mid \mathbf{in}(t)@l \mid \mathbf{read}(t)@l \mid \mathbf{eval}(P)@l$	
$\mid \mathbf{newloc}(u)$	
$t ::= f \mid f, t$	
$f ::= e \mid P \mid \ell \mid !x \mid !X \mid !u$	

Table 2. Processes Syntax

Process syntax is defined in Table 2, where **nil** stands for the process that cannot perform any actions, $P_1|P_2$ stands for the parallel composition of P_1 and P_2 , and $act.P$ stands for the process that executes the action act then behaves like P . Also tuples are modeled as basic processes, then a tuple et is in the tuple space of a node s if and only if s contains a process **out**(et).

The possible actions are: **out**(t)@ ℓ , **in**(t)@ ℓ , **read**(t)@ ℓ , **eval**(P)@ ℓ and **newloc**(u). The first action adds the result of evaluation of t , using the allocation environment ρ of the node where the action is performed, inside the tuple space of the site $\rho(\ell)$ (if it exists). If t is a tuple and ρ an environment we define the evaluation of t within the environment ρ , $\mathcal{T}[\![t]\!]_\rho$, as in Table 3. The operation **out**(t)@ ℓ is nonblocking.

$\mathcal{T}[\![e]\!]_\rho = \mathcal{E}[\![e]\!]$	$\mathcal{T}[\![\ell]\!]_\rho = \rho(\ell)$	$\mathcal{T}[\![!x]\!]_\rho = !x$
$\mathcal{T}[\![P]\!]_\rho = P\{\rho\}$	$\mathcal{T}[\![!u]\!]_\rho = !u$	$\mathcal{T}[\![!X]\!]_\rho = !X$
	$\mathcal{T}[\![f, t]\!]_\rho = \mathcal{T}[\![f]\!]_\rho, \mathcal{T}[\![t]\!]_\rho$	

Table 3. Tuple Evaluation Function

To retrieve information from a tuple space located at ℓ one can use the **in**(t)@ ℓ and **read**(t)@ ℓ primitives, differently from **out** these are blocking operations (i.e. the computation is blocked until the required action can be performed).

$match(v, v)$	$match(P, P)$	$match(s, s)$
$match(!x, v)$	$match(!X, P)$	$match(!u, s)$
$match(et_2, et_1)$	$match(et_1, et_2)$	$match(et_3, et_4)$
$match(et_1, et_2)$	$match((et_1, et_3), (et_2, et_4))$	

Table 4. The Matching Rules

The **in**(t)@ ℓ action looks for a tuple inside the tuple space at ℓ that satisfies the *matching predicate* defined in Table 4. If this tuple et exists then it is removed from the tuple space and the continuation process P is closed with respect to the substitution $[et/t]$ that replaces every variable in a formal field of t with the corresponding value in et .

The **read** operation behaves like **in** but it doesn't remove the tuple. Actions **in**(t)@ $\ell.P$ and **read**(t)@ $\ell.P$ act as binders for variables in the formal fields of t . A variable is *free* if and only if it isn't bound. We said that a process P is closed if and only if each variable in P is not *free*. From now on we will take in account only closed processes.

The primitive **eval**(P)@ ℓ spawns a process P at the site ℓ . The localities in P are evaluated with the allocation environment of the destination node.

The action **newloc**(u) creates a new node and binds the variable u to its new/fresh name s . The continuation process is closed with respect to the substitution $\{s/u\}$. Prefix **newloc**(u). P binds the locality variable u in P . Programmers, by means of **newloc** operations, can create private spaces.

Process identifiers are used in recursive process definitions. It is assumed that each process identifiers A has a *single* defining equation $A\langle\tilde{X}, \tilde{u}, \tilde{x}\rangle$ and that all free (values, processes or localities) variables in P are contained in $\{\tilde{X}, \tilde{u}, \tilde{x}\}$. We also assume that all occurrences of process identifiers in P are guarded (i.e., each process identifier occurs within the scope of a blocking **in/read** prefix).

2.1 Operational Semantics

The evolution of a KLAIM net is described by singling out the tuples that are inserted, withdrawn or read from each node, or the processes that are spawned to other sites or the new/fresh sites that are created.

Example 1. Consider the net

$$N_1 = s_1 ::_{\rho_1} \mathbf{out}(t)@s_2.\mathbf{nil} || s_2 ::_{\rho_2} \mathbf{nil}$$

after placing the result of evaluating tuple t ($et = \mathcal{T} \llbracket t \rrbracket \rho$) on s_2 , it evolves to the net

$$N_2 = s_1 ::_{\rho_1} \mathbf{nil} || s_2 ::_{\rho_2} \mathbf{out}(et)$$

$\mathbf{nil}\{\rho\} = \mathbf{nil}$
$X\{\rho\} = X$
$(\mathbf{out}(t)@l.P)\{\rho\} = \mathbf{out}(t\{\rho\})@l\{\rho\}.P\{\rho\}$
$(\mathbf{eval}(Q)@l.P)\{\rho\} = \mathbf{eval}(Q)@l\{\rho\}.P\{\rho\}$
$(\mathbf{in}(t)@l.P)\{\rho\} = \mathbf{in}(t\{\rho\})@l\{\rho\}.P\{\rho\}$
$(\mathbf{read}(t)@l.P)\{\rho\} = \mathbf{read}(t\{\rho\})@l\{\rho\}.P\{\rho\}$
$(\mathbf{newloc}(u).P)\{\rho\} = \mathbf{newloc}(u).P\{\rho\}$
$(P_1 \mid P_2)\{\rho\} = P_1\{\rho\} \mid P_2\{\rho\}$
$A\langle\tilde{P}, \tilde{\ell}, \tilde{e}\rangle\{\rho\} = P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}]\{\rho\} \quad \text{if } A\langle\tilde{X}, \tilde{u}, \tilde{x}\rangle \stackrel{def}{=} P$
$e\{\rho\} = e$
$(\ell)\{\rho\} = \rho(\ell)$
$!x\{\rho\} = !x$
$!u\{\rho\} = !u$
$!X\{\rho\} = !X$
$(f, t)\{\rho\} = f\{\rho\}, t\{\rho\}$

Table 5. Closure Laws

We use labeled transitions to describe the evolution of nets. These labels indicate source and destination of information movement, the information transmitted and the kind of movement. We define the set of transition labels, *Lab*, as follows:

$$a ::= \mathcal{O}(s_1, et, s_2) \mid \mathcal{I}(s_1, et, s_2) \mid \mathcal{R}(s_1, et, s_2) \mid \mathcal{E}(s_1, P, s_2) \mid \mathcal{N}(s_1, -, s_2)$$

and we use a , possibly indexed, to range over Lab .

In Example 1 the label is $a = \mathcal{O}(s_1, t, s_2)$.

We use: $s \in N$ to denote that there exists a site named s in the net N ; $s_\rho \in N$ if $s \in N$ and the allocation environment of s is ρ ; $s_\rho :: P$ if $s_\rho \in N$ and P is running on s .

The operational semantics of KLAIM is given in Table 6. Where $\equiv$ is the structural congruence defined as the least congruence relation R such that:

$$\begin{aligned} & (N_1 \parallel N_2) R (N_2 \parallel N_1), \\ & ((N_1 \parallel N_2) \parallel N_3) R (N_1 \parallel (N_2 \parallel N_3)), \\ & (s ::_\rho^\delta (P_1 \mid P_2)) R (s ::_\rho^\delta P_1 \parallel s ::_\rho^\delta P_2). \end{aligned}$$

It easy to prove that this new labeled operational semantics coincides with the previous operational semantics based on rewriting systems [7].

We also write $N \xrightarrow{*} N'$ if and only if:

1. $N' = N$;
2. $\exists a, N'' : N \xrightarrow{a} N''$ and $N'' \xrightarrow{*} N'$.

Example 2. In this example we analyze a Client-Server application. A client sends data to be evaluated by the server. The server evaluates them and sends back the result to the client. We have two sites, one for the client, and the other for the server.

At the server site, named s_S , there is a process that is waiting for a tuple containing two expressions and a site name. When such a tuple is present, the server returns the sum of the values to the site and restarts.

At the client site, named s_C , there is a process that sends, to the server sites, the tuple $(3, 5, \mathbf{self})$ and waits for the result.

The KLAIM net for this system is:

$$\begin{aligned} & s_C ::_{\rho_C} \mathbf{out}(3, 5, \mathbf{self})@_{server}.\mathbf{in}(!result)@\mathbf{self}.\mathbf{nil} \\ & \parallel \\ & s_S ::_{\rho_S} ProcServer \end{aligned}$$

ProcServer is defined as follow.

$$ProcServer \stackrel{def}{=} \mathbf{in}(!x_1, !x_2, !u)@\mathbf{self}.\mathbf{out}(x_1 + x_2)@u.ProcServer$$

The evolution of the net start with the insertion of tuple $(3, 5, s_C)$ by the client in the tuple space of s_S (label $\mathcal{O}(s_C, (3, 5, s_C), s_S)$). Then process *ProcServer* in s_S first removes tuple $(3, 5, s_C)$ (label $\mathcal{I}(s_S, (3, 5, s_C), s_S)$), then inserts tuple (8) in the tuple space of s_C (label $\mathcal{O}(s_S, (8), s_C)$). Finally tuple (8), is removed from s_C (label $\mathcal{I}(s_C, (8), s_C)$).

$et = \mathcal{T}[\![t]\!]_{\rho} \quad s' = \rho(\ell) \quad a = \mathcal{O}(s, et, s')$		
$s ::_{\rho} \mathbf{out}(t)@l.P \parallel s' ::_{\rho} P' \xrightarrow{a} s ::_{\rho} P \parallel s' ::_{\rho} (P' \mid \mathbf{out}(et))$		
$s' = \rho(\ell) \quad a = \mathcal{E}(s, Q, s')$		
$s ::_{\rho} \mathbf{eval}(Q)@l.P \parallel s' ::_{\rho} P' \xrightarrow{a} s ::_{\rho} P \parallel s' ::_{\rho} (P' \mid Q)$		
$match(\mathcal{T}[\![t]\!]_{\rho}, et) \quad s' = \rho(\ell) \quad a = \mathcal{I}(s, et, s')$		
$s ::_{\rho} \mathbf{in}(t)@l.P \parallel s' ::_{\rho} \mathbf{out}(et) \xrightarrow{a} s ::_{\rho} P[et/\mathcal{T}[\![t]\!]_{\rho}] \parallel s' ::_{\rho} \mathbf{nil}$		
$match(\mathcal{T}[\![t]\!]_{\rho}) \quad s' = \rho(\ell) \quad a = \mathcal{R}(s, et, s')$		
$s ::_{\rho} \mathbf{read}(t)@l.P \parallel s' ::_{\rho} \mathbf{out}(et) \xrightarrow{a} s ::_{\rho} P[et/\mathcal{T}[\![t]\!]_{\rho}] \parallel s' ::_{\rho} \mathbf{out}(et)$		
$s' \neq s \quad a = \mathcal{N}(s, -, s')$		
$s ::_{\rho} \mathbf{newloc}(u).P \xrightarrow{a} s ::_{\rho} P[s'/u] \parallel s' ::_{\rho} \mathbf{nil}$		
$s ::_{\rho} P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}] \xrightarrow{a} N \quad A(\tilde{X}, \tilde{u}, \tilde{x}) \stackrel{def}{=} P$		
$s ::_{\rho} A(\tilde{P}, \tilde{\ell}, \tilde{e}) \xrightarrow{a} N$		
$N_1 \xrightarrow{a} N_2 \quad a \neq \mathcal{N}(s_1, -, s_2)$		
$N_1 \parallel N \xrightarrow{a} N_2 \parallel N$		
$N_1 \xrightarrow{a} N_2 \quad a = \mathcal{N}(s_1, -, s_2) \quad s_2 \notin N \quad N_1 \equiv N_2 \quad N_1 \xrightarrow{a} N$		
$N_1 \parallel N \xrightarrow{a} N_2 \parallel N \quad N_2 \xrightarrow{a} N$		

Table 6. The Operational Semantics

3 A Logic for KLAIM

We now introduce a logic that allows us to specify and prove properties of mobile system specified in KLAIM. In our view the important features of a KLAIM system are the tuples residing at specific nodes and the actions that a system performs during its evolution.

Our logic aims at capturing these two aspects. It permits to specify the presence of a tuple et inside the tuple space of a node s , by means of the atomic formula $et@s$, and the possible evolutions by means of the modal operators $\langle \cdot \rangle$, indexed by sets of actions.

3.1 Syntax

We use σ as a generic element in $\mathcal{S} \cup VLoc$. We also use $\mathbb{V}\mathbb{A}\mathbb{R}$ for $VLoc \cup Var \cup VProc$ and its elements are denoted with id , while $\mathbb{V}\mathbb{A}\mathbb{L}$ stands for $Val \cup Proc \cup \mathcal{S}$ and its elements are ranged by v .

To denote sets of actions that a KLAIM system can perform, we define the set of *abstract actions* $ALab$. An abstract action α is defined as follows:

$$\alpha ::= \mathcal{O}(\sigma_1, t, \sigma_2) \mid \mathcal{I}(\sigma_1, t, \sigma_2) \mid \mathcal{R}(\sigma_1, t, \sigma_2) \mid \mathcal{E}(\sigma_1, P, \sigma_2) \mid \mathcal{N}(\sigma_1, -, \sigma_2)$$

Obviously $Lab \subset ALab$.

Let $VLog$ be the set of logical variable ranged over by κ . We define $\mathcal{L}$ as the set of formulae ϕ obtainable by the following grammar:

$$\phi ::= \mathbf{tt} \mid t@ \sigma \mid \langle A \rangle \phi \mid \kappa \mid \nu \kappa. \phi \mid \phi \vee \phi \mid \neg \phi$$

where A is a subset of $ALab$. We shall also assume that no variable κ occurs negatively (i.e. under the scope of an odd number of $\neg$ operators) in ϕ .

We will use: $\langle \alpha \rangle \phi$ for $\langle \{\alpha\} \rangle \phi$, $\langle - \rangle \phi$ for $\langle Lab \rangle \phi$ and $\langle -A \rangle$ for $\langle Lab - A \rangle \phi$.

We say that a variable id is *bound* in ϕ if every occurrence of id in ϕ appears in the scope of some $\langle A \rangle$ with $id \in \alpha$ for every $\alpha \in A$. A formula ϕ is closed if every variable in ϕ is bound.

Definition 1. We define $Subst \subseteq VLoc \rightarrow \mathcal{S} \uplus VProc \rightarrow VProc \uplus Var \rightarrow Val$, δ , *sometime with indexes*, will be used to denote elements of $Subst$.

If $\delta \in Subst$ and id is a variable then $\delta(id)$ is a value v of the same type of id . The closure of a formula ϕ with respect to a substitution δ ($\phi\{\delta\}$) is the formula ϕ' obtained from replacing *every variable* id in ϕ with $\delta(id)$. We also use $\delta_1 \cdot \delta_2$ for the substitution δ such that: $\delta(id) = \delta_2(id)$ if $\delta_2(id)$ is defined, $\delta(id) = \delta_1(id)$ otherwise.

3.2 Semantics

For specifying sets of actions that are characterized by common aspects, as source or destination of information movement, structure of the information transmitted and kind of movement, we use abstract actions.

Thus we first define the set of labels denoted by an abstract action α ($\mathcal{A}[\alpha]$) as follows:

$$\mathcal{A}[\alpha] = \{a \mid \exists \delta : a = \alpha\{\delta\}\}$$

i.e. $\mathcal{A}[\alpha]$ is the set of action a such that there exists a substitution δ for which $a = \alpha\{\delta\}$; if $a \in \mathcal{A}[\alpha]$ then we use δ_α^a for a δ' such that $\alpha\{\delta'\} = a$.

For example let $\alpha = \mathcal{I}(u, ("hello"), s)$ then

$$\mathcal{A}[\alpha] = \{\mathcal{I}(s', ("hello"), s) \mid s' \in \mathcal{S}\}$$

and for $a = \mathcal{I}(s'', ("hello"), s) \in \mathcal{A}[\alpha]$ we have that $\delta_\alpha^a = \{s''/u\}$.

Definition 2. We define the logical environment Env as $Env \subseteq [VLog \rightarrow Subst \rightarrow Net^*]$. We also use e , *sometime with indexes*, to denote elements in Env . Moreover we use $e \cdot [\kappa \mapsto g]$ for the logical environment e' such that $e'(\kappa') = e(\kappa')$ if $\kappa \neq \kappa'$, $e'(\kappa) = g$ otherwise.

We define $\mathcal{M}[\cdot] : \mathcal{L} \rightarrow Env \rightarrow Subst \rightarrow Net^*$ to denote the set of nets that are models of a logical formula. Function $\mathcal{M}[\cdot]$ is defined by structural induction as follows:

- $\mathcal{M}[\mathbf{tt}]e\delta = Net$;
- $\mathcal{M}[\kappa]e\delta = e(\kappa)\delta$
- $\mathcal{M}[t@ \sigma]e\delta = \{N \mid s = \sigma\{\delta\}, et = t\{\delta\}, \exists \rho. s ::_{\rho} \mathbf{out}(et) \in N\}$;
- $\mathcal{M}[\langle \alpha \rangle \phi]^He\delta = \{N \mid \exists a \exists N' : N \succ_a^a N' \wedge a \in \mathcal{A}[\alpha\{\delta\}] \wedge N' \in \mathcal{M}[\phi]^He\delta \cdot \delta_{\alpha\{\delta\}}^a\}$;
- $\mathcal{M}[\langle A_1 \cup A_2 \rangle \phi]e\delta = \mathcal{M}[\langle A_1 \rangle \phi]e\delta \cup \mathcal{M}[\langle A_2 \rangle \phi]e\delta$
- $\mathcal{M}[\phi_1 \vee \phi_2]e = \mathcal{M}[\phi_1]e\delta \cup \mathcal{M}[\phi_2]e\delta$;
- $\mathcal{M}[\neg \phi]e\delta = Net - \mathcal{M}[\phi]e\delta$;
- $\mathcal{M}[\nu \kappa. \phi]e\delta = \nu f_{\kappa, e}^{\phi} \delta$ where:
 1. $f_{\kappa, e}^{\phi} : [Subst \rightarrow Net^*] \rightarrow [Subst \rightarrow Net^*]$ is defined as follows:

$$f_{\kappa, e}^{\phi}(g) = \mathcal{M}[\phi]e \cdot [\kappa \mapsto g]$$

2. $\nu f_{\kappa, e}^{\phi} = \bigcup \{g \mid g \subseteq f_{\kappa, e}^{\phi}(g)\}$ where $g_1 \subseteq g_2$ if and only if for all δ $g_1(\delta) \subseteq g_2(\delta)$.

Other formulae like $[A]\phi$, $\mu \kappa. \phi$ or $\phi_1 \wedge \phi_2$ can be expressed with formulae in $\mathcal{L}$. Indeed $[A]\phi = \neg \langle A \rangle \neg \phi$, $\mu \kappa. \phi = \neg \nu \kappa. \neg \phi[\neg \kappa / \kappa]$ and $\phi_1 \wedge \phi_2 = \neg(\phi_1 \vee \neg \phi_2)$.

Definition 3. Let N be a net and ϕ be a closed formula, we say that N is a model of ϕ , written $N \models \phi$, if and only if $N \in \mathcal{M}[\phi]e_0\delta_0$, where $e_0 = \lambda \kappa. \delta_0$ and $\delta_0 = \emptyset$.

Example 3. If we consider the *Client/Server* application of Example 2, a property that we would like specify/verify is that if the tuple (x_1, x_2, u) is sent to the server then the tuple $(x_1 + x_2)$ is sent to the locality u from the server. This property can be specified with the formulae:

$$\begin{aligned} \phi &= \neg \nu \kappa. \neg (\langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle (\phi_1) \vee \neg (\neg \langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \mathbf{tt} \\ &\quad \vee \neg \langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \neg \kappa)) \\ \phi_1 &= \neg \nu \kappa'. \neg (\langle \mathcal{O}(u_2, (x_1 + x_2), u_1) \rangle \mathbf{tt} \vee \neg (\langle \neg \mathcal{O}(u_2, (x_1 + x_2), u_1) \rangle \mathbf{tt} \\ &\quad \vee \neg \langle \neg \mathcal{O}(u_2, (x_1 + x_2), u_1) \rangle \neg \kappa')) \end{aligned}$$

3.3 The Proof System

We now introduce a tableau based proof system for $\mathcal{L}$ formulae. This proof system is based on [2] where a tableau-based system for μ -calculus has been introduced.

The proof rules operate on *sequents* of the form $H \vdash N : \phi$, where H is a set of *hypothesis* of the form $N' : \phi'$, N is a net, and ϕ is a closed formula. More correctly we should have written $H \vdash_{Net} N : \phi$, because we interpret N over Net , we omit the annotation for the sake of simplicity. We will refer to sequents by π and to proofs by Π .

If ϕ_1 and ϕ_2 are formulae, we say that ϕ_1 is an *immediate sub-term* of ϕ_2 , written $\phi_1 <_I \phi_2$, if one of the following holds:

1. $\phi_2 = \neg\phi_1$;
2. $\phi_2 = \phi_1 \vee \phi_3$ or $\phi = \phi_3 \vee \phi_1$, for some ϕ_3 ;
3. $\phi_2 = \langle A \rangle \phi_1$;
4. $\phi_2 = \nu\kappa.\phi_1$.

We write $\prec$ for the transitive closure of $\prec_I$, and $\preceq$ for the transitive and reflexive closure of $\prec_I$.

$\frac{H \vdash N : \phi_i}{H \vdash N : \phi_1 \vee \phi_2} R1$	$\frac{H \vdash N : \neg\phi_1 \quad H \vdash N : \neg\phi_2}{H \vdash N : \neg(\phi_1 \vee \phi_2)} R2$
$\frac{H \vdash N : \phi}{H \vdash N : \neg\neg\phi} R3$	$\frac{H \vdash N' : \phi\{\delta_\alpha^a\}}{H \vdash N : \langle A \rangle \phi} R4 - \left[N \succ^a N', \alpha \in A, a \in \mathcal{A}[\alpha] \right]$
$\frac{H \vdash N_1 : \neg\phi\{\delta_\alpha^{a_1}\} \quad H \vdash N_2 : \neg\phi\{\delta_\alpha^{a_2}\} \quad \dots}{H \vdash N : \neg\langle A \rangle \phi} R5 - \left[\begin{array}{l} \forall i \ N \succ^{a_i} N_i, a_i \in \mathcal{A}[\alpha_i], \\ \alpha_i \in A \end{array} \right]$	
$\frac{H' \cup \{N : \nu\kappa.\phi\} \vdash N : \phi[\nu\kappa.\phi/\kappa]}{H \vdash N : \nu\kappa.\phi} R6 - [N : \nu\kappa.\phi \notin H]$	
$\frac{H' \cup \{N : \nu\kappa.\phi\} \vdash N : \neg\phi[\nu\kappa.\phi/\kappa]}{H \vdash N : \neg\nu\kappa.\phi} R7 - [N : \nu\kappa.\phi \notin H]$	
where $H' = H - \{N' : \phi' \mu\kappa.\phi \prec \phi'\}$	

Table 7. The proof system

Definition 4.

1. A sequent $H \vdash N : \phi$ is successful if
 - $\phi = \mathbf{tt}$.
 - $\phi = \nu\kappa.\phi'$ and $N : \nu\kappa.\phi' \in H$;
 - $\phi = \neg\langle A \rangle \phi'$, and $\nexists a \in \mathcal{A}[A]$ such that $N \succ^a N'$;
 - $\phi = et@s$ and $s ::_\rho \mathbf{out}(et) \in N$;
 - $\phi = \neg et@s$ and $s ::_\rho \mathbf{out}(et) \notin N$;
2. Π is a successful proof for π if the following conditions hold:
 - Π is built using the rules on Table 7;
 - π is the root of Π ;
 - every leaf on Π is a successful sequent.
3. π is provable if and only if there exists a successful proof Π for π .

We define the models of a formula ϕ with the hypothesis H , $\mathcal{M}[\phi]^H$, as follows:

- $\mathcal{M}[\mathbf{tt}]^H e\delta = Net$;
- $\mathcal{M}[\kappa]^H e\delta = e(\kappa)\delta$
- $\mathcal{M}[t@s]^H e\delta = \{N | s = \sigma\{\delta\}, et = t\{\delta\}, s ::_\rho \mathbf{out}(et) \in N\}$;

- $\mathcal{M}[\langle \alpha \rangle \phi]^H e\delta = \{N \mid \exists a \exists N' : N \succ^a N' \wedge a \in \mathcal{A}[\alpha\{\delta\}] \wedge N' \in \mathcal{M}[\phi]^H e\delta \cdot \delta_{\alpha\{\delta\}}^a\};$
- $\mathcal{M}[\langle \alpha \rangle \phi]^H e\delta = \{N \mid \exists a \exists N' : N \succ^a N' \wedge a \in \mathcal{A}[\alpha\{\delta\}] \wedge N' \in \mathcal{M}[\phi]^H e\delta \cdot \delta_{\alpha\{\delta\}}^a\};$
- $\mathcal{M}[\phi_1 \vee \phi_2]^H e\delta = \mathcal{M}[\phi_1]^H e\delta \cup \mathcal{M}[\phi_2]^H e\delta;$
- $\mathcal{M}[\neg \phi]^H e\delta = \text{Net} - \mathcal{M}[\phi]^H e\delta;$
- $\mathcal{M}[\nu \kappa. \phi] e\delta = \nu f_{\kappa, e}^{\phi, h} \delta \cup h\delta$ where:
 1. $f_{\kappa, e}^{\phi, h} : [\text{Subst} \rightarrow \text{Net}^*] \rightarrow [\text{Subst} \rightarrow \text{Net}^*]$ is defined as follows:

$$f_{\kappa, e}^{\phi, h}(g) = f_{\kappa, e}^{\phi}(g \cup h)$$

2. $h : \text{Subst} \rightarrow \text{Nets}^*$ is defined as follows:

$$h\delta = \{N \mid N : \nu \kappa. \phi\{\delta\} \in H\}$$

3. $\nu f_{\kappa, e}^{\phi, h} = \bigcup \{g \mid g \subseteq f_{\kappa, e}^{\phi, h}(g)\}.$

If $H = \emptyset$ then $\mathcal{M}[\phi]^H e\delta = \mathcal{M}[\phi] e\delta.$

Definition 5. Let N be a net, and let ϕ be a closed formula, we say that N is a model of ϕ under the hypothesis H , written $N \models_H \phi$, if and only if $N \in \mathcal{M}[\phi]^H e_0 \delta_0$, with $e_0 = \lambda \kappa. \delta_0$ and $\delta_0 = \emptyset$.

Theorem 1. If there exists a proof Π for $H \vdash N : \phi$ then $N \models_H \phi$.

Theorem 2. Let N be such that the set $\{N' \mid N \succ^* N'\}$ is finite then, for all closed formula ϕ , $N \models_H \phi$ implies $H \vdash N : \phi$ provable.

Theorem 3. Let ϕ be a closed formula such that:

- if $\nu \kappa. \phi'$ is a subformula of ϕ then it is negative in ϕ ;
- if $\langle A \rangle \phi'$ of ϕ is such that, if there exists $\alpha = \mathcal{N}(\sigma, -, u) \in A$, then $\langle A \rangle \phi'$ is no negative in ϕ ;

then for all net N and for all set of hypothesis H if $N \models_H \phi$ then $H \vdash N : \phi$ is provable.

Example 4. We want now to show how, using the proof system, we can prove that system CS of Example 2 satisfies formula ϕ of Example 3.

Thus we want prove that sequent $\emptyset \vdash CS : \phi$ is provable, i.e. there exists a proof for it. Now the only rule that we can apply to the sequent is $R7$. Thus we start our proof as follows:

$$\frac{CS : \nu \kappa. \phi' \vdash CS : \neg \neg (\langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle (\phi_1) \vee \neg (\neg \langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \mathbf{tt} \vee \langle \neg \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \neg \nu \kappa. \phi'))}{\emptyset \vdash CS : \phi}$$

where

$$\phi' = \neg(\langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle(\phi_1) \vee \neg(\neg\langle -\mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \mathbf{tt} \vee \langle -\mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \neg\kappa))$$

We can now proceed by applying rules *R3* and *R1* obtaining:

$$CS : \nu\kappa.\phi' \vdash CS : \langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \phi_1$$

Net *CS* can only evolve, by action $\mathcal{O}(s_C, (3, 5, s_C), s_S)$, to

$$\begin{aligned} SC' &= s_C ::_{\rho_C} \mathbf{in}(!result)@\mathbf{self.nil} \\ &\parallel \\ &s_S ::_{\rho_S} \mathbf{in}(!x_1, !x_2, !u)@\mathbf{self.out}(x_1 + x_2)@u.ProcServer \\ &\quad | \mathbf{out}(3, 5, s_C) \end{aligned}$$

then applying rule *R4* we have that:

$$\frac{CS : \nu\kappa.\phi' \vdash CS' : \neg\nu\kappa'.\neg(\langle \mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt} \vee \neg(\neg\langle -\mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt} \vee \langle -\mathcal{O}(s_S, (8), s_C) \rangle \neg\kappa'))}{CS : \nu\kappa.\phi' \vdash CS : \langle \mathcal{O}(u_1, (x_1, x_2, u), u_2) \rangle \phi_1}$$

Let ϕ'_1 be such that

$$\phi'_1 = \neg(\langle \mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt} \vee \neg(\neg\langle -\mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt} \vee \langle -\mathcal{O}(s_S, (8), s_C) \rangle \neg\kappa'))$$

then by rule *R7* we have that

$$\frac{CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1 \vdash CS' : \neg\phi'_1[\nu\kappa'.\phi'/\kappa']}{CS : \nu\kappa.\phi' \vdash CS' : \neg\nu\kappa'.\phi'_1}$$

As in a previous case, applying rules *R3* and *R1*, we obtain the sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1 \vdash CS' : \neg(\neg\langle -\mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt} \vee \langle -\mathcal{O}(s_S, (8), s_C) \rangle \neg\nu\kappa'.\phi'_1)$$

applying *R2* we have to prove sequents:

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1 \vdash CS' : \neg\neg\langle -\mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt} \quad (1)$$

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1 \vdash CS' : \neg\langle -\mathcal{O}(s_S, (8), s_C) \rangle \neg\nu\kappa'.\phi'_1 \quad (2)$$

Net *CS'* can only evolve, by an action $\mathcal{I}(s_S, (3, 5, s_C), s_S)$, to the net:

$$CS'' = s_C ::_{\rho_C} \mathbf{in}(!result)@\mathbf{self.nil} \parallel s_S ::_{\rho_S} \mathbf{out}(3 + 5)@s_C.ProcServer$$

Then by rule *R3* and *R4*, for (1), we obtain the successfully sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1 \vdash CS' : \mathbf{tt}$$

while for (2) we obtain, by rule *R5*, sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1 \vdash CS'' : \neg\nu\kappa'.\phi'_1$$

Applying rules $R7$, $R3$ and $R1$ again, we obtain the sequent

$$CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1, CS'' : \nu\kappa'.\phi'_1 \vdash CS''' : \langle \mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt}$$

Net CS'' evolves, by $\mathcal{O}(s_S, (8), s_C)$, to the net

$$CS''' = s_C ::_{\rho_C} \mathbf{in}(!result)@\mathbf{self.nil}|\mathbf{out}(8) \parallel s_S ::_{\rho_S} ProcServer$$

thus, by rule $R4$, we have

$$\frac{CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi_1, CS'' : \nu\kappa'.\phi'_1 \vdash CS''' : \mathbf{tt}}{CS : \nu\kappa.\phi', CS' : \nu\kappa'.\phi'_1, CS'' : \nu\kappa'.\phi'_1 \vdash CS''' : \langle \mathcal{O}(s_S, (8), s_C) \rangle \mathbf{tt}}$$

hence we have obtained a proof of $\emptyset \vdash CS : \neg\nu\kappa.\phi'$.

4 An Extended Example

In this section we consider a larger example of a Distribute Information System management.

We assume that a Database system is distributed over three different sites, named Inf_i ($i \in \{1, 2, 3\}$). A node, named *Manager*, manages the database system sending processes for updating the information on the nodes. The updating process chooses a path to reach every node. Only one updating-process at a time can be executed in a site. For this reason inside the tuple space of Inf_i there is the tuple $"F''$. An updating process can be evaluated in an Inf_i node only when tuple $"F''$ is in its tuple space.

The net of the distributed database is defined as follows:

$$Inf_1 :: \mathbf{out}("F'') \parallel Inf_2 :: \mathbf{out}("F'') \parallel Inf_3 :: \mathbf{out}("F'')$$

In the tuple space of node *Manager* there is a tuple (" G'' ") for each node Inf_i . An updating process can be started only when at least a tuple (" G'' ") is in the tuple space of *Manager*.

Process *StartAgent* looks for a tuple (" G'' "). When this tuple is found, the process *CallUpdate*, which starts the updating procedure, is called. Guarding *CallUpdate* in *StartAgent* with an $\mathbf{in}("G'')$ we ensure that the system is deadlock free.

$$StartAgent = \mathbf{in}("G'')@\mathbf{self}. (CallUpdate(Inf_1, Inf_2, Inf_3) | StartAgent)$$

$$CallUpdate\langle u_1, u_2, u_3 \rangle = \mathbf{in}("F'')@u_1.\mathbf{out}("updating")@u_1. \\ \mathbf{eval}(Update(u_2, Update(u_3, FUpdate(Manager))))@u_1.\mathbf{nil}$$

$$Update\langle u, X \rangle = \mathbf{in}("F'')@u.\mathbf{out}("updating")@u.\mathbf{eval}(X)@u. \\ \mathbf{in}("updating")@\mathbf{self}.\mathbf{out}("F'')@\mathbf{self}.\mathbf{nil}$$

$$FUpdate\langle u \rangle = \mathbf{in}("updating")@\mathbf{self}.\mathbf{out}("F'')@\mathbf{self}.\mathbf{eval}(Success)@u.\mathbf{nil}$$

$$Success = \mathbf{out}("G'')@\mathbf{self}.\mathbf{nil}$$

The manager node is define as follows:

$$\text{Manager} :: \text{StartAgent}|\text{out}(Inf_1)|\text{out}(Inf_2)|\text{out}(Inf_3) \\ |\text{out}("G")|\text{out}("G")|\text{out}("G")$$

For this system, we would like to specify that if a process $Update(s, P)$ (respectively $FUpdate(s)$) is evaluated in a site Inf_i , for some site s and some process P , then no processes are evaluated on Inf_i until process P (respectively $Success$) is evaluated from Inf_i to the site s . This property is specified with the following formulae:

$$\phi_1 = \neg\langle\mathcal{E}(u_1, Update(u_2, X), u_3)\rangle\neg(\nu\kappa_1. \langle\mathcal{E}(u_3, X, u_2)\rangle\mathbf{tt}\vee \\ \neg(\langle\mathcal{E}(u_4, X', u_3)\rangle\mathbf{tt}\vee \\ \langle-\mathcal{E}(u_3, X, u_2)\rangle\neg\kappa_1)$$

$$\phi_2 = \neg\langle\mathcal{E}(u_1, FUpdate(u_2), u_3)\rangle\neg(\nu\kappa_2. \langle\mathcal{E}(u_3, Success, u_2)\rangle\mathbf{tt}\vee \\ \neg(\langle\mathcal{E}(u_4, X', u_3)\rangle\mathbf{tt}\vee \\ \langle-\mathcal{E}(u_3, Success, u_2)\rangle\neg\kappa_2)$$

We wish that ϕ_1 and ϕ_2 was verified in every reachable state of our system. This is specified with the formula:

$$\phi = \nu\kappa. \neg((\neg\phi_1 \vee \neg\phi_2) \vee \neg\langle-\rangle\neg\kappa)$$

Due to space limitation we omit the proof for ϕ .

5 Behaviours of Nets

In this section we introduce a new equivalence relation between KLAIM nets and we will show as it is in full agreement with the one induced by the proposed logics.

Nets will be compared according to their action tree or *behaviour*. The behaviours of nets are generated respect the following syntax:

$$\Gamma ::= \perp \quad | \quad \omega \quad | \quad a \rightarrow \Gamma \quad | \quad \Gamma \wedge \Gamma \quad | \quad et@s$$

The set of all possible behaviour will be denoted by Γ .

We will write $N : \Gamma$ to indicate that the net N has the behavior Γ . A particular behaviour $\perp$ is introduced to represent *fully unspecified behaviour*; every net N has $\perp$ ($N : \perp$) as a possible behaviour. A net N has a behaviour $et@s$ if the tuple et is in the tuple spaces of the site s of N .

The behaviour $a \rightarrow \Gamma$ represent the set of nets that are able to perform an action a and then behaves like Γ , so a net N has a behaviour $\Gamma = a \rightarrow \Gamma'$ if N' exists such that $N \xrightarrow{a} N'$ and $N' : \Gamma'$.

A net N has a behaviour $\Gamma_1 \wedge \Gamma_2$ if it has both Γ_1 and Γ_2 ($N : \Gamma_1 \wedge \Gamma_2$ if $N : \Gamma_1$ and $N_2 : \Gamma_2$).

The behaviour ω represent the capability of performing any actions; no net has behaviour ω .

Definition 6. We say that $N : \Gamma$ if and only if we are able to prove that with the following rules:

$$\frac{}{N : \perp} \quad \frac{N \xrightarrow{a} N' \quad N' : \Gamma}{N : a \rightarrow \Gamma} \quad \frac{N : \Gamma_1 \quad N : \Gamma_2}{N : \Gamma_1 \wedge \Gamma_2}$$

To reason on behaviours we introduce an ordering between them.

Definition 7. $\leq$ is the smallest relation defined as follows:

- $\Gamma \leq \omega$
- $\perp \leq \Gamma$
- if $\Gamma_1 \leq \Gamma_2$ then $a \rightarrow \Gamma_1 \leq a \rightarrow \Gamma_2$
- if $\Gamma_1 \leq \Gamma_2$ and $\Gamma_2 \leq \Gamma_3$ then $\Gamma_1 \leq \Gamma_3$
- $\Gamma_1 \wedge \Gamma_2 \leq \Gamma_2 \wedge \Gamma_1$
- $\Gamma_1 \leq \Gamma \wedge \Gamma_1$
- if $\Gamma_1 \leq \Gamma_2$ then $\Gamma_1 \wedge \Gamma \leq \Gamma_2 \wedge \Gamma$

If we interpret behaviours as requirements on computing agents then the ordering $\Gamma' \leq \Gamma$ indicates that a net with a behaviour Γ satisfies more requirements than a net with a behaviour Γ' .

In this point of view ω is the highest while $\perp$ is the lowest. If $N_1 : \Gamma_1 \wedge \Gamma_2$ then N_1 has both the behaviours Γ_1 and Γ_2 and if $N_2 : \Gamma_2$ then N_1 satisfies more requirements. So $\Gamma_2 \leq \Gamma_1 \wedge \Gamma_2$ and the operator $\wedge$ is commutative and associative.

Definition 8.

1. We write $N_1 \sqsubseteq N_2$ if and only if for all Γ_1 if $N_1 : \Gamma_1$ then there exists Γ_2 , with $N_2 : \Gamma_2$, such that $\Gamma_1 \leq \Gamma_2$;
2. We write $N_1 \simeq N_2$ if and only if $N_1 \sqsubseteq N_2$ and $N_2 \sqsubseteq N_1$.

Theorem 4. For all net N_1, N_2 we have that $N_1 \simeq N_2$ if and only if for all formula $\phi \in \mathcal{L}$ $N_1 \models \phi$ if and only if $N_2 \models \phi$

References

1. Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(10):444–458, October 1989. Technical Correspondence.
2. Rance Cleaveland. Tableau-based model checking in the propositional μ -calculus. *Acta Informatica*, 27(8):725–747, September 1990.
3. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
4. D. Gelernter. Multiple tuple spaces in linda. In J.Hartmanis G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *LNCS*, pages 20–27, 1989.
5. Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

6. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989. SU Fisher Research 511/24.
7. Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special issue: Mobility and Network Aware Computing.
8. Rocco De Nicola and Michele Loreti. A logic for klaim (full paper). Available at <ftp://rap.dsi.unifi.it/papers/fullMLK.ps>.

Kleene under a Demonic Star

Jules Desharnais¹, Bernhard Möller², and Fairouz Tchier³

¹ Département d'informatique, Université Laval, Québec QC G1K 7P4 Canada
`Jules.Desharnais@ift.ulaval.ca`

² Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany
`Bernhard.Moeller@informatik.uni-augsburg.de`

³ Mathematics Department, King Saud University, P.O.Box 22452, Riyadh 11495,
Saudi Arabia,
`f66m002@ksu.edu.sa`

Abstract. In relational semantics, the input-output semantics of a program is a relation on its set of states. We generalize this in considering elements of Kleene algebras as semantical values. In a nondeterministic context, the *demonic* semantics is calculated by considering the worst behavior of the program. In this paper, we concentrate on while loops. *Calculating* the semantics of a loop is difficult, but *showing the correctness* of any candidate abstraction is much easier. For deterministic programs, Mills has described a checking method known as the *while statement verification rule*. A corresponding programming theorem for nondeterministic iterative constructs is proposed, proved and applied to an example. This theorem can be considered as a generalization of the while statement verification rule to nondeterministic loops.

Keywords: while loop, demonic semantics, relational abstraction, verification, Kleene algebra, rule, generalization.

1 Introduction

We use elements of Kleene algebras as abstractions of the input-output semantics of nondeterministic programs. In the concrete Kleene algebra of homogeneous binary relations, the operators $\cup$ and $;$ have been used for many years to define the so-called *angelic semantics*, which assumes that a program goes right when there is a possibility to go right. The *demonic choice* $\sqcup$ and *demonic composition* $\sqcap$ do the opposite: if there is a possibility to go wrong, a program whose semantics is given by these operators goes wrong. The demonic semantics of a while loop is given as a fixed point of a monotonic function involving the demonic operators.

While there is no systematic way to calculate the relational abstraction of a while loop directly from the definition, it is possible to check the correctness of any candidate abstraction. For deterministic programs, Mills [15,16] has described a checking method known as the *while statement verification rule*. We generalize this rule to nondeterministic loops.

The rest of this paper is organized as follows. In Section 2, we present our mathematical tool, namely Kleene algebra [8]. There, a concept of type can be defined that allows an abstract treatment of the domain (of definedness) of an element, and also of assertions. After some auxiliary results (Section 3 on fixed points and Section 4 on demonic operators), we present in Section 5 a generalization of the *while statement verification rule* of Mills. This is followed by an example of application in Section 6.

We note here that half of the generalized theorem was shown by Sekerinski [20], who uses an approach based on predicative programming [12]. A related theorem has been given by Norvell [18] in the framework of predicative programming with time bounds. Norvell's theorem shows how to refine the specification R of a while loop under the condition that R is *strongly bounded*, which guarantees termination after a finite amount of time. Further refinement theorems for loops can be found in [2], presented in the framework of predicate transformers.

The main novelties in the present paper are the following. First, we fully generalize Mills's approach to the nondeterministic case. This was already achieved by J. Desharnais and F. Tchie [21,22] using binary homogeneous relations. Second, at the same time we abstract from relational semantics to the more general setting of Kleene algebras with types as introduced by B. Möller in [17]. In this we present some derived operations and laws that will also be useful for further applications of Kleene algebras. It is remarkable that the proofs in the generalized setting are considerably simpler and more perspicuous than the corresponding ones in terms of relations or predicate transformers.

2 Kleene Algebras

2.1 Definition and Basic Laws

In our definition of a Kleene algebra, we follow [8], since we want to admit general recursive definitions, not just the Kleene star. We are well aware that there are different definitions (see e.g. [13]).

Definition 2.1. A Kleene algebra (KA) is a sextuple $(K, \leq, \top, \cdot, 0, 1)$ satisfying the following properties:

- (a) $(K, \leq)$ is a complete lattice with least element 0 and greatest element $\top$. The supremum of a subset $L \subseteq K$ is denoted by $\sqcup L$.
- (b) $(K, \cdot, 1)$ is a monoid.
- (c) The operation $\cdot$ is universally disjunctive (i.e. distributes through arbitrary suprema) in both arguments.

The supremum of two elements $x, y \in K$ is given by $x + y \triangleq \sqcup \{x, y\}$.

Perhaps the best-known example of a KA is LAN $\triangleq (\mathcal{P}(A^*), \subseteq, A^*, \bullet, \emptyset, \varepsilon)$, the algebra of formal languages over some alphabet A , where A^* is the set of all finite words over A , $\bullet$ denotes concatenation and ε the empty word (as usual, we identify a singleton language with its only element).

Another important KA is $\text{REL} \triangleq (\mathcal{P}(M \times M), \subseteq, M \times M, ;, \emptyset, I)$, the algebra of homogeneous binary relations over some set M under relational composition $;$. More generally than the concrete relation algebra REL , every abstract relation algebra (see e.g. [7,19]) is a KA.

Definition 2.2. *A KA is called Boolean if its underlying lattice $(K, \leq)$ is a Boolean algebra, i.e. a completely distributive and complemented lattice. The complement of an element $a \in K$ is denoted by $\bar{a}$.*

2.2 Types

Definition 2.3. *A type of a KA is an element t with $t \leq 1$.*

This definition is best illustrated in the KA REL . There, a type $1_T \subseteq I$ is a partial identity relation of the form $1_T \triangleq \{(x, x) \mid x \in T\}$ for some subset $T \subseteq M$. So it models the assertion of belonging to T .

Now, restriction of a relation $R \subseteq M \times M$ to arguments of type T , i.e. the relation $R \cap T \times M$, can also be described by composing R with 1_T from the left: $R \cap T \times M = 1_T ; R$. Similarly, co-restriction is modeled by composing a partial identity from the right. Finally, consider types $S, T \subseteq M$ and binary relation $R \subseteq M \times M$. Then $R \subseteq S \times T \Leftrightarrow 1_S ; R ; 1_T = R$. In other words, the “default typing” $M \times M$ of R can be narrowed down to $S \times T$ iff restriction to S and co-restriction to T do not change R .

These observations are the basis for our view of types as subidentities and our algebraic treatment of restriction and co-restriction. For a different, but related, approach see [13].

Lemma 2.4. *Assume a Boolean KA. Then the following hold:*

- (a) *All types are idempotent, i.e. $t \leq 1 \Rightarrow t \cdot t = t$.*
- (b) *The infimum of two types is their product: $s, t \leq 1 \Rightarrow s \cdot t = s \sqcap t$. In particular, all types commute under the $\cdot$ operation.*
- (c) *For all families L of types, $(\bigsqcap L) \cdot \top = \bigsqcap (L \cdot \top)$.*

Definition 2.5. *The negation of a type $t \leq 1$ in a typed KA is $\neg t \triangleq \bar{t} \sqcap 1$.*

2.3 Domain and Codomain

Definition 2.6. *In a KA $(K, \leq, \top, \cdot, 0, 1)$, we define, for $a \in K$, the domain $\ulcorner a$ via the Galois connection $\forall (y : y \leq 1 : \ulcorner a \leq y \stackrel{\text{def}}{\Leftrightarrow} a \leq y \cdot \top)$.*

This is well defined because of Lemma 2.4, see also [1]. Hence the operation $\ulcorner$ is universally disjunctive and therefore monotonic and strict. Moreover, the definition implies $a \leq \ulcorner a \cdot \top$. The co-domain $\lceil a$ is defined symmetrically.

We list a number of useful properties of the domain operation (see again also [1]); analogous ones hold for the co-domain operation.

Lemma 2.7. *Consider a KA $(K, \leq, \top, \cdot, 0, 1)$ and $a, b, c \in K$.*

- (a) $\ulcorner a = \min\{x : x \leq 1 \wedge x \cdot a = a\}$, (e) $x \leq 1 \Rightarrow \ulcorner x = x$,
 (b) $\ulcorner a \cdot a = a$, (f) $\ulcorner(a \cdot b) \leq \ulcorner(a \cdot \ulcorner b)$,
 (c) $x \leq 1 \wedge x \cdot a = a \Rightarrow \ulcorner a \leq x$, (g) $\ulcorner a = 0 \Leftrightarrow a = 0$.
 (d) $\ulcorner(a \cdot b) \leq \ulcorner a$,

According to Lemma 2.7(g) the domain of an element also decides its “definedness” if we identify 0 with $\perp$ as used in denotational semantics.

2.4 Locality of Composition

It should be noted that the converse inequation of Lemma 2.7(f) does not follow from our axiomatization. A counterexample is given in [10]. Its essence is that composition does not work “locally” in that only the domain of the right factor of a composition would decide about its definedness.

Therefore we say that a KA has *left-local composition* if it satisfies

$$\ulcorner b = \ulcorner c \Rightarrow \ulcorner(a \cdot b) = \ulcorner(a \cdot c) .$$

Right-locality is defined symmetrically. A KA has *local composition* if its composition is both left-local and right-local.

Lemma 2.8.

- (a) *A KA has left-local composition iff it satisfies $\ulcorner(a \cdot b) = \ulcorner(a \cdot \ulcorner b)$.*
 (b) *If a KA has left-local composition then $\ulcorner(\ulcorner a \cdot b) = \ulcorner a \sqcap \ulcorner b = \ulcorner a \cdot \ulcorner b$.*

Analogous properties hold for right-locality. In the sequel we only consider KAs with local composition. All examples given in Section 2.1 satisfy that property.

2.5 Type Implication

Types also play the rôle of assertions. The following operator will be instrumental in propagating assertions through compositions.

Definition 2.9. *The binary operator $\rightarrow$, called type implication, is defined as follows:*

$$a \rightarrow b \triangleq \neg \ulcorner(a \cdot \neg \ulcorner b) .$$

Hence $a \rightarrow b$ characterizes the type of points from which no computation as described by a may lead outside the domain of b . If a and b are types then a straightforward calculation shows that $a \rightarrow b = \neg a + b$, so that both the name “implication” and the symbol are justified.

This operator is closely related to the *monotype factor* as defined by Backhouse in [4]. For the case of a type t , one can interpret $a \rightarrow t$ also as $[a]t$, where $[a]$ is the modal “always” operator as used in dynamic logic (see, e.g., [11]).

Lemma 2.10. *Let t be a type.*

- (a) $1 \rightarrow b = \top b$
- (b) $(a + b) \rightarrow c = (a \rightarrow c) \cdot (b \rightarrow c)$
- (c) $a \cdot b \rightarrow c = a \rightarrow (b \rightarrow c)$ (Currying)
- (d) $(t \cdot a \rightarrow b) \cdot t = (a \rightarrow b) \cdot t$ (Modus Ponens)
- (e) $(a \rightarrow t \cdot b) \cdot a = (a \rightarrow t \cdot b) \cdot a \cdot t$ (Type Propagation)
- (f) $a \rightarrow b \cdot \top c = a \rightarrow b \cdot c$ (Domain Absorption)
- (g) $a \rightarrow t \cdot b = (a \rightarrow t) \cdot (a \rightarrow b) = (a \rightarrow b) \cdot (a \rightarrow t)$ (Weak Distributivity)

3 Fixed Points

We recall a few basic facts about fixed points.

Definition 3.1. *A function f between complete lattices is strict if $f(0) = 0$ and co-strict if $f(\top) = \top$. Further, f is called continuous if $f(\sqcup L) = \sqcup f(L)$ for every non-empty chain L , and co-continuous iff $f(\sqcap L) = \sqcap f(L)$ for every non-empty chain L .*

Every universally disjunctive function is continuous and strict; every universally conjunctive function is co-continuous and co-strict. Moreover, every continuous or co-continuous function is monotonic.

Theorem 3.2. (Knaster/Tarski and Kleene)

- (a) *A monotonic function f on a complete lower (upper) semilattice has a least (greatest) fixed point $\mu(f)$ ($\nu(f)$) provided its set $\{x : x \geq f(x)\}$ of contracted elements (its set $\{x : x \leq f(x)\}$ of expanded elements) is non-empty. These fixed points satisfy*

$$\begin{aligned}\mu(f) &= \sqcap \{x : f(x) = x\} = \sqcap \{x : f(x) \leq x\}, \\ \nu(f) &= \sqcup \{x : f(x) = x\} = \sqcup \{x : x \leq f(x)\}.\end{aligned}$$

- (b) *Every monotonic function f on a complete lattice has a least fixed point $\mu(f)$ and a greatest fixed point $\nu(f)$.*
- (c) *If f is continuous, then $\mu(f) = \sqcup \{i : i \in \mathbb{N} : f^i(0)\}$. If f is co-continuous, then $\nu(f) = \sqcap \{i : i \in \mathbb{N} : f^i(\top)\}$.*

Because we assume our KAs to be complete lattices (Definition 2.1), least and greatest fixed points of monotonic functions exist.

Definition 3.3. *Let $(X, \leq)$ be an ordered set. The order $\leq$ is extended to the set of functions on X by $f \leq g \Leftrightarrow \forall (x : x \in X : f(x) \leq g(x))$.*

Theorem 3.4. (See, e.g., [3]) *The following properties hold:*

- (a) $f \leq g \Rightarrow \mu(f) \leq \mu(g)$ (μ monotonic),
- (b) *Let g be continuous and strict. Then*
 $f \circ g = g \circ h \Rightarrow \mu(f) = g(\mu(h))$ (μ -fusion law),
- (c) $\mu(f \circ g) = f(\mu(g \circ f))$ (permutation law).

Analogous laws hold for the greatest fixed point. This can be shown more easily than by direct proof using the notion of a *dual* function.

Definition 3.5. Let f be a function on a Boolean lattice. The dual function of f , denoted $f^\#$, is defined by $f^\#(x) \triangleq f(\overline{x})$.

Lemma 3.6. Let f be an function on a Boolean lattice. If $\mu(f)$ and $\nu(f)$ exist then $\nu(f^\#) = \overline{\mu(f)}$ and $\mu(f^\#) = \overline{\nu(f)}$.

From this and Theorem 3.4, one obtains

Theorem 3.7. (a) $f \leq g \Rightarrow \nu(f) \leq \nu(g)$ (ν monotonic),
 (b) Let g be co-continuous and co-strict. Then
 $f \circ g = g \circ h \Rightarrow \nu(f) = g(\nu(h))$ (ν -fusion law),
 (c) $\nu(f \circ g) = f(\nu(g \circ f))$ (permutation law).

3.1 Finite Iteration

Two central operations in KAs are *finite iteration* and *non-empty finite iteration*, defined for every element a by

$$a^* = \mu(x :: a \cdot x + 1) \quad \text{and} \quad a^+ = \mu(x :: a \cdot x + a) .$$

By the monotonicity of $+$ and $\cdot$, and the Knaster-Tarski-Kleene Theorem 3.2, these operations are well defined. Since $+$ and $\cdot$ are even continuous, we have

$$a^* = \bigsqcup(i : i \geq 0 : a^i) \quad \text{and} \quad a^+ = \bigsqcup(i : i > 0 : a^i) ,$$

where $a^0 = 1$ and $a^{i+1} = a \cdot a^i$. This explains the name “finite iteration”. In the algebra of relations, a^* coincides with the reflexive transitive closure and a^+ with the transitive closure.

3.2 Infinite Iteration and Termination

We now introduce notions that are useful to describe the set of initial states of a program for which termination is guaranteed. These notions are the *infinite iteration* of an element, its *terminating part* and *progressive finiteness*.

Definition 3.8. The infinite iteration of an element a is $a^\omega \triangleq \nu(x :: a \cdot x)$. The terminating part of an element a , denoted $\mathcal{T}(a)$, is the complement of the infinite iteration: $\mathcal{T}(a) \triangleq \overline{a^\omega} = \mu(x :: \overline{a \cdot x})$ (by Lemma 3.6).

In the algebra of relations, the terminating part is also known as the *initial part* [19]. It is a vector, i.e., an element v satisfying $v \cdot \top = \top$, characterizing the set of points s_0 such that there is no infinite chain $s_0, s_1, s_2, \dots$, with $(s_i, s_{i+1}) \in a$, for all $i \geq 0$. In the semantics of while programs, an analogue of the terminating part will be used to represent sets of states from which no infinite looping is possible.

Definition 3.9. *An element a is said to be progressively finite iff $\mathcal{T}(a) = \top$ [19].*

In the algebra of relations, progressive finiteness of a relation R is the same as well-foundedness of $R^\sim$. By Boolean algebra, a is progressively finite iff $a^\omega = 0$.

We now list some useful properties of infinite iteration.

Theorem 3.10. *Let a and b be elements.*

- (a) $a \leq b \Rightarrow a^\omega \leq b^\omega$,
- (b) $a^* \cdot a^\omega = a^+ \cdot a^\omega = a^\omega$,
- (c) $\neg\Gamma(a^\omega) \cdot a$ is progressively finite,
- (d) If b is progressively finite and $a \leq b$ then also a is progressively finite.

Proof. (a) Immediate from monotonicity of the fixed point operators.

(b) Easy calculation.

(c) Set $q \triangleq \neg\Gamma(a^\omega) \cdot a$. Since $q \leq a$ we get $q^\omega \leq a^\omega$ and hence $\Gamma(q^\omega) \leq \Gamma(a^\omega)$. On the other hand, by Lemma 2.7(d,e),

$$\Gamma(q^\omega) = \Gamma(q \cdot q^\omega) = \Gamma(\neg\Gamma(a^\omega) \cdot a \cdot q^\omega) \leq \Gamma(\neg\Gamma(a^\omega)) = \neg\Gamma(a^\omega) .$$

So $\Gamma(q^\omega) \leq \Gamma(a^\omega) \sqcap \neg\Gamma(a^\omega) = 0$ and hence $q^\omega = 0$.

(d) Straightforward from (a).

Finally, we can give an analogue of the terminating part at the level of types:

Definition 3.11. $\mathcal{T}_r(a) \triangleq \mu(x :: a \rightarrow x)$.

By the correspondence with the modal box operator mentioned in Section 2.5, $\mathcal{T}_r(a) = \mu(x :: [a]x)$. In the propositional μ -calculus, this is known as the *halting predicate* (see, e.g., [11]). It is easy to check that $\neg\Gamma(a^\omega)$ is a fixed point of $(x :: a \rightarrow x)$. Hence,

Corollary 3.12. (a) $\mathcal{T}_r(a) \leq \neg\Gamma(a^\omega)$.

(b) $\mathcal{T}_r(a) \cdot a$ is progressively finite.

Proof. (a) is immediate from the least fixed point property of $\mathcal{T}_r(a)$. For (b), by monotonicity of ω , we get $(\mathcal{T}_r(a) \cdot a)^\omega \leq (\neg\Gamma(a^\omega) \cdot a)^\omega = 0$ using Theorem 3.10(c).

3.3 Connecting Finite and Infinite Iteration

The next theorem connects finite and infinite iteration.

Theorem 3.13. *Let $f(x) \triangleq a \cdot x + b$. Then $\mu(f) = a^* \cdot b$ and $\nu(f) = a^* \cdot b + a^\omega$.*

The proof is by fixed point fusion. For the second assertion, this depends crucially on the complete distributivity of the underlying lattice. The theorem entails the following corollary that highlights the importance of progressive finiteness in the simplification of fixed point-related properties.

Corollary 3.14. *Let again $f(x) \triangleq a \cdot x + b$. If a is progressively finite, then f has a unique fixed point, viz. $a^* \cdot b$ [3].*

To conclude this section, we study analogous iterations at the level of types.

Theorem 3.15. *Let t be a type and set $h(x) \triangleq \ulcorner(a \cdot x) + t$ and $k(x) \triangleq \neg t \cdot (a \rightarrow x)$.*

(a) $x \leq 1 \Rightarrow h(x) = \neg k(\neg x)$,	(d) $\mathcal{T}_r(a) \leq \ulcorner(a^* \cdot \neg \ulcorner a)$,
(b) $\mu(h) = \ulcorner(a^* \cdot t)$,	(e) $\mu(k) = \mathcal{T}_r(a) \cdot (a^* \rightarrow \neg t)$,
(c) $\nu(h) = \ulcorner(a^* \cdot t) + \neg \mathcal{T}_r(a)$,	(f) $\nu(k) = (a^* \rightarrow \neg t)$.

Proof. Part (a) is straightforward. Parts (b) and (e) follow by fixed point fusion, which is applicable by continuity and strictness of the $\ulcorner$ and $\cdot$ operations. Note that $h(x) \leq 1$ and $k(x) \leq 1$ for any $x \in K$. Hence, any fixed point of h or k is a type. Because the types constitute a complete Boolean algebra, one can consider h and k to be functions on the set of types for the purpose of calculating fixed points. Thus, by Part (a), h and k are dual in the sense of Definition 3.5. Lemma 3.6 gives Parts (c) and (f). For Part (d), we first show $a \rightarrow x \leq \ulcorner(a \cdot x) + \neg \ulcorner a$; the proof uses shunting, locality of composition, distributivity and Boolean algebra:

$$\begin{aligned} (a \rightarrow x) \leq \ulcorner(a \cdot x) + \neg \ulcorner a &\Leftrightarrow \neg \ulcorner(a \cdot \neg \ulcorner x) \leq \ulcorner(a \cdot x) + \neg \ulcorner a \Leftrightarrow \\ \ulcorner a \leq \ulcorner(a \cdot \ulcorner x) + \ulcorner(a \cdot \neg \ulcorner x) &\Leftrightarrow \ulcorner a \leq \ulcorner a . \end{aligned}$$

Now, using the above derivation, the monotonicity of μ , and (b), we can conclude

$$\mathcal{T}_r(a) = \mu(x :: a \rightarrow x) \leq \mu(x :: \ulcorner(a \cdot x) + \neg \ulcorner a) = \ulcorner(a^* \cdot \neg \ulcorner a) .$$

4 The Demonic Operators

4.1 Refinement Ordering

We now define a partial ordering, called the *refinement ordering*. This ordering induces a complete upper semilattice, called a *demonic semilattice*. The associated operations are demonic join ($\sqcup$), demonic meet ($\sqcap$) and demonic composition ($\circ$). Again, we generalize from the case of relation algebra to arbitrary KAs. For more details on relational demonic semantics and demonic operators, see [4,5,6,7,9,21].

Definition 4.1. *We say that an element a refines an element b [14], denoted by $a \sqsubseteq b$, iff $\ulcorner b \leq \ulcorner a \wedge \ulcorner b \cdot a \leq b$.*

It is easy to show that $\sqsubseteq$ is indeed a partial ordering.

Theorem 4.2. (a) *The partial order $\sqsubseteq$ induces a complete upper semilattice, i.e., every subset $L \subseteq K$ has a least upper bound (wrt $\sqsubseteq$) $\sqcup L$ which is called its demonic join. One has*

$$\sqcup L = (\prod(a : a \in L : \ulcorner a)) \cdot \sqcup L \quad \text{with} \quad \ulcorner(\sqcup L) = \prod(a : a \in L : \ulcorner a) .$$

(b) $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$.

(c) If a and b satisfy the condition $\top(a \sqcap b) = \top a \sqcap \top b$, their greatest lower bound, denoted $a \sqcap b$ and called their demonic meet, exists and its value is

$$a \sqcap b = (a \sqcap b) + \neg \top a \cdot b + \neg \top b \cdot a \quad \text{with} \quad \top(a \sqcap b) = \top a + \top b .$$

Otherwise, the greatest lower bound does not exist.

In relational terms, the existence condition for $\sqcap$ simply means that for each argument in the intersection of their domains, a and b have to agree in at least one result value.

4.2 Demonic Composition

Definition 4.3. Let a and b be elements. The demonic composition of a and b , denoted by $a \sqcup b$, is defined as $a \sqcup b \triangleq (a \rightarrow b) \cdot a \cdot b$.

In the algebra of relations, a pair (s, t) belongs to $a \sqcup b$ if and only if it belongs to $a \cdot b$ and there is no possibility of reaching from s , via a , an element u that does not belong to the domain of b . For example, with $a \triangleq \{(0, 0), (0, 1), (1, 2)\}$ and $b \triangleq \{(0, 0), (2, 3)\}$, one finds that $a \sqcup b = \{(1, 3)\}$; the pair $(0, 0)$, which belongs to $a \cdot b$, does not belong to $a \sqcup b$, since $(0, 1) \in a$ and 1 is not in the domain of b . Note that we assign to $\sqcup$ and $\cdot$ the same binding power.

A fundamental property is

Theorem 4.4. The demonic composition is associative.

For the next theorem we need a notion of determinacy [10].

Definition 4.5. An element a is deterministic iff $\text{CD}(a)$ holds, where

$$\text{CD}(a) \triangleq \forall(b : b \leq a : b = \top b \cdot a) \quad (\text{characterization by domain}).$$

We quote from [10]:

Lemma 4.6. All types are deterministic. If a is deterministic and $b \leq a$, then b is deterministic as well.

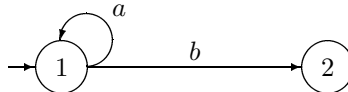
The following properties are shown by straightforward calculations.

Theorem 4.7. (a) a deterministic $\Rightarrow a \sqcup b = a \cdot b$,
 (b) $\top a \cdot \top b = 0 \Rightarrow (a + b) \sqcup c = a \sqcup c + b \sqcup c \wedge a \sqcap b = a + b$.

5 The Semantics of Nondeterministic Loops

5.1 Intuition and Notation

A general nondeterministic loop is best described by a graph of the form



It may “execute” a as long as the intermediate states remain in the domain of a and it may exit if a state in the domain of b is reached. The domains of a and b need not be disjoint. Since a may be nondeterministic, it can take a starting state s to many successor states. If among these there exists a state outside the domains of a and b (abnormal termination), then in the demonic view s must be excluded from the domain of the loop semantics. Hence, in addition to $\mathcal{T}_\Gamma(a)$, we introduce a type $\mathcal{P}(a, b)$ ($\mathcal{P}$ stands for *proper*) that characterizes the states from which no abnormal termination is possible.

We now define the corresponding semantic functions formally. Let a and b be elements. The abbreviations φ , φ_Γ , $\mathcal{P}(a, b)$ and s_μ are defined as follows:

$$\begin{aligned}\varphi(x) &\triangleq (a \rightarrow x) \cdot (a \cdot x + b), & \mathcal{P}(a, b) &\triangleq a^* \rightarrow (a + b), \\ \varphi_\Gamma(x) &\triangleq (a \rightarrow x) \cdot \ulcorner(a + b), & s_\mu &\triangleq \mathcal{P}(a, b) \cdot \mathcal{T}_\Gamma(a) \cdot a^* \cdot b.\end{aligned}\tag{1}$$

The subscript Γ in φ_Γ expresses that φ_Γ is a domain-oriented counterpart of φ .

The element s_μ , which we take as the semantics of the loop, is the restriction of the angelic loop semantics $a^* \cdot b$ to $\mathcal{P}(a, b)$ and $\mathcal{T}_\Gamma(a)$. Hence the domain of s_μ represents the set of states from which proper termination is guaranteed. We want to show that s_μ is the least fixed point of φ .

Lemma 5.1. *Assume, as in Theorem 3.13, $f(x) \triangleq a \cdot x + b$.*

- (a) *If s is a type then $\varphi(s \cdot x) = \varphi_\Gamma(s \cdot x) \cdot f(x)$.*
- (b) *$\varphi(x) = (a \rightarrow x) \cdot f(x) = \varphi_\Gamma(x) \cdot f(x) = a \sqcap x + (a \rightarrow x) \cdot b$.*
- (c) *If $\ulcorner a \cdot \ulcorner b = 0$, then $\varphi(x) = a \sqcap x \sqcap b$.*

Proof. (a) $\varphi(s \cdot x)$
 $= \llbracket \text{definitions} \rrbracket$
 $(a \rightarrow s \cdot x) \cdot (a \cdot s \cdot x + b)$
 $= \llbracket \text{distributivity and type propagation (Lemma 2.10(e))} \rrbracket$
 $(a \rightarrow s \cdot x) \cdot (a \cdot x + b)$
 $= \llbracket \text{by Lemma 2.4(b) and Lemma 2.7(b),} \rrbracket$
 $\quad \text{since } \ulcorner(a \cdot x + b) \leq \ulcorner(a + b) \rrbracket$
 $(a \rightarrow s \cdot x) \cdot \ulcorner(a + b) \cdot (a \cdot x + b)$
 $= \llbracket \text{definitions} \rrbracket$
 $\varphi_\Gamma(s \cdot x) \cdot f(x)$

(b) Immediate from (a) by setting $s = 1$ and from the definitions.

(c) This follows from Part (b) by monotonicity of $\ulcorner$, Boolean laws and Theorem 4.7(b).

Item (c) justifies why we are talking about a demonic star operator.

5.2 Properties of the Semantics

The following lemma presents the relationship between the fixed points of the functions φ and φ_Γ (Equations (1)). It is again proved by simple calculation. For (a), this involves local composition, domain absorption and type propagation.

Lemma 5.2. (a) $\ulcorner(\varphi(x)) = \varphi_\ulcorner(\ulcorner x)$.

(b) If y is a fixed point of φ then $\ulcorner y$ is a fixed point of $\varphi_\ulcorner$.

In the following we give bounds on the fixed points of $\varphi_\ulcorner$.

Theorem 5.3. (a) $\mu(\varphi_\ulcorner) = \mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner(a)$ and $\nu(\varphi_\ulcorner) = \mathcal{P}(a, b)$.

(b) If y is a fixed point of φ , then $\mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner a \leq \ulcorner y \leq \mathcal{P}(a, b)$.

Proof. Immediate from Lemma 5.2(b) and Theorem 3.15(e,f).

Corollary 5.4. $s_\mu = \mu(\varphi_\ulcorner) \cdot \mu(f)$, where $f(x) \triangleq a \cdot x + b$.

The next theorem characterizes the domain of s_μ . It is the set of points for which normal termination is guaranteed (no possibility of abnormal termination or infinite loop).

Theorem 5.5. (a) $\mathcal{P}(a, b) \leq \ulcorner(a^* \cdot b) + \neg \mathcal{T}_\ulcorner(a)$,

(b) $\mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner(a) \leq \mathcal{T}_\ulcorner(a) \cdot \ulcorner(a^* \cdot b)$,

(c) $\ulcorner s_\mu = \mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner(a) = \mu(\varphi_\ulcorner)$.

Proof. (a) $\mathcal{P}(a, b) \leq \ulcorner(a^* \cdot b) + \neg \mathcal{T}_\ulcorner(a)$

$\Leftrightarrow \{ \text{shunting} \}$

$\mathcal{T}_\ulcorner(a) \leq \ulcorner(a^* \cdot b) + \neg \mathcal{P}(a, b)$

$\Leftrightarrow \{ \text{Equations (1), Definition 2.9, Boolean algebra, Lemma 2.8} \}$

$\mathcal{T}_\ulcorner(a) \leq \ulcorner(a^* \cdot \ulcorner b) + \ulcorner(a^* \cdot \neg \ulcorner a \cdot \neg \ulcorner b)$

$\Leftrightarrow \{ \text{distributivity, Boolean algebra} \}$

$\mathcal{T}_\ulcorner(a) \leq \ulcorner(a^* \cdot (\ulcorner b + \neg \ulcorner a))$

$\Leftarrow \{ \text{monotonicity} \}$

$\mathcal{T}_\ulcorner(a) \leq \ulcorner(a^* \cdot \neg \ulcorner a)$

$\Leftrightarrow \{ \text{Theorem 3.15(d)} \}$

true

(b) Immediate from (a).

(c) $\ulcorner s_\mu$

$= \{ \text{Equations (1)} \}$

$\ulcorner(\mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner(a) \cdot a^* \cdot b)$

$= \{ \text{locality of composition (Lemma 2.8)} \}$

$\mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner(a) \cdot \ulcorner(a^* \cdot b)$

$= \{ (b), \text{ Boolean algebra (notably idempotence of } \cdot \text{ on types)} \}$

$\mathcal{P}(a, b) \cdot \mathcal{T}_\ulcorner(a)$

In the following theorem, we show that s_μ is a fixed point of φ .

Theorem 5.6. $\varphi(s_\mu) = s_\mu$.

$$\begin{aligned}
\textit{Proof.} \quad & \varphi(s_\mu) \\
&= \llbracket \text{Equations (1) and Corollary 5.4} \rrbracket \\
&\quad \varphi(\mu(\varphi_\tau) \cdot \mu(f)) \\
&= \llbracket \text{Lemma 5.1(a)} \rrbracket \\
&\quad \varphi_\tau(\mu(\varphi_\tau) \cdot \mu(f)) \cdot f(\mu(f)) \\
&= \llbracket \text{Corollary 5.4, fixed point property} \rrbracket \\
&\quad \varphi_\tau(s_\mu) \cdot \mu(f) \\
&= \llbracket \text{Equations (1), domain absorption} \rrbracket \\
&\quad \varphi_\tau(\ulcorner s_\mu \urcorner) \cdot \mu(f) \\
&= \llbracket \text{Theorem 5.5(c)} \rrbracket \\
&\quad \varphi_\tau(\mu(\varphi_\tau)) \cdot \mu(f) \\
&= \llbracket \text{fixed point property} \rrbracket \\
&\quad \mu(\varphi_\tau) \cdot \mu(f) \\
&= \llbracket \text{Corollary 5.4} \rrbracket \\
&\quad s_\mu
\end{aligned}$$

The following theorem uniquely characterizes the least fixed point of φ by a simple condition and shows that s_μ is the least fixed point of φ .

Theorem 5.7. *Recall Equations (1). For all elements a ,*

- (a) $c = \mu(\varphi) \Leftrightarrow \varphi(c) = c \wedge \ulcorner c \leq \mathcal{T}_\tau(a)$,
- (b) $\mu(\varphi) = s_\mu$.

Proof. (a) $(\Rightarrow)$ Assume $c = \mu(\varphi)$. The property $\varphi(c) = c$ then obviously follows. From Theorem 5.6 and Theorem 3.2(a), we get $c \leq s_\mu$ and hence, by Theorem 5.5(c), $\ulcorner c \leq \ulcorner s_\mu = \mathcal{P}(a, b) \cdot \mathcal{T}_\tau(a) \leq \mathcal{T}_\tau(a)$.

$(\Leftarrow)$ Assume $\varphi(c) = c$ and $\ulcorner c \leq \mathcal{T}_\tau(a)$. Theorem 5.3 implies $\mathcal{P}(a, b) \cdot \mathcal{T}_\tau(a) \leq \ulcorner c \leq \mathcal{P}(a, b)$. Hence $\ulcorner c = \mathcal{P}(a, b) \cdot \mathcal{T}_\tau(a) = \mu(\varphi_\tau)$. This is used in the following derivation, which also employs Lemma 5.1(b), domain absorption (Lemma 2.10(f)), a fixed point property and distributivity.

$$\begin{aligned}
c &= \varphi(c) = \varphi_\tau(c) \cdot f(c) = \varphi_\tau(\ulcorner c \urcorner) \cdot f(c) = \varphi_\tau(\mu(\varphi_\tau)) \cdot f(c) = \\
&\quad \mu(\varphi_\tau) \cdot f(c) = \mu(\varphi_\tau) \cdot a \cdot c + \mu(\varphi_\tau) \cdot b .
\end{aligned}$$

By Theorem 5.5(c), Corollary 3.12(b) and Theorem 3.10(d), $\mu(\varphi_\tau) \cdot a$ is progressively finite. Invoking Corollary 3.14 shows that the function $(x :: \mu(\varphi_\tau) \cdot a \cdot x + \mu(\varphi_\tau) \cdot b)$ has a unique fixed point. Thus all elements c such that $\varphi(c) = c$ and $\ulcorner c \leq \mathcal{T}_\tau(a)$ are equal. But $\mu(\varphi)$ is such an element, as we have shown above (part $\Rightarrow$). We conclude that $c = \mu(\varphi)$.

- (b) By Theorem 5.6, $s_\mu = \varphi(s_\mu)$. By Theorem 5.5(c), $\ulcorner s_\mu \leq \mathcal{T}_\tau(a)$. Now the claim is a consequence of part (a) of this theorem.

5.3 Relating Angelic and Demonic Semantics

In the following, we will show that the element s_μ is the greatest fixed point with respect to $\sqsubseteq$ of the function φ (Equations (1)). But first, we show

Lemma 5.8. *The function φ is monotonic wrt $\sqsubseteq$.*

Proof. Assume $x \sqsubseteq y$. First, using Lemma 5.2(a) and monotonicity of φ_τ wrt $\leq$,

$$\lceil \varphi(y) \leq \lceil \varphi(x) \Leftrightarrow \varphi_\tau(\lceil y) \leq \varphi_\tau(\lceil x) \Leftarrow \lceil y \leq \lceil x .$$

Second, using Lemma 5.2(a), Equations (1), Lemma 2.10(g), type propagation, the hypothesis and monotonicity of φ wrt $\leq$,

$$\begin{aligned} \lceil \varphi(y) \cdot \varphi(x) &= \varphi_\tau(\lceil y) \cdot \varphi(x) = (a \rightarrow \lceil y) \cdot \lceil (a + b) \cdot (a \rightarrow x) \cdot (a \cdot x + b) = \\ &= (a \rightarrow \lceil y \cdot x) \cdot (a \cdot \lceil y \cdot x + b) = \varphi(\lceil y \cdot x) \leq \varphi(y) . \end{aligned}$$

Recall that $\sqsubseteq$ is a complete $\sqcup$ -semilattice. Since the function φ is $\sqsubseteq$ -monotonic and s_μ is a fixed point, i.e. an expanded element, by Theorem 3.2 the $\sqsubseteq$ -greatest fixed point of the function φ exists and is given by

$$w = \bigsqcup (x :: x = \varphi(x)) . \quad (2)$$

Theorem 5.9. *The element s_μ (Equations (1)) is the $\sqsubseteq$ -greatest fixed point of φ , that is $s_\mu = \nu_{\sqsubseteq}(\varphi)$.*

Proof. Since $s_\mu = \varphi(s_\mu)$, by (2) we get $s_\mu \sqsubseteq w$. Hence, the definition of $\sqsubseteq$ implies $\lceil w \leq \lceil s_\mu$. Using Theorem 5.5(c) now gives $\lceil w \leq \mathcal{T}_\tau(a)$, so that, by Theorem 5.7(a), we finally obtain $w = \mu_{\leq}(\varphi) = s_\mu$.

In other words, the least fixed point of φ wrt $\leq$ is equal to the greatest fixed point of the same function φ wrt $\sqsubseteq$.

6 Application

In Mills's approach, the semantics w of a deterministic loop **do** $g \rightarrow \mathbf{C}$ **od** is given as the least fixed point (wrt $\leq$) of the function

$$w_{gc}(x) \triangleq g \cdot c \cdot x + \neg g ,$$

where the type g is the semantics of the loop guard g and the element c is the semantics of the loop body $\mathbf{C}$.

Lemma 6.1. *If the loop body c is deterministic, then*

$$w_{gc}(x) = (g \cdot c \rightarrow x) \cdot (g \cdot c \cdot x + \neg g) = g \sqcap c \sqcap x \sqcap \neg g .$$

Hence, in this case, the demonic and angelic semantics coincide, as expected. Moreover, under mild additional assumptions on the underlying KA, the semantics of the loop is a deterministic element as well.

Calculating the semantics of a loop is difficult, but *showing the correctness* of any candidate element is much easier. For deterministic programs, Mills [15,16] has described a checking method known as the *while statement verification rule*.

In a nondeterministic context, the abstraction is calculated by considering the worst behavior of the program (*demonic semantics*) [21]. Given a loop guard and a loop body, Theorem 5.7 (with $a \triangleq g \cdot c$ and $b \triangleq \neg g$; notice that $\lceil a \cdot \lceil b = 0 \rceil$) can be used to verify if an element w is indeed the semantics of the loop.

The following example is rather contrived, but it is simple and fully illustrates the various cases that may happen. Consider the following loop, where the variable n ranges over the set of integers [7,22]:

Example 6.2. Consider the program

```

do  $n > 0 \rightarrow$  if  $n = 1 \rightarrow n := 1 \quad \sqcup \quad n = 1 \rightarrow n := -3$ 
       $\sqcup \quad n = 3 \rightarrow n := 2 \quad \sqcup \quad n = 3 \rightarrow n := -1$ 
       $\sqcup \quad n \geq 4 \rightarrow n := n-4$ 
fi od

```

Notice that all $n > 0$ such that $n \bmod 4 = 1$ may lead to termination with a final value $n' = -3$, but may also lead to an infinite loop via the value $n = 1$; therefore these initial values of n do not belong to the domain of the element w that is the demonic semantics of the loop. Note also that all $n > 0$ such that $n \bmod 4 = 3$ may lead to termination with a final value $n' = -1$, but may also lead to a value $n = 2$, for which the loop body is not defined (by the semantics of **if fi**); these n do not belong to the domain of w . Because they also lead to $n = 2$, all $n > 0$ such that $n \bmod 4 = 2$ do not belong to the domain of w .

The semantics of the loop guard in the concrete KA REL is given by:

$$g = \{n > 0 \wedge n' = n\} \quad (\text{whence } \neg g = \{n \leq 0 \wedge n' = n\}).$$

The semantics of the loop body is:

$$\begin{aligned}
c = & \{n = 1 \wedge n' = n\} \sqcup \{n' = 1\} \sqcup \{n' = -3\} \\
& \sqcup \{n = 3 \wedge n' = n\} \sqcup \{n' = 2\} \sqcup \{n' = -1\} \\
& \sqcup \{n \geq 4 \wedge n' = n\} \sqcup \{n' = n - 4\}.
\end{aligned}$$

By Theorem 4.7(a), $g \sqcup c = g \cdot c = c$. Using Theorem 5.7(a), we show that

$$w \triangleq \{(n \leq 0 \wedge n' = n) \vee (n > 0 \wedge n \bmod 4 = 0 \wedge n' = 0)\}$$

is the semantics of the loop. The condition $\varphi(w) = w$ of Theorem 5.7(a) follows from straightforward calculations. The second condition $\lceil w \leq \mathcal{T}_r(g \cdot c)$ can be established informally by noting that the domain of w is $\{n \leq 0 \vee n \bmod 4 = 0\}$, and that there is no infinite sequence by $g \cdot c$ for any n in the domain of w .

A more satisfactory way to show $\lceil w \leq \mathcal{T}_r(g \cdot c)$ is to calculate $\mathcal{T}_r(g \cdot c)$. However, because $\mathcal{T}_r(g \cdot c)$ characterizes the domain of guaranteed termination of the associated loop, there is no systematic way to compute it (this would solve the halting problem). To demonstrate termination of the loop from every state in the domain of w , classical proofs based on variant functions or well-founded sets could be given. But formal arguments based on the definition of the terminating part (Definition 3.8) can also be used [7].

In this example, Theorem 5.7 was used to *verify* that the guessed semantics w of the loop was correct, given the semantics g of the loop guard and c of the loop body. The theorem can also be used in the other direction. If we are given a specification w , we can guess g and c , and then apply Theorem 5.7 to verify the correctness of the guess. If it is correct, then a loop of the form $\mathbf{do\ } g \rightarrow \mathbf{C\ } \mathbf{od}$, where $\mathbf{C}$ is an implementation of c , is correct with respect to w .

Acknowledgments. The authors thank Thorsten Ehm and Dexter Kozen for helpful comments. This research was supported by FCAR (Fonds pour la Formation de Chercheurs et l'Aide à la Recherche, Québec) and NSERC (Natural Sciences and Engineering Research Council of Canada).

References

1. C. J. Aarts. Galois connections presented calculationally. Eindhoven University of Technology, Dept. of Mathematics and Computer Science, July 1992.
2. R. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.
3. R. C. Backhouse *et al.* Fixed point calculus. *Inform. Proc. Letters*, 53:131–136, 1995.
4. R. C. Backhouse and J. van der Woude. Demonic operators and monotype factors. *Mathematical Structures in Comput. Sci.*, 3(4):417–433, 1993.
5. R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *Theoret. Comput. Sci.*, 43:123–147, 1986.
6. N. Boudriga, F. Elloumi, and A. Mili, On the lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544–571, 1992.
7. C. Brink, W. Kahl, and G. Schmidt (eds). *Relational Methods in Computer Science*. Springer, 1997.
8. J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.
9. J. Desharnais, N. Belkhit, S. B. M. Sghaier, F. Tchier, A. Jaoua, A. Mili, and N. Zaguia. Embedding a demonic semilattice in a relation algebra. *Theoret. Comput. Sci.*, 149(2):333–360, 1995.
10. J. Desharnais and B. Möller. Characterizing functions in Kleene algebras. In J. Desharnais (ed.), *Proc. 5th Seminar on Relational Methods in Computer Science (RelMiCS'5)*. Université Laval, Canada, pages 55–64, 2000.
11. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. Forthcoming book.
12. E. Hehner. Predicative programming, Parts I and II. *CACM*, 27:134–151, 1984.
13. D. Kozen. Kleene algebra with tests. *ACM TOPLAS* 19(3):427–443, 1997.
14. A. Mili, J. Desharnais, and F. Mili. Relational heuristics for the design of deterministic programs. *Acta Inform.*, 24(3):239–276, 1987.
15. H. D. Mills. The new math of computer programming. *CACM*, 18(1):43–48, 1975.
16. H. D. Mills, V. R. Basili, J. D. Gannon and R. G. Hamlet. *Principles of Computer Programming. A Mathematical Approach*. Allyn and Bacon, Inc., 1987.
17. B. Möller. Typed Kleene algebras. Universität Augsburg, Institut für Informatik, Report, 1999
18. T. S. Norvell. Predicative semantics of loops. In R. S. Bird and L. Meertens (eds), *Algorithmic Languages and Calculi*, Chapman & Hall, 1997, pages 415–437.

19. G. Schmidt and T. Ströhlein. *Relations and Graphs*. EATCS Monographs in Computer Science, Springer-Verlag, Berlin, 1993.
20. E. Sekerinski. A calculus for predicative programming. *Second International Conf. on the Mathematics of Program Construction*. R. S. Bird, C. C. Morgan and J. C. P. Woodcock (eds), Oxford, June 1992, *Lect. Notes in Comput. Sci.*, Vol. 669, Springer-Verlag, 1993.
21. F. Tchier. Sémantiques relationnelles démoniaques et vérification de boucles non-déterministes. Ph.D. Thesis, Département de Mathématiques, Université Laval, Canada, 1996.
22. F. Tchier and J. Desharnais. Applying a generalization of a theorem of Mills to generalized looping structures. *Colloquium Science and Engineering for Software Development*, organized in the memory of Dr. Harlan D. Mills, and affiliated to the *21st International Conference on Software Engineering*, Los Angeles, 18 May 1999, pages 31–38.

Pointwise Relational Programming

Oege de Moor and Jeremy Gibbons

Programming Research Group, Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom

Abstract. The point-free relational calculus has been very successful as a language for discussing general programming principles. However, when it comes to specific applications, the calculus can be rather awkward to use: some things are more clearly and simply expressed using variables. The combination of variables and relational combinators such as converse and choice yields a kind of nondeterministic functional programming language. We give a semantics for such a language, and illustrate with an example application.

1 Introduction

The following question appeared on a first year exam in algorithm design:

The data type of trees is given by

$$\text{data } \textit{Tree } a \quad = \quad \textit{Tip } a \mid \textit{Bin } (\textit{Tree } a) \, a \, (\textit{Tree } a) .$$

That is, a tree is a tip labelled with an element, or it consists of node that contains two subtrees and a label. A tree is said to be a *search tree* if its inorder traversal is strictly increasing. Design an algorithm that, given the preorder traversal of a search tree t , returns the tree t itself.

The original statement of the question also asked candidates for efficient algorithms to perform inorder and preorder traversals. At the time the exam was set, this seemed a reasonable question. The solution had not been covered in the lectures, but students had seen similar algorithms for tree building. In the exam itself, however, no one got the answer right, so apparently this kind of question is too hard. That is discouraging, especially in view of the highly sophisticated problems that the same students can solve in mathematics exams. Why is programming so much harder?

In computer science, students are generally not provided with a coherent body of theorems that guide them to problem solutions. This contrasts with mathematics, where students do get a number of strategic theorems and subsidiary results that provide beacons in the quest for solutions. The program derivation community has for some time been aware of this difficulty, and worked on finding a body of theorems that aids the design of algorithms, and that encapsulates common problem solving techniques in precise mathematical facts [17, 19, 29]. In particular, the point free relational calculus has proved extraordinarily successful in the derivation of general programming principles [1, 4, 9,

10, 13, 14, 21, 20]. The calculus is said to be *point free* because it does not name the data values: instead, one manipulates programs and specifications through a set of combining forms, much in the style originally suggested by Backus [5] for functional programming. As the above references demonstrate, this style is appropriate for proving generic results. When applied to specific examples, however, it can be somewhat painful to use. Properties that are tacitly applied in a pointwise style become opaque when expressed using combinators. For example, when applying associativity of list concatenation

$$(x \mathbin{++} y) \mathbin{++} z = x \mathbin{++} (y \mathbin{++} z)$$

in a pointwise style, one simply leaves out the brackets. One cannot do the same in the point free statement (writing *cat* for $(++)$ and *assoc* for the re-associating isomorphism)

$$cat \cdot (cat \times id) = cat \cdot (id \times cat) \cdot assoc .$$

As argued by Backhouse [3], silent application of ubiquitous properties such as associativity allows one to concentrate on the important mathematical facts, without being distracted by excess formality. It is our belief that the relational calculus (and the programming principles expressed in it) is unusable for program derivation in its present form, because it does distract from the essence of particular problems.

In this paper, we explore the suggestion that it may be preferable to conduct relational derivations of concrete programs in a pointwise calculus of nondeterministic mappings. The presentation is largely informal: we introduce concepts and notations through an example that is similar to the above exam question, but a little harder. Ideally, this slightly more complicated example would have been covered in lectures. By learning the general results in that derivation, students would find the above exam question easy, and we invite the reader to test our hypothesis by solving the exam question after reading this paper.

The semantics of nondeterministic mappings is a nettlesome topic, and it behoves us to outline the semantics we have in mind. It will come as no surprise that the basis is a translation from nondeterministic expressions to relational calculus, and the change from traditional relational calculus is therefore mostly notational.

2 A Programming Problem

Consider the problem of constructing a heap from a given sequence x , so that the inorder traversal of the heap is x itself. We shall derive a linear time algorithm that was first suggested by Robert Floyd in the context of precedence parsing [18].

Trees. A *tree* is either empty, or it consists of two subtrees and an element. In the notation of Haskell [12], we can write

$$\text{data Tree } a = \text{Null} \mid \text{Bin (Tree } a) a (\text{Tree } a) .$$

Inorder Traversal. The *inorder traversal* of a tree is returned by the function $\text{flatten} :: \text{Tree } a \rightarrow [a]$:

$$\begin{aligned}\text{flatten } \text{Null} &= [] \\ \text{flatten } (\text{Bin } s \ a \ t) &= \text{flatten } s \ ++ \ [a] \ ++ \ \text{flatten } t .\end{aligned}$$

Heaps. A tree is said to be a *heap* if the label of each node is at most the elements below it. To wit, define the predicate $\text{heap} :: \text{Tree } a \rightarrow \text{Bool}$ by

$$\begin{aligned}\text{heap } \text{Null} &= \text{True} \\ \text{heap } (\text{Bin } s \ a \ t) &= \text{heap } s \ \wedge \ \text{heap } t \ \wedge \ \text{above } a \ s \ \wedge \ \text{above } a \ t .\end{aligned}$$

The predicate $\text{above} :: a \rightarrow \text{Tree } a \rightarrow \text{Bool}$ checks that a is at most the label of its descendant:

$$\begin{aligned}\text{above } a \ \text{Null} &= \text{True} \\ \text{above } a \ (\text{Bin } s \ b \ t) &= a \leq b .\end{aligned}$$

Note that we do *not* require heaps to be balanced.

Specification. Our aim is to construct a total function build that is a right-inverse of flatten , and whose result is a heap. Formally, $\text{build} :: [a] \rightarrow \text{Tree } a$ should satisfy

$$\text{flatten}(\text{build } x) = x \quad \text{and} \quad \text{heap}(\text{build } x) .$$

We find it convenient to write the definition of build as a composition of *nondeterministic mappings*. For the purpose of this example, a nondeterministic mapping is merely another way of denoting a binary relation. The mapping $\text{flatten}^\circ :: [a] \rightarrow \text{Tree } a$ takes a list x , and returns some tree whose inorder traversal is x . Note that flatten° is nondeterministic, because it has the choice between many different trees. The mapping $\text{heap}^? :: \text{Tree } a \rightarrow \text{Tree } a$ is a subset of the identity relation, only mapping a tree t to itself if t is a heap. Thus, $\text{heap}^?$ is partial. Our problem is to construct a refinement build of the composition $\text{heap}^? \cdot \text{flatten}^\circ$:

$$\text{build} \sqsubseteq \text{heap}^? \cdot \text{flatten}^\circ .$$

This specification does not state the additional requirement that build is a total function.

Our specification makes use of two operators that are generally useful in the sequel. The *converse* of a mapping $f :: A \rightarrow B$ simply swaps the pairs in the binary relation that f denotes, and we write $f^\circ :: B \rightarrow A$. Given a total function $p :: A \rightarrow \text{Bool}$, the *test* of p is a subset $p^?$ of the identity relation consisting of precisely those elements that satisfy p .

2.1 Representing Trees by Spines

One could try and derive a solution directly from the above specification, but this will lead to a quadratic time algorithm. We therefore seek a different representation of trees. The choice of such a representation is the main inventive step in our derivation.

Both in algorithm design and in the study of programming language interpreters, it is very common to represent a tree by its *spine*. The type of spines is given by the synonym

$$\text{type Spine } a = [(a, \text{Tree } a)] .$$

A spine is turned into a tree by rolling it up from the left. That is, the function $\text{roll} :: \text{Spine } a \rightarrow \text{Tree } a$ is defined by

$$\begin{aligned} \text{roll} &= \text{foldl } \text{bin } \text{Null} \\ &\quad \text{where } \text{bin } s (a, t) = \text{Bin } s a t . \end{aligned}$$

The function roll is in fact a bijection. Here $\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$ is the function that takes a binary operator, a seed value and a list, and sums the list from left to right, starting with the given seed value.

Before proceeding, it is useful to introduce a dual of foldl , named foldr . Its type is $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$, and foldr step seed sums a list from right to left, using the operator step , and starting with the given value seed .

Having made the decision to represent trees by spines, we can sketch a possible derivation, making two wishful steps that will need further elaboration in the sequel:

$$\begin{aligned} &\text{heap?} \cdot \text{flatten}^\circ \\ = &\quad \{\text{since } \text{roll} \text{ is bijective}\} \\ &\text{heap?} \cdot \text{roll} \cdot \text{roll}^\circ \cdot \text{flatten}^\circ \\ = &\quad \{\text{since converse is contravariant}\} \\ &\text{heap?} \cdot \text{roll} \cdot (\text{flatten} \cdot \text{roll})^\circ \\ = &\quad \{\text{WISH: inverting } \text{flatten} \cdot \text{roll}\} \\ &\text{heap?} \cdot \text{roll} \cdot \text{foldr } \text{add } [] \\ = &\quad \{\text{define } \text{hp} = \text{heap} \cdot \text{roll}\} \\ &\text{roll} \cdot \text{hp?} \cdot \text{foldr } \text{add } [] \\ \sqsubseteq &\quad \{\text{WISH: fusion}\} \\ &\text{roll} \cdot \text{foldr } \text{join } [] \end{aligned}$$

Note that all steps are equalities except for the last one, which is a refinement. We only reduce the amount of nondeterminism by choosing a specific tree that has the heap property. If we can fulfill the wishes in the above calculation, and join is a total function, we can set

$$\text{build} = \text{roll} \cdot \text{foldr } \text{join } [] ,$$

and our problem is solved.

Let us now consider the two unproven steps in some more detail. In the first wishful step, we assumed the existence of a mapping add such that

$$(flatten \cdot roll)^\circ = foldr\ add\ [] .$$

The second wishful step aims to promote the test for heapiness of a spine $hp?$, so that

$$hp? \cdot foldr\ add\ [] \sqsubseteq foldr\ join\ [] .$$

To discharge this requirement, we shall have to synthesise a total function $join$ that satisfies the above containment.

2.2 First Wishful Step: Inverting $flatten \cdot roll$

Abbreviate $flar = flatten \cdot roll$. Before inverting $flar$, it is worthwhile to examine it a bit further. To start with, note the type:

$$flar :: Spine\ a \rightarrow [a] .$$

To flatten a spine, we could first flatten its components, and then concatenate the result. That is,

$$flar = concat \cdot map\ flat ,$$

where $flat :: (a, Tree\ a) \rightarrow [a]$ flattens a single spine component:

$$flat\ (a, t) = a : flatten\ t .$$

It now follows that $flar$ distributes over $[]$ and $(++)$ as follows:

$$\begin{aligned} flar\ [] &= [] \\ flar\ (x ++ y) &= flar\ x ++ flar\ y . \end{aligned}$$

This concludes our little exploration of $flar$ itself; we now turn to its inversion.

Converse Function Theorem [13] *Let $f :: A \rightarrow [B]$ be a total function. Let $e :: A$ and $(\oplus) :: B \rightarrow A \rightarrow A$ be mappings that are jointly surjective onto A . Furthermore assume that*

$$f\ e \sqsupseteq [] \text{ and } f\ (a \oplus x) \sqsupseteq a : f\ x .$$

Then we have $f^\circ = foldr\ (\oplus)\ e$.

To say that two mappings are *jointly surjective* onto A is to say that the union of their ranges is A . To apply the above result to $f = flar$, we start with the right hand side of the last condition, that is $a : flar\ x$. Without assuming anything about the structure of x , it is difficult to make progress, so let us say that

$x = y \mathbin{++} z$, for some y and z . This is no restriction, because concatenation is surjective. We now calculate:

$$\begin{aligned}
 & a : \textit{flar} (y \mathbin{++} z) \\
 = & \{ \textit{flar} \text{ distributes over } (++) \} \\
 & a : \textit{flar} y \mathbin{++} \textit{flar} z \\
 = & \{ \text{definition of } \textit{flar} \} \\
 & a : \textit{flatten}(\textit{roll} y) \mathbin{++} \textit{flar} z \\
 = & \{ \text{definition of } \textit{flat} \text{ (in reverse)} \} \\
 & \textit{flat}(a, \textit{roll} y) \mathbin{++} \textit{flar} z \\
 = & \{ \text{since } \textit{flar} = \textit{concat} \cdot \textit{map} \textit{flat} \} \\
 & \textit{flar}((a, \textit{roll} y) : z)
 \end{aligned}$$

It follows that for $a \oplus (y \mathbin{++} z) = (a, \textit{roll} y) : z$, the condition $\textit{flar} (a \oplus x) \sqsubseteq a : \textit{flar} x$ is satisfied. The other conditions of the converse function theorem are trivially satisfied, and so we conclude that

$$\begin{aligned}
 \textit{flar}^\circ & = \textit{foldr} \textit{add} [] \\
 \textit{add} a (y \mathbin{++} z) & = (a, \textit{roll} y) : z .
 \end{aligned}$$

Some readers will raise their eyebrows at our loose use of non-injective patterns in definitions, such as $(y \mathbin{++} z)$. Indeed, we could have written the following instead, using more conventional notation:

$$\begin{aligned}
 \textit{add} a \, yz & = (a, t) : z \\
 & \text{where } (t, z) = \textit{roll1} (\textit{split} \, yz) .
 \end{aligned}$$

Here $\textit{roll1}(y, z) = (\textit{roll} y, z)$, and $\textit{split}$ is the converse of concatenation, that is $(\textit{uncurry}(++))^\circ$.

The remainder of the derivation is completely routine for those who are familiar with an algebraic style of programming, and we have only spelt out the details for the benefit of those who wish to learn about this style. Readers may wish to skip to the next section at this point, and only return to the full details of the derivation later.

2.3 Second Wishful Step: Promoting $hp?$

Our next task is to promote the test for heapiness into the above expression for $\textit{flar}^\circ$. Recall the exact statement of that proof obligation:

$$hp? \cdot \textit{foldr} \textit{add} [] \sqsubseteq \textit{foldr} \textit{join} [] ,$$

where $\textit{join}$ is a total function. Instead of directly deriving a total function, we shall first find a total mapping $\textit{step}$ such that

$$hp? \cdot \textit{foldr} \textit{add} [] \sqsubseteq \textit{foldr} \textit{step} [] .$$

Then, in a later subsection, we shall refine $\textit{step}$ to a total function $\textit{join}$.

Synthesis of *step*. Our starting point is the well-known

Fusion Theorem [1] *Let $f : B \rightarrow A$, $(\oplus) :: C \rightarrow B \rightarrow B$, $e :: B$, $(\otimes) :: C \rightarrow A \rightarrow A$, and $d :: A$ be mappings. Then*

$$f \cdot \text{foldr}(\oplus) e \sqsubseteq \text{foldr}(\otimes) d ,$$

provided $f(a \oplus x) \sqsubseteq a \otimes f x$, and $f e \sqsubseteq d$.

To prove the above containment, it therefore suffices to find a mapping *step* so that

$$hp?(add\ a\ yz) \sqsubseteq step\ a\ (hp?\ yz) .$$

Clearly, in order to do so, we shall need a number of properties of *hp*. Recall the definition $hp = heap \cdot roll$. It is easy to verify that

$$\begin{aligned} hp [] &= True \\ hp ((a, t) : z) &= heap\ t \ \wedge \ hp\ z \ \wedge \ bet\ a\ (t, z) . \end{aligned}$$

Here $bet\ a\ (t, z)$ checks whether a lies between t and z . That is:

$$\begin{aligned} bet\ a\ (t, z) &= above\ a\ t \ \wedge \ fit\ a\ z \\ &\quad \text{where } fit\ a\ [] = True \\ &\quad \quad fit\ a\ ((b, t) : u) = b \leq a . \end{aligned}$$

Armed with these facts, a straightforward calculation shows that we have the desired containment

$$hp?(add\ a\ yz) \sqsubseteq step\ a\ (hp?\ yz) .$$

for the following definition of *step*:

$$\begin{aligned} step\ a\ yz &= (a, t) : x \\ &\quad \text{where } (t, x) = (bet\ a)?(roll\ y, z) \\ &\quad \quad \text{where } (y \uplus z) = yz . \end{aligned}$$

Again the use of a non-injective pattern could have been avoided at the expense of a little perspicuity:

$$\begin{aligned} step\ a\ yz &= (a, t) : z \\ &\quad \text{where } (t, z) = (bet\ a)?(roll1\ (split\ yz)) \end{aligned}$$

with the auxiliary definitions

$$\begin{aligned} roll1\ (y, z) &= (roll\ y, z) \\ split &= (\lambda(x, y).x \uplus y)^\circ . \end{aligned}$$

Refining *step* to *join*. To complete our derivation, we need to refine the above definition of *step* to a total function *join*. We first concentrate on finding an effective definition of *roll1* · *split*. Write f^* for the reflexive transitive closure of a mapping f .

Leap Frog Lemma for Closure [13] *Let $f :: A \rightarrow B$ and $g :: A \rightarrow A$ and $h : B \rightarrow B$ be mappings. If $f \cdot g = h \cdot f$, then*

$$f \cdot g^* = h^* \cdot f .$$

We have

$$\textit{split} \, yz = \textit{shift}^*([], yz) ,$$

where $\textit{shift}(y, [a] \uplus z) = (y \uplus [a], z)$. Note that

$$\textit{roll1} \cdot \textit{shift} = \textit{shunt} \cdot \textit{roll1}$$

where $\textit{shunt}(s, [(a, t)] \uplus z) = (\textit{Bin} \, s \, a \, t, z)$. By the above leap frog property, we conclude

$$\textit{roll1}(\textit{split} \, yz) = \textit{shunt}^*(\textit{roll1}([], yz)) = \textit{shunt}^*(\textit{Null}, yz) .$$

In words, one can generate all possible results of $\textit{roll1}(\textit{split} \, yz)$ by repeated shunting.

In the definition of *step*, we wished to have a result of $\textit{roll1}(\textit{split} \, yz)$ that satisfies the predicate *bet a*. Arbitrarily, we chose the first such result encountered during the shunting, thus obtaining the function $\textit{join} \sqsubseteq \textit{step}$:

$$\begin{aligned} \textit{join} \, a \, yz &= (a, t) : x \\ &\text{where } (t, x) = \textit{search}(\textit{Null}, yz) \\ &\quad \textit{search}(t, x) = \text{if } \textit{bet} \, a \, (t, x) \\ &\quad \quad \text{then } (a, t) : x \\ &\quad \quad \text{else } \textit{search}(\textit{shunt}(t, x)) . \end{aligned}$$

Now consider whether this definition of *join* is indeed a total function. It is certainly single-valued. To see that it is also total, note that if there are no elements in the spine where a fits in between, then either a is strictly smaller than all elements in the spine, or it is strictly greater. If a is strictly smaller, it can be inserted at the end of the spine, and if it is strictly greater, it can be inserted at the head.

The full program that we derived is included in an appendix. To see that it is linear time, note that after a node has been shunted, it is never considered again. It follows that if we sum the cost of the *join* operation over the whole duration of the algorithm, the total cost is proportional to the number of nodes, and hence linear.

2.4 Afterthoughts

Looking back at the derivation, an appeal to the Converse Function Theorem was not really necessary here. In the inversion of *flar*, we could simply have proved an inclusion using the Fusion Theorem in lieu of the Converse Function Theorem. That would however have been unsatisfactory. With hardly any additional proof effort, the Converse Function Theorem shows exactly where the reduction of indeterminacy occurs, namely in the choice of a particular heap, and not in the inversion of *flar*.

Although the derivation *per se* is not the main concern of this paper, the reader may wish to compare our argument with those given in [2, 11, 15, 22, 28], all of whom consider the same (or a similar) problem. We believe that our derivation proceeds in greater strides, all of which can be applied to other inversion problems. In particular, they apply to the exam question quoted at the beginning of this paper. That problem is easier than the one we just covered, because it is not necessary to consider the change of representation from trees to spines.

3 Semantics

So far we have been entirely informal about the meaning of our expressions, relying on the reader's intuition about nondeterministic mappings. It is well known that there are difficult semantical problems in dealing with nondeterminism and higher order functions, and we ought to address these at least to the extent that the above derivation can be fully justified. Our starting point is the standard semantics of λ -expressions in cartesian closed categories. It will be useful if the reader has some knowledge of elementary category theory [6].

For the purpose of this paper, one can think of a category as a universe of typed specifications: it consists of objects (types) and arrows (specifications or programs). Each arrow h has a source type A and a target type B : we write $h : A \rightarrow B$ to indicate this type information. For each object A there is a distinguished arrow $id : A \rightarrow A$. Arrows can be composed, subject to some typing rules. That is, when $f : A \rightarrow B$ and $g : B \rightarrow C$, their composite is $g \cdot f : A \rightarrow C$. Composition is associative and has identity element id .

The canonical example of a category is *Fun*, where the objects are sets and the arrows are total functions. Another example is *Rel*, where the objects are also sets, but where the arrows are binary relations, *i.e.* sets of pairs, composed in the usual manner.

3.1 Cartesian Closed Categories

A *binary functor* on a category is a binary operation $\oplus$ on objects and on arrows, which respects typing ($f \oplus g : A \oplus C \rightarrow B \oplus D$ when $f : A \rightarrow B$ and $g : C \rightarrow D$), identities ($id \oplus id = id$) and composition ($(f \cdot g) \oplus (h \cdot j) = (f \oplus h) \cdot (g \oplus j)$).

A category *has finite products* if it has a binary functor $\times$, with arrows $fst : A \times B \rightarrow A$ and $snd : A \times B \rightarrow B$, where for arrows f and g with a common

source there is a unique arrow h such that $fst \cdot h = f$ and $snd \cdot h = g$. This unique h is written $\langle f, g \rangle$. In the category *Fun*, cartesian product of sets with parallel composition of functions is a categorical product.

A *cartesian closed* category is a category with finite products and a binary functor $\Rightarrow$ where there exists an isomorphism between arrows of type $A \times B \rightarrow C$ and arrows of type $A \rightarrow (B \Rightarrow C)$. In the category *Fun*, $B \Rightarrow C$ is the set of functions from B to C . The two sides of the isomorphism are named *curry* and *uncurry* respectively.

It is useful to define *evaluation* by

$$\begin{aligned} eval & : (A \Rightarrow B) \times A \rightarrow B \\ eval & = uncurry \, id . \end{aligned}$$

Note that we have $uncurry \, f = eval \cdot (f \times id)$.

Semantics of λ -Calculus. Each cartesian closed category determines a semantics for typed λ -calculi. The way this works is by mapping primitive types to the objects of the category, and then defining mappings from composite types to objects, and from expressions to the arrows of the category. Thorough introductions to the subject may be found in [16, 23].

To be precise, if we have a mapping *obj* from primitive types to objects, we extend the translation to composite types as follows (1 stands for the nullary product):

$$\begin{aligned} funobj \, t & = obj \, t \\ funobj \, 1 & = 1 \\ funobj \, (S \times T) & = funobj \, S \times funobj \, T \\ funobj \, (S \rightarrow T) & = funobj \, S \Rightarrow funobj \, T \end{aligned}$$

We stress once more that *funobj* maps type expressions to objects of the category *Fun*. Hence, although we use the same symbol $\times$ on both sides of the equation, their meanings are quite different.

To give a meaning to closed λ -expressions, we map expressions of type T to arrows of type $1 \rightarrow funobj \, T$. The translation is in terms of a function *fun* that takes the expression to be translated, as well as a list of free variables as its arguments:

$$\begin{aligned} funtrans \, E & = fun \, E \, [] \\ fun \, x \, xs & = snd \cdot fst^i, \text{ where } xs_i \text{ is the first occurrence of } x \text{ in } xs \\ fun \, c \, xs & = funcon \, c \cdot fst^n, \text{ where } n \text{ is the length of } xs \\ fun \, (F \, E) \, xs & = eval \cdot \langle fun \, F \, xs, fun \, E \, xs \rangle \\ fun \, (\lambda x. E) \, xs & = curry \, (fun \, E \, (x : xs)) \end{aligned}$$

Given a list xs of variables $[x_0 : T_0, x_1 : T_1, \dots, x_{n-1} : T_{n-1}]$, and $E : T$, the type of $fun \, E \, xs$ is

$$(((1 \times funobj \, T_{n-1}) \times funobj \, T_{n-2}) \dots \times funobj \, T_0) \rightarrow funobj \, T$$

Note that in specifying the translation of constant expressions, we have assumed the existence of an operator *funcon* that maps each constant $(c : T)$ to its interpretation as an arrow $1 \rightarrow \text{funobj } T$. When the above translation is applied in the category of sets and total functions, it gives the standard semantics where λ -expressions are read as total functions.

3.2 Relations

Our aim, however, is to read λ -expressions as arrows in *Rel*, not *Fun*. Furthermore, we would like to extend the above translation to expressions that involve nondeterministic choice, intersection and arbitrary patterns in definitions. An obvious strategy presents itself: find appropriate new definitions of *curry* and *uncurry* (and hence of *eval*), and use these instead of the usual functional definitions.

A Failed Attempt to Define Exponents. The first problem we encounter in such an attempt is that pairing is not a product in *Rel*. There is an obvious definition taking a pair of relations $R : A \rightarrow B$ and $S : A \rightarrow C$ to $\langle R, S \rangle : A \rightarrow B \times C$:

$$(b, c)\langle R, S \rangle a \equiv bRa \wedge cRa .$$

But it is not the case that

$$\text{fst} \cdot \langle R, S \rangle = R .$$

Take, for example, $R = \text{id}$ and $S = \emptyset$.

We could attempt to carry on nevertheless, and look for an appropriate equivalent of $B \Rightarrow C$ in *Rel*. That is, we seek a definition of $(\sim)$ so that relations of type $A \times B \rightarrow C$ are in one-to-one correspondence with relations of type $A \rightarrow (B \sim C)$. The obvious choice is

$$B \sim C = B \times C .$$

It follows that we have an isomorphism curry_r and uncurry_r , and this isomorphism mimics the situation in a cartesian closed category.

We could now apply the above translation of λ -terms, using curry_r for *curry* and uncurry_r for *uncurry*. But that leads to rather unpleasant surprises. For instance, if we translate the expression

$$(\lambda f. \lambda x. f(f\ x)) \text{succ}$$

(where *succ* is the successor function) the result is the empty relation. We conclude that this is not our intended semantics.

A Successful Definition of Exponents. A *topos* is a category where relational composition is well-defined and associative (in the jargon: a *regular* category), and furthermore there exists an isomorphism between relations of type $A \rightarrow B$ and functions of type $A \rightarrow P B$. The category of sets and total functions is a topos, and there $P B$ is the set of all subsets of B . The transition from relations to set-valued functions is called *power transpose* and written

$$\Lambda R a = \{ b \mid b R a \} .$$

Using this additional structure, we define:

$$\begin{aligned} B \rightsquigarrow C &= P(B \sim C) \\ \text{curry}_p R &= \Lambda(\text{curry}_r R) \\ \text{uncurry}_p S &= \text{uncurry}_r(\in \cdot S) . \end{aligned}$$

This does not define an isomorphism, for we have $\text{uncurry}_p(\text{curry}_p R) = R$, but not in general $\text{curry}_p(\text{uncurry}_p S) = S$. It is however the case that these definitions coincide with the functional ones in a precise sense. Note that we can define a function

$$\begin{aligned} \rho &: (A \Rightarrow B) \rightarrow (A \rightsquigarrow B) \\ \rho &= \text{curry}_p \text{eval} . \end{aligned}$$

Now for any functions f and g of appropriate type we have:

$$\rho \cdot \text{curry}_p f = \text{curry}_p f \text{ and } \text{uncurry}_p(\rho \cdot g) = \text{uncurry}_p g .$$

Furthermore, we note that the conversion ρ is injective.

In analogy with the definition of *eval*, we define

$$\text{eval}_p = \text{uncurry}_p \text{id} .$$

It is easily checked that $\text{eval}_p \cdot (\rho \times \text{id}) = \text{eval}$, so again this definition coincides with the functional one on functional arguments.

3.3 Relational Semantics

Our first task is to define a mapping from types to objects of *Rel*, interpreting arrows as our newly defined exponent:

$$\begin{aligned} \text{relobj } A &= \text{obj } A \\ \text{relobj } 1 &= 1 \\ \text{relobj } (S \times T) &= \text{relobj } S \times \text{relobj } T \\ \text{relobj } (S \rightarrow T) &= \text{relobj } S \rightsquigarrow \text{relobj } T . \end{aligned}$$

As before *obj* is the interpretation of primitive types.

The relational translation is the same as that for functions, except that we use curry_p and uncurry_p in lieu of *curry* and *uncurry*:

$$\begin{aligned} \text{reltrans } E &= \text{rel } E [] \\ \text{rel } x \text{ } xs &= \text{snd} \cdot \text{fst}^i, \text{ where } xs_i \text{ is the first occurrence of } x \text{ in } xs \\ \text{rel } c \text{ } xs &= \text{relcon } c \cdot \text{fst}^n, \text{ where } n \text{ is the length of } xs \\ \text{rel } (F E) \text{ } xs &= \text{eval}_p \cdot \langle \text{rel } F \text{ } xs, \text{rel } E \text{ } xs \rangle \\ \text{rel } (\lambda x. E) \text{ } xs &= \text{curry}_p (\text{rel } E (x : xs)) . \end{aligned}$$

The function *relcon* maps each constant $c : T$ to the corresponding relation of type $1 \rightarrow \text{relobj } T$.

We can now extend the translation to expressions that did not have a semantics before. Below we discuss three examples of new expression forms: many more are possible, and the reader may wish to try and define some of her own favourites for combining specifications.

Choice. Consider $E_1 \sqcup E_2$, the nondeterministic choice between E_1 and E_2 . We define this via union of relations:

$$\text{rel } (E_1 \sqcup E_2) \, xs = (\text{rel } E_1 \, xs) \cup (\text{rel } E_2 \, xs) .$$

It is easy to check that with this definition, function application distributes through choice. In particular, we have that

$$(\lambda x. x - x) (0 \sqcup 1) = 0 ,$$

so one could say that we have a call-by-value semantics for choice.

General Patterns. As another example, consider the expression form for general pattern matching, $\lambda x : P \rightarrow E$. The idea is that P can be any expression whatsoever: patterns are not restricted to injective constructor functions. The variable x is the bound variable in the abstraction. In our example derivation, we used this device to split a list in an arbitrary way. The translation of these generalised abstractions is defined as follows:

$$\text{rel } (\lambda x : P \rightarrow E) \, xs = \text{comp} \cdot \langle \text{rel } (\lambda x. E) \, xs, \text{conv} \cdot \text{rel } (\lambda x. P) \, xs \rangle .$$

This definition uses two auxiliary functions, namely

$$\begin{aligned} \text{conv} & : (A \rightsquigarrow B) \rightarrow (B \rightsquigarrow A) \\ \text{conv} & = A(\text{swap} \cdot \in) \\ \text{comp} & : ((A \rightsquigarrow B) \times (C \rightsquigarrow A)) \rightarrow (C \rightsquigarrow B) \\ \text{comp} & = \text{curry}_p(\text{eval}_p \cdot (\text{id} \times \text{eval}_p) \cdot \text{assoc}) . \end{aligned}$$

The function *swap* : $X \times Y \rightarrow Y \times X$ swaps the components of a pair. In the definition of *comp*, the function *assoc* is the obvious rebracketing isomorphism $(X \times Y) \times Z \rightarrow X \times (Y \times Z)$.

To illustrate, consider

$$\text{rotate} = (\lambda x : (\text{fst } x \text{ ++ snd } x) \rightarrow \text{snd } x \text{ ++ fst } x) .$$

This mapping takes a list, and returns any rotation of its argument.

Fixpoints. Because *all* terms are interpreted as relations, including the higher order ones, we can take fixpoints by introducing a new constant for each object A

$$fix : (A \rightarrow A) \rightarrow A ,$$

which is interpreted as

$$\begin{aligned} relcon\ fix &: 1 \rightarrow (A \rightsquigarrow A) \rightsquigarrow A \\ relcon\ fix &= curry_p(dom \cdot cap \cdot (relid \times id)) . \end{aligned}$$

That is, we intersect the argument relation with the identity, and then take the domain of that intersection to find its fixpoints; any fixpoint of F is a possible result of $fix\ F$. The auxiliary operators in this definition are given by

$$\begin{aligned} cap &= \Lambda(\in \cdot fst \cap \in \cdot snd) \\ dom &= \Lambda(fst \cdot \in) \\ relid &= curry_p\ snd . \end{aligned}$$

Admittedly this treatment of fixpoints is a little unusual, because if there are multiple fixpoints, we simply select one of them nondeterministically. Furthermore, if there is no fixpoint of F , then $fix\ F$ is the empty relation. Of course it is also possible to define a least fixpoint operator of type

$$((A \rightarrow B) \rightarrow (A \rightarrow B)) \rightarrow (A \rightarrow B)$$

that returns a minimal fixpoint with respect to relation inclusion; the definitions are only slightly more involved than those given here.

3.4 Refinement

The possibility of nondeterminism means that we can consider refinement of relational expressions. We propose a definition modelling reduction of nondeterminism, but not enforcing preservation of domains; other definitions are possible.

Naturally, the definition involves inclusion, and the relational translation *reltrans*. For expressions $E_1, E_2 : T$ with free variables xs , we define

$$E_1 \sqsubseteq E_2 \equiv reltrans\ E_2\ xs \subseteq ref\ T \cdot reltrans\ E_1\ xs .$$

The ‘mediation’ $ref\ T$ is needed to cover higher-order cases. For example, we would like to have

$$\lambda x.x \sqcup x+1 \sqsubseteq \lambda x.x+1$$

but the relational translations of both sides (arrows $1 \rightarrow (Int \rightsquigarrow Int)$ in *Rel*) are not related simply by inclusion. So we define $ref\ T : relobj\ T \rightarrow relobj\ T$ by induction over T as follows:

$$\begin{aligned} ref\ A &= id \\ ref\ (S \times T) &= ref\ S \times ref\ T \\ ref\ (S \rightarrow T) &= \in \setminus ((id \times ref\ T) \cdot \in) \end{aligned}$$

where the operator ‘ $\setminus$ ’ (called ‘weakest prespecification’ in [20]) is defined by $(R \subseteq S \setminus T) \equiv (S \cdot R \subseteq T)$, or in points, $x\ (R \setminus S)\ y \equiv \forall z : z\ R\ x \Rightarrow z\ S\ y$.

Note that $id \subseteq ref\ T$ and $ref\ T \cdot ref\ T \subseteq ref\ T$, so $\sqsubseteq$ is a preorder. However, it is not a partial order; for example, the two expressions $(\lambda x.x \sqcup x+1)$ and $(\lambda x.x \sqcup x+1) \sqcup (\lambda x.x+1)$ refine to each other, but have different semantics.

3.5 Consistency of Functional and Relational Semantics

Our relational semantics harbours a number of surprises for the unwary. For example, recall the illustration of the choice operator given above; it is not the case that $(\lambda x.x - x)(0 \sqcup 1)$ is equal to $(0 \sqcup 1) - (0 \sqcup 1)$ — the former always evaluates to 0, whereas the latter may evaluate to -1 , 0 or 1. It follows that β -conversion is no longer a valid equality; however, we do have

$$(0 \sqcup 1) - (0 \sqcup 1) \sqsubseteq (\lambda x.x - x)(0 \sqcup 1)$$

reflecting the fact that the first expression is more nondeterministic than the second.

The validity of η -conversion

$$(\lambda x.E x) = E$$

is also compromised. Let $\text{twice} = \lambda f.f \cdot f$ and consider, for example,

$$\text{twice}(id \sqcup succ) \quad \text{and} \quad \text{twice}(\lambda x.(id \sqcup succ)x) .$$

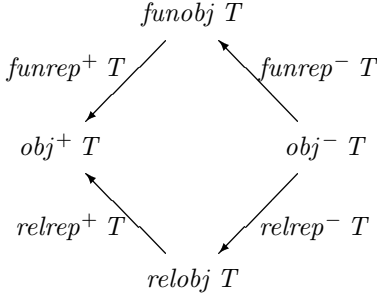
The expression on the left can evaluate to id or $succ \cdot succ$. The expression on the right, however, offers the choice between id , $succ$ and $succ \cdot succ$. Like β -reduction, η -conversion can be weakened to an inequality.

Given these mildly surprising facts, and all the additional power that a relational semantics brings, it is necessary to pin down in what sense the relational semantics coincides with the functional semantics. Intuitively, we would like to express the fact that if we feed functional inputs into the relational semantics, the result is the same as feeding those inputs into the functional semantics, and converting the result to a relation. To make this precise, we need some further terminology.

Positive and Negative Interpretation of Types. The *positive* interpretation of a type replaces every positive arrow with a relation exponent, and every negative arrow with a function exponent. Dually, the *negative* interpretation turns positive arrows into function exponents, and negative arrows into relation exponents. These two interpretations of types are defined by mutual recursion:

$$\begin{aligned} obj^+(S \rightarrow T) &= obj^- S \rightsquigarrow obj^+ T \\ obj^+(S \times T) &= obj^+ S \times obj^+ T \\ obj^+ A &= obj A \\ \\ obj^-(S \rightarrow T) &= obj^+ S \Rightarrow obj^- T \\ obj^-(S \times T) &= obj^- S \times obj^- T \\ obj^- A &= obj A \end{aligned}$$

We can now define four representation mappings to convert between the different interpretations of types that we have defined:



Intuitively, the negative occurrences of arrows are ‘inputs’ whereas the positive occurrences are ‘outputs’. One could thus think of funrep^+ as converting all the output functions into relations. The other mappings can be given similar interpretations. To define these four mappings, we shall need to use the arrow $\rho : (A \Rightarrow B) \rightarrow (A \rightsquigarrow B)$ that was defined earlier.

$$\begin{aligned}
 \text{funrep}^+ (S \rightarrow T) &= \rho \cdot (\text{funrep}^- S \Rightarrow \text{funrep}^+ T) \\
 \text{funrep}^+ (S \times T) &= \text{funrep}^+ S \times \text{funrep}^+ T \\
 \text{funrep}^+ A &= \text{id}
 \end{aligned}$$

$$\begin{aligned}
 \text{funrep}^- (S \rightarrow T) &= \text{funrep}^+ S \Rightarrow \text{funrep}^- T \\
 \text{funrep}^- (S \times T) &= \text{funrep}^- S \times \text{funrep}^- T \\
 \text{funrep}^- A &= \text{id}
 \end{aligned}$$

$$\begin{aligned}
 \text{relrep}^+ (S \rightarrow T) &= \text{relrep}^- S \rightsquigarrow \text{relrep}^+ T \\
 \text{relrep}^+ (S \times T) &= \text{relrep}^+ S \times \text{relrep}^+ T \\
 \text{relrep}^+ A &= \text{id}
 \end{aligned}$$

$$\begin{aligned}
 \text{relrep}^- (S \rightarrow T) &= (\text{relrep}^+ S \rightsquigarrow \text{relrep}^- T) \cdot \rho \\
 \text{relrep}^- (S \times T) &= \text{relrep}^- S \times \text{relrep}^- T \\
 \text{relrep}^- A &= \text{id}
 \end{aligned}$$

Finally, we define for a list of typed variables

$$\text{list } [x_0 : T_0, \dots, x_{n-1} : T_{n-1}] f = (((\text{id} \times f \ T_{n-1}) \times f \ T_{n-2}) \dots f \ T_0).$$

We can now formulate the following

Theorem. Suppose that for each constant $c : T$, we have

$$\text{relrep}^+ T \cdot \text{relcon } c = \text{funrep}^+ T \cdot \text{funcon } c,$$

and furthermore there exists an arrow $\text{mincon } c : 1 \rightarrow \text{obj}^- T$ so that

$$\begin{aligned}
 \text{funrep}^- T \cdot \text{mincon } c &= \text{funcon } c \\
 \text{relrep}^- T \cdot \text{mincon } c &= \text{relcon } c.
 \end{aligned}$$

Let $E : T$ be an expression with free variables xs that is built from variables, constants, λ -abstraction and application. Furthermore, assume that E does not contain any β -redexes. Then we have

$$\begin{aligned} & \text{funrep}^+ T \cdot \text{fun } T \text{ } xs \cdot \text{list } xs \text{ funrep}^- \\ = & \text{relrep}^+ T \cdot \text{rel } T \text{ } xs \cdot \text{list } xs \text{ relrep}^- . \end{aligned}$$

This statement formalises our requirement that the relational semantics be consistent with the functional semantics. We believe it also holds true without the restriction to β -normal form, but we were unable to complete a proof before this paper had to go to press.

4 Epilogue

We do not advocate that the point free relational calculus be abandoned. Indeed, results such as the Converse Function Theorem are most conveniently proved, for all datatypes at once, in a point free style. Specific instantiations of generic results (in a point free style) tend to be painful however. The technical reason is that properties such as associativity, which we silently apply in a pointwise style, become exceedingly verbose. The suggestion to use nondeterministic mappings for program calculation was made earlier in a pioneering paper by Meertens [25]. Similar ideas, but in a slightly different spirit, can also be found in the CIP specification language [7, 8] and in refinement of expression languages [26, 30].

The semantics that we have sketched leaves many questions unanswered. In particular, there is an equivalence of categories between the category of typed lambda calculi with surjective pairing and the category of cartesian closed categories. To obtain a similar result for pointwise relational calculi, their logical properties have to be axiomatised. We have not even started to do this. It is hoped, however, that this paper might inspire others to join us in answering such questions.

We have benefited much from the pioneering work of Naumann and Martin on defining appropriate exponents for relations and predicate transformers [24, 27]. Here we have chosen a very simple notion of exponent, so that functions get mapped to their representation as binary relations, which in turn are represented as sets of pairs. We believe that for the purpose of deriving functional programs, the semantics is adequate and preferable to the more complicated constructions of [27]. Naumann aims at a semantics for higher-order imperative programs, which is a much more difficult problem.

Acknowledgements

We would like to thank Richard Bird, Luke Ong and David Naumann for many enjoyable discussions on the topic of this paper. Lambert Meertens first kindled our interest in nondeterminacy in the late eighties, but at that time we did not have the courage to see it through.

References

- [1] C. J. Aarts, R. C. Backhouse, P. F. Hoogendijk, E. Voermans, and J. C. S. P. Van der Woude. A relational theory of datatypes. September 1992.
- [2] A. Augusteijn. An alternative derivation of a binary heap construction function. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 368–374. Springer-Verlag, 1992.
- [3] R. C. Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, 1989.
- [4] R. C. Backhouse, P. De Bruin, P. F. Hoogendijk, G. Malcolm, T. S. Voermans, and J. C. S. P. Van der Woude. Polynomial relators. In M. Nivat, C. S. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology, Workshops in Computing*, pages 303–362. Springer-Verlag, 1992.
- [5] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:613–641, 1978.
- [6] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.
- [7] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Wössner. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [8] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Texts and Monographs in Computer Science. Springer-Verlag, 1982.
- [9] R. Berghammer, P. Kempf, G. Schmidt, and T. Ströhlein. Relation algebra and logic of programs. In H. Andreka and J. D. Monk, editors, *Algebraic Logic*, volume 54 of *Colloquia Mathematica Societatis Janos Bolyai*, pages 37–58. North-Holland, 1991.
- [10] R. Berghammer and B. Von Karger. Formal derivation of CSP programs from temporal specifications. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 180–196. Springer-Verlag, 1995.
- [11] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO ASI Series F*, pages 151–216. Springer-Verlag, 1989.
- [12] R. S. Bird. *Introduction to Functional Programming in Haskell*. International Series in Computer Science. Prentice Hall, 1998.
- [13] R. S. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1996.
- [14] C. Brink and G. Schmidt, editors. *Relational Methods in Computer Science*. Springer-Verlag, 1996. Supplemental Volume of the Journal *Computing*, to appear.
- [15] W. Chen and J. T. Udding. Program inversion: More than fun! *Science of Computer Programming*, 15(1):1–13, 1990.
- [16] P. L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, 1986.
- [17] E. W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice Hall, 1976.
- [18] R. W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, 1963.

- [19] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
- [20] C. A. R. Hoare and J. He. The weakest prespecification, I. *Fundamenta Informaticae*, 9(1):51–84, 1986.
- [21] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, 1990.
- [22] E. Knapen. Relational programming, program inversion and the derivation of parsing algorithms. Computing science notes, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1993.
- [23] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.
- [24] C. E. Martin. *Preordered categories and predicate transformers*. D.Phil. thesis, Computing Laboratory, Oxford, UK, 1991.
- [25] L. Meertens. Algorithmics – towards programming as a mathematical activity. In J. W. De Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 3–42. North-Holland, 1987.
- [26] J. M. Morris. Programming by expression refinement: The KMP algorithm. Chapter 37 of W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business*. Springer-Verlag, 1990.
- [27] D. A. Naumann. Towards squiggly refinement algebra. In D. Gries and W. P. De Roever, editors, *Programming Concepts and Methods*, pages 346–365. Chapman and Hall, 1998.
- [28] B. Schoenmakers. Inorder traversal of a binary heap and its inversion in optimal time and space. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 291–301. Springer-Verlag, 1992.
- [29] D. R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.
- [30] N. T. E. Ward. *A refinement calculus for nondeterministic expressions*. Ph.D. thesis, University of Queensland, 1994.

5 Appendix: Haskell Program

This is the full text of the program that we derived in Section 2.

```
data Tree a = Null | Bin (Tree a) a (Tree a)
  deriving Show
```

```
build :: Ord a => [a] -> Tree a
build = roll . foldr join []
  where
    join a t = search (Null,t)
    search (t,x) = if bet a (t,x)
                  then (a,t):x
                  else search (shunt (t,x))
```

```
roll = foldl bin Null
  where
    bin s (a,t) = Bin s a t
```

```
shunt (s,(a,t):ts) = (Bin s a t,ts)
```

```
bet a (t,x) = above a t && fit a x
```

```
above a Null      = True
above a (Bin s b t) = a <= b
```

```
fit a []          = True
fit a ((b,t):u)   = b <= a
```


Towards a Toolkit for Actor System Specification

Carolyn L. Talcott

Stanford University, CA

clt@cs.stanford.edu

1 Introduction

This paper represents a first step towards developing a toolkit for modular reasoning about components of object-based open distributed systems. Our starting point is the actor theory framework [10,11], a general semantic framework, based on the actor computation model, for specifying and reasoning about such systems. System components are modeled as actor theory configurations and an actor theory determines both an operational semantics and an abstract trace-like interaction semantics for its configurations. The “tools” presented here take the form of relations, operations, and transformations on actor theories that can be used to establish refinement and interaction equivalence relations between actor theory configurations. The use of these tools is illustrated by simple but representative examples.

An *interaction simulation* relation is presented that provides a means of establishing interaction equivalence of configurations. The configurations may belong to different actor theories, as long as the theories are concerned with a common set of interactions. Interaction simulation can be thought of as a form of fairness-preserving bisimulation. An *inclusion relation* and *summation operation* are presented that provide a way of structuring actor theories and relating the semantics given to a configuration by a summand theory to that given by the summation. An *internalization* transformation is presented that provides a means of hiding communications internal to a group of actors, making the group appear more like a multi-threaded process than a collection of independent agents. This transformation is typically applied to a configuration obtained by combining several components, and sets the stage for further simplification, for example to relate a composite configuration to a component having a much simpler description. A *big-step* transformation is presented that allows sequences of rule applications to be treated as a single step, thus reducing the amount of interleaving to be considered and hiding detail that is not of interest. This transformation can be applied to simplify the result of an internalization transformation. It is also useful in simplifying the theory obtained as the semantics of a program or specification. Such theories typically have rules dealing with each program or specification construct, giving rise to computations with many book-keeping steps that for purposes of reasoning about a specific program clutter the picture.

Plan The remainder of this paper is organized as follows. The actor theory framework is summarized in § 2. The relations and operations of the toolkit described above are presented in § 3. Related and future work are discussed in § 4.

2 The Actor Theory Framework

A long-term goal of our work is to develop a semantic foundation for open distributed systems that supports modular specification and reasoning about components of open distributed systems at many levels of detail. We take the actor model of computation [9,1] as our starting point. Actors are independent computational agents that interact solely via asynchronous message passing. An actor can create other actors; send and receive messages; and modify its own local state. An actor can only effect the local state of other actors by sending them messages, and it can only send messages to its acquaintances: addresses of actors it was given upon creation; addresses it received in a message; or addresses of actors it created. Actor computation semantics admits only fair computations. What we observe about an actor system component is its patterns of interaction with its environment.

The actor theory framework used here evolved, with continued use, from the notion of abstract actor structure presented in [10], details can be found in [11]. An actor theory specifies a set of *actor names*, and a set of *messages* that may be communicated between actors. It also specifies a set of *actor states* and local *reaction rules* that determine the behavior of actors in a given state. *Configurations* encapsulate a collection of actors and messages allowing access only to certain actors called *receptionists*. An actor theory also specifies local properties from which the admissibility requirements on computation paths are derived. Admissibility generalizes the usual actor fairness property, allowing for flexibility in the way that internal computation is modeled, and for expressing constraints both on the communication infrastructure and on the environment with which a component interacts. The operational semantics of a configuration is derived from the reaction rules by adding rules for interaction with the environment. The interaction semantics of a component is an abstract (denotational) semantics obtained from the admissible computations by hiding the details of internal computation and observing only the interactions with the environment. More precisely, an actor theory, $A\mathit{Th}$, is a structure:

$$A\mathit{Th} = \langle \langle \mathbf{A}, \mathbf{Msg}, \mathbf{S}, acq, inActs \rangle, \langle \mathbf{L}, RR \rangle, \langle \mathbf{Af}, acq, aActs, Focus \rangle \rangle$$

The first part is the basic structural part in which $\mathbf{A}$ is the set of actor names, $\mathbf{Msg}$ is the set of messages and $\mathbf{S}$ is the set of states. $acq : \mathbf{Msg} \cup \mathbf{S} \rightarrow \mathbf{P}_\omega(\mathbf{A})$ is the acquaintance function, and $inActs : \mathbf{S} \rightarrow \mathbf{P}_\omega(\mathbf{A})$ is the internal actors function, where $\mathbf{P}_\omega(\_)$ is the finite power set operator. The acquaintances of a message are the actor names communicated in the message. The internal actors of a state are the names of actors active in that state, and must be non-empty. The acquaintances of a state are the actor names known by the internal actors,

and include the internal actors. We use the term *message packet* to refer to a pair consisting of an actor name (called the target) and a message. We write $a \triangleleft M$ for the packet with target a and message M and we let $\mathbf{MP}$ denote the set of message packets. We let a range over $\mathbf{A}$, M range over $\mathbf{Msg}$, mp range over $\mathbf{MP}$, A range over $\mathbf{P}_\omega(\mathbf{A})$, s range over $\mathbf{S}$, σ range over $\mathbf{M}_\omega(\mathbf{S})$, μ range over $\mathbf{M}_\omega(\mathbf{MP})$, where for any set X , $\mathbf{M}_\omega(X)$ is the finite multisets of X , with multiset union operator $_ \cdot _$.

The second part of *ATH* is the rules part, in which $\mathbf{L}$ is the set of labels, and RR is the set of reaction rules. Each reaction rule, $rr \in RR$, has the form

$$l : s \xrightarrow[\mu_s]{\mu_r} s' \cdot \sigma$$

where: l is the label; s the source state; μ_r is the multiset of consumed packets; $s' \cdot \sigma$ is the multiset of target states, s' being the new state of the internal actors of s (possibly with new internal actors) and σ contains the states of any newly created groups of actors; and μ_s is the multiset of sent packets. An actor theory must obey the fundamental acquaintance (locality) laws of actors [2]. In particular, only messages addressed to internal actors may be consumed, and each acquaintance of a target state or of a sent message must be either an acquaintance of the source state or an acquaintance of a consumed message, or a newly created actor.

The third part of *ATH* is the admissibility part. $\mathbf{Af}$ is the set of action focus points, $af \in \mathbf{Af}$. An action focus can be thought of as a thread of activity within a group of actors. $acq, aActs : \mathbf{Af} \rightarrow \mathbf{P}_\omega(\mathbf{A})$ give the acquaintances and active actors in an action focus point, $Focus : RR \rightarrow \mathbf{Af}$ assigns to each rule an action focus point. We require that the active actors of an action focus be non-empty and that they be included in its acquaintances. (To simplify the discussion we have omitted the admissibility features that express constraints on the environment, see [11] for treatment of these additional features.)

Finally, we require actor theories to be parametric in the set of actor names. Thus bijections on actor names (*renamings*) must extend to messages, states, and action focus points such that operations on these sets commute with any extended renaming.

The *operational semantics* of an actor theory is given by a labelled transition relation on configurations derived from the reaction rules. A *configuration* consists of an interior and an interface. An interior can be thought of as representing a global snapshot of an actor system with respect to some idealized observer [1]. It consists of a multiset of states and message packets, in which the states have disjoint sets of internal actor names. We let $\mathbf{I}$ stand for the set of configuration interiors, and let I range over $\mathbf{I}$. The interface of a configuration specifies two (disjoint) sets of actor names: the *receptionists* and the *externals*. Receptionists are those internal actors visible outside the component and the externals are names of actors outside the component that are possibly known by actors within the component. Both sets may grow as computation proceeds. I_χ^ρ is a configuration with interior I , receptionists, ρ , and external actors, χ . The receptionists of a configuration must be a subset of the internal actors, the

externals must be disjoint from the internal actors, and the acquaintances of the interior must be either internal actors or members of the interface externals. We let $\mathbf{K}$ stand for the set of configurations, and let K range over $\mathbf{K}$. The acquaintance, internal actors, and renaming functions extend naturally to interiors and configurations.

The labelled transition relation has elements of the form $K \xrightarrow{tl} K'$, where K is the *source* of the transition and K' is the *target* and where tl is the *label*. Transition labels are either internal labels, input labels, output labels, or the idle label, **idle**. An internal label has the form $l(af, \mu_r)$ where l is a rule label, af is the action focus, and μ_r is the set of consumed message packets. An input label has the form $\text{in}(a \triangleleft M)$, indicating a message is arriving from the environment. An output label has the form $\text{out}(a \triangleleft M)$ indicating a message is being transmitted to the environment.

Definition 1 (Transition rules).

$$\begin{aligned}
(\text{internal}) \quad & \langle s_0 \cdot \mu_r \cdot I_0 \rangle_\chi^\rho \xrightarrow{l(af, \mu_r)} \langle s_1 \cdot \sigma \cdot \mu_s \cdot I_0 \rangle_\chi^\rho \\
& \text{if } rr = l : s_0 \xrightarrow[\mu_s]{\mu_r} s_1 \cdot \sigma \in RR, \quad af = \text{Focus}(rr), \\
& \text{and } (\text{inActs}(s_1 \cdot \sigma) - \text{inActs}(s_0)) \cap (\text{inActs}(I_0) \cup \chi) = \emptyset \\
(\text{in}) \quad & I_\chi^\rho \xrightarrow{\text{in}(a \triangleleft M)} \langle I \cdot a \triangleleft M \rangle_{\chi \cup (\text{acq}(M) - \rho)}^\rho \\
& \text{if } a \in \rho \wedge \text{acq}(M) \cap \text{inActs}(I) \subseteq \rho \\
(\text{out}) \quad & \langle I \cdot a \triangleleft M \rangle_\chi^\rho \xrightarrow{\text{out}(a \triangleleft M)} I_{\chi \cup (\text{acq}(M) - \chi)}^{\rho \cup (\text{acq}(M) - \chi)} \quad \text{if } a \in \chi \\
(\text{idle}) \quad & I_\chi^\rho \xrightarrow{\text{idle}} I_\chi^\rho
\end{aligned}$$

The computation path semantics of a configuration is the set of (infinite) transition sequences leading from that configuration.

Definition 2. A *computation path*, π , is an infinite sequence of transitions of the form

$$\pi = [K_i \xrightarrow{tl_i} K_{i+1} \mid i \in \mathbf{Nat}]$$

We let $\mathcal{P}$ be the set of computation paths and let $\mathcal{P}(K)$ be the set of computation paths of a configuration— is the set of computation paths whose initial configuration is K .

Definition 3. *Admissible computation paths* are those in which every enabled action focus or message packet is eventually either fired or becomes permanently disabled. An action focus is enabled if it is the focus of an enabled internal transition and a message packet is enabled if its target is an external actor or if there is an enabled internal transition that consumes it. An internal transition tl is enabled at stage i in a path if there is some K such that $K_i \xrightarrow{tl} K$ where K_i is the source configuration of $\pi(i)$. $\mathcal{A}$ is the set of admissible paths, and the set of admissible paths for K is $\mathcal{A}(K) = \mathcal{P}(K) \cap \mathcal{A}$.

The *interaction semantics*, $\llbracket K \rrbracket$, of a configuration K is the set of interaction paths obtained from admissible computations by hiding the configurations and internal rule application steps, while keeping the interface of the initial configuration and the order of the interactions. If $\mathbf{AMP} \subseteq \mathbf{MP}$, then $\llbracket K \rrbracket_{\mathbf{AMP}}$ is the set of interaction paths in which only inputs from $\mathbf{AMP}$ occur. An interaction path has an interface and a sequence of interactions. The function $cp2ip$ maps computation paths to interaction paths. $cp2ip(\pi)$ has the same interface as the interface of the initial configuration of π , and the interaction sequence is represented by the set of pairs of the form (i, io) such that the label of i th transition of π is the input/output label io .

Definition 4 (Interaction Semantics $\llbracket K \rrbracket$).

$$\begin{aligned} cp2ip([I_j^{\rho_j} \xrightarrow{tl_j} I_{j+1}^{\rho_{j+1}} \mid j \in \mathbf{Nat}]) \\ = \{(j, tl_j) \mid j \in \mathbf{Nat} \text{ and } tl_j \text{ is an input/output label}\}_{\chi_0}^{\rho_0} \\ \llbracket K \rrbracket = \{cp2ip(\pi) \mid \pi \in \mathcal{A}(K)\} \\ \llbracket K \rrbracket_{\mathbf{AMP}} = \{ip \in \llbracket K \rrbracket \mid \text{all packets input in } ip \text{ are in } \mathbf{AMP}\} \end{aligned}$$

When considering several actor theories such that a configuration may belong to more than one, we make explicit the intended interpretation of a configuration by writing $K : Ath$ for the interpretation of K in the actor theory Ath .

Definition 5 (Interaction Equivalence). Two configurations are interaction equivalent (written $K_0 : Ath_0 \stackrel{i}{\simeq} K_1 : Ath_1$) if they have the same interaction semantics, that is, if $\llbracket K_0 : Ath_0 \rrbracket = \llbracket K_1 : Ath_1 \rrbracket$. They are interaction equivalent for inputs in $\mathbf{AMP}$ (written $K_0 : Ath_0 \stackrel{i}{\simeq}_{\mathbf{AMP}} K_1 : Ath_1$) if they have the same restricted interaction semantics, that is, if $\llbracket K_0 : Ath_0 \rrbracket_{\mathbf{AMP}} = \llbracket K_1 : Ath_1 \rrbracket_{\mathbf{AMP}}$.

The notion of actor theory is more general than traditional actor semantics and hence lacks some of the nice properties. An important special case is the class of *transition-label fair* actor theories in which admissibility corresponds to fairness for internal transition labels and external mail delivery.

Definition 6 (Transition label fair). An actor theory is transition-label fair if the following two conditions hold:

1. if $x \in \mathbf{Af} \cup \mathbf{MP}$ is enabled infinitely often in a computation path, π , then x is an externally addressed packet, or there is some internal transition label tl that fires x and such that tl is enabled infinitely often in π .
2. if internal transition label tl is enabled in an admissible computation path, then it either becomes permanently disabled or it fires.

To define specific actor theories we define the sets of messages and states of each actor theory by giving constructor functions. The defined sets are the sets generated by the constructors. Rules are given schematically using terms built

from message and state constructors, constants and variables. We use a rule label together with a list of terms (possibly containing variables) instantiating the variables appearing in the rule to denote a rule instance.

Example 1 (The Ticker AT.). A Ticker is a component with a single receptionist that accepts requests for the time, and replies with a number. Every such request gets exactly one reply, and the number in the reply must eventually get larger. The Ticker actor theory, ATh_{Ticker} , is defined as follows. We let a, c range over $\mathbf{A}$ and n ranges over $\mathbf{Nat}$. The structural part of ATh_{Ticker} is given by three message constructors and one state constructor.

- $\text{tick} : \text{Msg}, \quad \text{time} @ _ : \mathbf{A} \rightarrow \text{Msg}, \quad \text{reply} : \text{Nat} \rightarrow \text{Msg}$
- $T : \mathbf{A} \times \text{Nat} \rightarrow \mathbf{S}$

with the acquaintance and internal actors functions defined by

$$\begin{aligned} \text{acq}(\text{tick}) &= \text{acq}(\text{reply}(n)) = \emptyset \quad \text{and} \quad \text{acq}(\text{time} @ c) = \{c\} \\ \text{acq}(T(a, n)) &= \text{inActs}(T(a, n)) = \{a\} \end{aligned}$$

The rules part of ATh_{Ticker} has two rule schemes

$$\text{tick} : T(a, n) \xrightarrow[a \triangleleft \text{tick}]{a \triangleleft \text{tick}} T(a, n+1), \quad \text{time} : T(a, n) \xrightarrow[c \triangleleft \text{reply}(n)]{a \triangleleft \text{time} @ c} T(a, n)$$

The tick rule models the passing of time, and the time rule models the reaction to time requests. For the admissibility part we take the action focus points to be actor names, $\mathbf{Af} = \mathbf{A}$, with active actors and acquaintances defined by $a\text{Acts}(a) = \text{acq}(a) = \{a\}$. The focus function is defined by

$$\text{Focus}(\text{tick}(a, n)) = \text{Focus}(\text{time}(a, n)) = a$$

The configuration $\langle T(t, 0) \cdot t \triangleleft \text{tick} \rangle^t$ thus describes a Ticker component, that is its interaction semantics meets the requirements (stated informally) at the beginning of this example. We note that ATh_{Ticker} is transition-label fair.

3 Operations and Relations on Actor Theories

We are interested in relations on and operations for transforming and combining actor theories that allow us to move around in the space of actor theories in semantically meaningful ways to support modular reasoning about actor system components. In this section we describe some of these relations and operations and demonstrate their use. We assume a given set of actor names, $\mathbf{A}$, that is the same for all actor theories under consideration.

3.1 Interaction Simulation

One way to show that two configurations, $K_0 : Ath_0$ and $K_1 : Ath_1$, are interaction equivalent for inputs **AMP** is to find an interaction preserving correspondence between their admissible computations with inputs in **AMP**. An *interaction simulation* is a relation on configurations and enabled transitions that provides a local means of defining such a correspondence. The main difficulty is to make sure that the induced correspondence of computations preserves admissibility. To simplify the presentation, we restrict attention to transition-label fair actor theories (Definition 6).

Definition 7 (Interaction simulation). Let Ath_0, Ath_1 be transition-label fair actor theories. and let **AMP** be a subset of the packets common to both theories. A binary relation $\sim_C$ on $(K_0 : Ath_0) \times (K_1 : Ath_1)$ is an interaction simulation for **AMP** if for each related pair $K_0 : Ath_0 \sim_C K_1 : Ath_1$ the following conditions hold.

- (1) K_0 and K_1 have the same interface and the same multiset of packets with external target.
- (2) there is a bijection ϕ_C between enabled internal transitions (definition 3) of K_0 and K_1 such that (extending as the identity on input/output and idle transitions) if $K_0 \xrightarrow{tl} K'_0$, with inputs in **AMP**, then there is K'_1 such that $K'_0 \sim_C K'_1$ and $K_1 \xrightarrow{\phi_C(tl)} K'_1$, and dually with 0-1 interchanged.

Lemma 1. Let Ath_0, Ath_1 be transition-label fair actor theories with **AMP** a subset of the packets common to both theories, and let $\sim_C$ be an interaction simulation on Ath_0, Ath_1 for **AMP**. Then

$$K_0 \sim_C K_1 \Rightarrow K_0 : Ath_0 \stackrel{i}{\simeq}_{\mathbf{AMP}} K_1 : Ath_1$$

3.2 Inclusion and Summation

Definition 8 (Inclusion). An inclusion mapping $\varphi : Ath_0 \mapsto Ath_1$ is a mapping of the messages, states, and action focus points of Ath_0 to the corresponding sets of Ath_1 , that is the identity on messages and injective on action focus points, and that, preserves acquaintance, internal actors, active actors, and extended renamings. Furthermore, extending the mapping homomorphically to packet, multiset, and configuration formation, rules of the included theory map to rules of the including theory, preserving action focus, and there are no additional rules for states of the included theory, although there may of course be new rules for new states.

Lemma 2. If $\varphi : Ath_0 \mapsto Ath_1$ is an actor theory inclusion, $\mathbf{MP}_0$ is the set of packets of Ath_0 , and K is a configuration of Ath_0 , then

$$K : Ath_0 \stackrel{i}{\simeq}_{\mathbf{MP}_0} \varphi(K) : Ath_1$$

Definition 9 (Sum). Let Ath_0 , Ath_1 , and Ath_2 be actor theories such that the identity mappings $\varphi_j : Ath_0 \mapsto Ath_j$ for $j \in \{1, 2\}$ are actor theory inclusions, and such that the intersections of the sets of messages, states, and action focus points of Ath_1 and Ath_2 are the sets of messages, states, and action focus points of Ath_0 . Then the sum Ath_{12} of Ath_1 and Ath_2 with shared part Ath_0 (written $Ath_{12} = Ath_1 +_{Ath_0} Ath_2$) is formed by taking component-wise union of the structures.

Lemma 3. The sum operation (when defined) is associative and commutative. Furthermore, if $\varphi_j : Ath_0 \mapsto Ath_j$ for $j \in \{1, 2\}$ are actor theory inclusions with Ath_0 being the intersection of Ath_1 and Ath_2 as above, and $Ath_{12} = Ath_1 +_{Ath_0} Ath_2$, then the identity mappings $\varphi_j^{12} : Ath_j \mapsto Ath_{12}$, are actor theory inclusions for $j \in \{1, 2\}$. Thus, if K_j is a configuration of Ath_j , then

$$K_j : Ath_j \xrightarrow{i}_{\mathbf{MP}_j} K_j : Ath_{12} \quad \text{for } j \in \{1, 2\}.$$

We now introduce the idea of a dispatcher component, give a simple actor theory that describes the behavior of a dispatcher, and then form a composite description of a dispatcher, using the sum operation. This composite description will be shown to be interaction equivalent to the simple one, using the operations and relations of the toolkit.

Definition 10 (Dispatcher Interaction Specification). A dispatcher component has a single receptionist, say d , and two external actors, say w_0, w_1 . The dispatcher accepts input packets of the form $d \triangleleft \mathbf{task}(v) @ c$ where c is an actor name other than d (the customer) and v describes the task. The interaction paths of a dispatcher have the property that every input $d \triangleleft \mathbf{task}(v) @ c$ has a unique corresponding output $w \triangleleft \mathbf{task}(v) @ c$ where $w \in \{w_0, w_1\}$, with w chosen according to some unspecified internal decision process, and these are the only outputs.

In the following we assume given a set $\mathbf{Val}$ equipped with an acquaintance function $acq : \mathbf{Val} \rightarrow \mathbf{P}_\omega(\mathbf{A})$ for task parameters. We let v range over $\mathbf{Val}$, d, t, w_0, w_1, c range over $\mathbf{A}$, and n range over $\mathbf{Nat}$.

Example 2 (Simple Dispatcher AT.). The actor theory Ath_D provides a simple single actor component with the interaction semantics of a dispatcher. Ath_D has one message constructor and one state constructor:

$$- \mathbf{task}(\_) @ \_ : \mathbf{Val} \times \mathbf{A} \rightarrow \mathbf{Msg}, \quad D : \mathbf{A}^3 \rightarrow \mathbf{S}$$

such that

$$\begin{aligned} acq(\mathbf{task}(v) @ c) &= acq(v) \cup \{c\}, & acq(D(d, w_0, w_1)) &= \{d, w_0, w_1\} \\ inActs(D(d, w_0, w_1)) &= \{d\}. \end{aligned}$$

Ath_D has two rules:

$$\begin{aligned} \mathbf{disp} : D(d, w_0, w_1) &\xrightarrow[w_0 \triangleleft \mathbf{task}(v) @ c]{d \triangleleft \mathbf{task}(v) @ c} D(d, w_0, w_1), \\ \mathbf{togl} : D(d, w_0, w_1) &\longrightarrow D(d, w_1, w_0). \end{aligned}$$

ATh_D uses actor names tagged with rule indicators as action focus points:

$Focus(\mathbf{disp}(d, w_0, w_1, v, c)) = d_d$ and $Focus(\mathbf{togl}(d, w_0, w_1)) = d_t$. The simple dispatcher component, $Disp_D(d, w_0, w_1)$, is defined by

$$Disp_D(d, w_0, w_1) = \langle D(d, w_0, w_1) \rangle_{w_0, w_1}^d.$$

Defining $\mathbf{AMP}_D = \{a \triangleleft \mathbf{task}(v) @ c \mid c \in \mathbf{A} - \{a\}, v \in \mathbf{Val}\}$ it is easy to verify that each interaction path in $\llbracket Disp_D(d, w_0, w_1) \rrbracket_{\mathbf{AMP}_D}$ satisfies the dispatcher specification (Definition 10).

A dispatcher can also be constructed as a composition of an actor that serves as a switch and another actor that performs the internal decision process. A switch actor knows three external actors: two workers and a decision helper. It accepts a request of the form $\mathbf{task}(v) @ c$, and then sends a request for decision information (a number) to the decision helper. When a reply to the decision information request is received, the switch chooses a worker based on the reply modulo 2. We assume that the decision helper is a ticker, thus the switches information request is the ticker $\mathbf{time}$ request message.

Example 3 (Switch AT.). Our switch actor is described by the actor theory, ATh_{Switch} . ATh_{Switch} has three message and two state constructors:

- $\mathbf{task}(\_) @ \_ : \mathbf{Val} \times \mathbf{A} \rightarrow \mathbf{Msg}$, $\mathbf{time} @ \_ : \mathbf{A} \rightarrow \mathbf{Msg}$,
- $\mathbf{reply}(\_) : \mathbf{Nat} \rightarrow \mathbf{Msg}$,
- $\mathbf{S} : \mathbf{A}^4 \rightarrow \mathbf{S}$, $\mathbf{Sw} : \mathbf{A}^4 \times \mathbf{Val} \times \mathbf{A} \rightarrow \mathbf{S}$

such that letting $A = \{d, t, w_0, w_1\}$

$$\begin{aligned} acq(\mathbf{task}(v) @ c) &= acq(v) \cup \{c\} \wedge acq(\mathbf{time} @ c) = \{c\} \wedge acq(\mathbf{reply}(n)) = \emptyset \\ acq(\mathbf{S}(d, t, w_0, w_1)) &= A \wedge acq(\mathbf{Sw}(d, t, w_0, w_1, v, c)) = A \cup \{c\} \\ inActs(\mathbf{S}(d, t, w_0, w_1)) &= inActs(\mathbf{Sw}(d, t, w_0, w_1, v, c)) = \{d\}. \end{aligned}$$

ATh_{Switch} has two rules:

$$\mathbf{task} : \quad \mathbf{S}(d, t, w_0, w_1) \xrightarrow[t \triangleleft \mathbf{time} @ d]{d \triangleleft \mathbf{task}(v) @ c} \mathbf{Sw}(d, t, w_0, w_1, v, c)$$

$$\mathbf{disp} : \quad \mathbf{Sw}(d, t, w_0, w_1, v, c) \xrightarrow[w_j \triangleleft \mathbf{task}(v) @ c]{d \triangleleft \mathbf{reply}(n)} \mathbf{S}(d, t, w_0, w_1) \quad \text{where } j = n \bmod 2.$$

The admissibility part of ATh_{Switch} uses actor names as action focus points with

$$Focus(\mathbf{task}(d, t, w_0, w_1, v, c)) = d \quad \text{and} \quad Focus(\mathbf{disp}(d, t, w_0, w_1, v, c, n)) = d$$

Example 4 (Switch + Ticker AT). To be able to combine the switch and ticker actors we need an actor theory that contains both the switch and ticker descriptions. Thus we form the sum

$$ATh_{\text{SwT}} = ATh_{\text{Ticker}} + ATh_{\text{SwT-shared}} ATh_{\text{Switch}}$$

where $ATH_{\text{SwT-shared}}$ is the shared part consisting of the **time** and **reply** messages and the set of action focus points. The states and rules are disjoint. Note that by Lemma 3 the identity inclusion maps on messages, states, and action focus points give actor theory inclusions of the Ticker and the Switch into the combined theory. Thus the semantics of switch and ticker actors is preserved in the combined theory.

Now we define a dispatcher component using a single switch and ticker.

$$\text{Disp}_{ST}(d, w_0, w_1) = \langle S(d, t, w_0, w_1) \cdot T(t, 0) \cdot t \triangleleft \mathbf{tick} \rangle_{w_0, w_1}^d$$

Note that the Ticker is not externally visible.

Theorem 1. $\text{Disp}_{ST}(d, w_0, w_1)$ satisfies the interaction requirements given in Definition 10. In fact, the ST Dispatcher is interaction equivalent to the simple dispatcher for inputs $\mathbf{AMP}_D$.

$$\text{Disp}_{ST}(d, w_0, w_1) : ATH_{\text{SwTD}} \stackrel{i}{\simeq} \mathbf{AMP}_D \text{Disp}_D(d, w_0, w_1) : ATH_D$$

The interaction equivalence is established by transforming the configuration $\text{Disp}_{ST}(d, w_0, w_1) : ATH_{\text{SwTD}}$ in a semantics preserving manner to a configuration that is interaction similar, using the internalization and big-step transformations described below.

3.3 Internalization

To reason about an actor system component with multiple actor states, it is often useful to simplify the description by treating a multiset of states as one state which also has an internal mail pool. Packets to internal actors can be delivered to the internal pool at any time. Rules of the original theory are modified to take packets from the internal pool and to put packets sent to internal actors directly into the pool thus hiding internal communications and allowing for further simplifications. This actor theory transformation is called *internalization*.

Definition 11 (Internalization). The internalization $\text{Intern}(ATH)$ of actor theory ATH defined as follows. $\text{Intern}(ATH)$ has the same the messages as ATH . A state of $\text{Intern}(ATH)$ is an interior of ATH such that the target of any packet is an interior actor. The acquaintance and internal actor functions are given naturally by considering each internalized state to be an interior of ATH .

$\text{Intern}(ATH)$ has a delivery rule as well as an internalization, $\bar{l}$, of each rule, l , of ATH .

$$\mathbf{dlv} : \langle \sigma, \mu \rangle \xrightarrow{mp} \langle \sigma, \mu \cdot mp \rangle \text{ if } \text{target}(mp) \in \text{inActs}(\sigma)$$

$$\bar{l} : \langle \sigma \cdot s, \mu \cdot \mu_r \rangle \xrightarrow[\mu_s^O]{\mu_r} \langle \sigma \cdot s' \cdot \sigma', \mu \cdot \mu_s^I \rangle \text{ if } l : s \xrightarrow[\mu_s^I \cdot \mu_s^O]{\mu_r} s' \cdot \sigma' \in RR,$$

$$\text{target}(\mu_s^I) \subseteq \text{inActs}(\sigma \cdot s' \cdot \sigma'), \quad \text{and} \quad \text{target}(\mu_s^O) \cap \text{inActs}(\sigma \cdot s' \cdot \sigma') = \emptyset$$

For the admissibility part of $Intern(ATH)$ the action focus points are finite sets of actor names (for the delivery rule) and internal transition labels (for the internalized rules).

$$\begin{aligned} \mathbf{Af} : Intern(ATH) &= \mathbf{P}_\omega(\mathbf{A}) \cup \mathbf{tLabel}_I \\ \mathbf{tLabel}_I &= \mathbf{tLabel} - (\mathbf{in}(\mathbf{MP}) \cup \mathbf{out}(\mathbf{MP}) \cup \{\mathbf{idLe}\}) \end{aligned}$$

where the active actors and focus functions are defined by

$$\begin{aligned} aActs(A) &= A, & aActs(af, \mu) &= aActs(af) \\ Focus(\mathbf{dlv}(\sigma, \mu, mp)) &= inActs(\sigma) \\ Focus(\overline{rr}) &= l(Focus(rr), \mu_r) \text{ if } \overline{rr} = \bar{l} : \langle \sigma \cdot s, \mu \cdot \mu_r \rangle \xrightarrow[\mu_s^O]{} \langle \sigma \cdot s' \cdot \sigma', \mu \cdot \mu_s^I \rangle \end{aligned}$$

Lemma 4. Internalization preserves interaction semantics for transition-label fair actor theories. That is, if ATH is transition-label fair and $K = \langle \sigma \cdot \mu_I \cdot \mu \rangle_\chi^\rho$ with $target(\mu_I) \subseteq inActs(\sigma)$, and $\overline{K} = \langle \langle \sigma, \mu_I \rangle \cdot \mu \rangle_\chi^\rho$, then

$$K : ATH \stackrel{i}{\simeq} \overline{K} : Intern(ATH)$$

Example 5 (Switch+Ticker Internalization). As an example of internalization we carry out the first step of Switch+Ticker transformation:

$$ATH_{\text{SwTi}} = Intern(ATH_{\text{SwT}})$$

We are only interested in composite states with one switch and one ticker. Let **Task** be the set defined by $\mathbf{Task} = (\mathbf{task}(\mathbf{Val}) @ \mathbf{A}) \cup \{\mathbf{nil}\}$, and define

$$\begin{aligned} \mathbf{ST} : \mathbf{A}^4 \times \mathbf{Nat} \times \mathbf{Task} \times \mathbf{MP}_{st} &\rightarrow \mathbf{S} \\ \mathbf{ST}(d, t, w_0, w_1, n, \mathbf{nil}, \mu_{st}) &= \langle \mathbf{S}(d, t, w_0, w_1) \cdot \mathbf{T}(t, n) \cdot \mu_{st} \rangle \\ \mathbf{ST}(d, t, w_0, w_1, n, \mathbf{task}(v) @ c, \mu_{st}) &= \langle \mathbf{Sw}(d, t, w_0, w_1, v, c) \cdot \mathbf{T}(t, n) \cdot \mu_{st} \rangle. \end{aligned}$$

The rules generated for the ST states are

$$\begin{aligned} \mathbf{dlv} : \quad & \mathbf{ST}(d, t, w_0, w_1, n, x, \mu_{st}) \\ & \xrightarrow{mp} \mathbf{ST}(d, t, w_0, w_1, n, x, \mu_{st} \cdot mp) \text{ if } target(mp) \in \{d, t\} \\ \overline{\mathbf{task}} : \quad & \mathbf{ST}(d, t, w_0, w_1, n, \mathbf{nil}, d \triangleleft \mathbf{task}(v) @ c \cdot mp_{st}) \\ & \longrightarrow \mathbf{ST}(d, t, w_0, w_1, n, \mathbf{task}(v) @ c, mp_{st} \cdot t \triangleleft \mathbf{time} @ d) \\ \overline{\mathbf{tick}} : \quad & \mathbf{ST}(d, t, w_0, w_1, n, x, t \triangleleft \mathbf{tick} \cdot \mu_{st}) \\ & \longrightarrow \mathbf{ST}(d, t, w_0, w_1, n + 1, x, t \triangleleft \mathbf{tick} \cdot \mu_{st}) \end{aligned}$$

$$\begin{aligned}
\overline{\text{time}} : \quad & ST(d, t, w_0, w_1, n, x, mp_{st} \cdot t \triangleleft \text{time} @ d) \\
& \longrightarrow ST(d, t, w_0, w_1, n, x, mp_{st} \cdot d \triangleleft \text{reply}(n)) \\
\overline{\text{disp}} : \quad & ST(d, t, w_0, w_1, n, \text{task}(v) @ c, mp_{st} \cdot d \triangleleft \text{reply}(n)) \\
& \xrightarrow{w_j \triangleleft \text{task}(v) @ c} ST(d, t, w_0, w_1, n, \text{nil}, mp_{st}) \quad \text{where } j = n \bmod 2.
\end{aligned}$$

The action focus and active actors of, for example, the time rule are

$$\begin{aligned}
\text{Focus}(\overline{\text{time}}(d, t, w_0, w_1, n, x, mp_{st})) &= \text{time}(t, t \triangleleft \text{time} @ d) \\
\text{aActs}(\text{time}(t, t \triangleleft \text{time} @ d)) &= \{t\}
\end{aligned}$$

By lemma 4 we have

$$\begin{aligned}
& \langle ST(d, t, w_0, w_1, n, \text{nil}, \mu_{st}) \cdot \mu \rangle_{w_0, w_1}^d : \text{A}Th_{\text{SwTi}} \\
& \quad \simeq^i \\
& \langle S(d, t, w_0, w_1) \cdot T(t, n) \cdot \mu_{st} \cdot \mu \rangle_{w_0, w_1}^d : \text{A}Th_{\text{SwTD}}
\end{aligned}$$

3.4 Big-Step Transformations

A canonical form picks one or more computation paths from the inverse image (under *cp2ip*) of each interaction path for a configuration. Restricting attention to paths in a canonical form is a useful way to reduce the number of computation paths that must be considered in proving interaction equivalence. One way that canonical forms arise is by permuting internal transitions so that certain subsequences are adjacent, thus reducing interleaving. Big-step transformations are a means of defining a set of rule sequences that generates exactly the computations of a canonical form.

Definition 12 (Big Step). Let *A*Th be an actor theory with messages **Msg**, states **S**, etc. A big-step (in *A*Th) is a sequence of rules

$$[l_i : s_i \xrightarrow[(\mu_s)_i]{(\mu_r)_i} s_{i+1} \mid i \leq k]$$

such that

- for $i \leq k$ there are σ_{i+1} such that $l_i : s_i \xrightarrow[(\mu_s)_i]{(\mu_r)_i} s_{i+1} \cdot \sigma_{i+1}$ is a rule of *A*Th
- there is at most one receive step: $(\mu_r)_i$ is non-empty for at most one i
- there are no sends or new groups created before a receive: if $(\mu_r)_i$ is non-empty, then $(\mu_s)_j$ and σ_{j+1} are empty for $j < i$

The *big-step rule* associated to a big-step as above is

$$[l_0; \dots; l_k] : s_0 \xrightarrow[\mu_s]{\mu_r} s_{k+1} \cdot \sigma$$

where $\mu_r = (\mu_r)_0 \cdot \dots \cdot (\mu_r)_k$, $\mu_s = (\mu_s)_0 \cdot \dots \cdot (\mu_s)_k$, and $\sigma = \sigma_1 \cdot \dots \cdot \sigma_{k+1}$, are the cumulative messages received and sent, and the states newly generated. The focus of a big-step rule is the focus of its first step. Note that any contiguous subsequence of a big-step is also a big-step.

Appropriate big-steps can be generated by marking some subset of the states of an actor theory as “stopping points” and considering rule sequences that lead from stopping point to stopping point.

Definition 13. Big Step State Marking Transform

A subset $\mathbf{S}_0 \subset \mathbf{S}$ is a big-step marking for ATH if the following conditions hold

- States newly generated by any rule are in $\mathbf{S}_0$: if $l : s \xrightarrow[\mu_s]{\mu_r} s' \cdot \sigma$ is a rule of ATH , then σ is a multiset from $\mathbf{S}_0$.
- If $\pi \in \mathcal{A}(\langle \sigma \cdot s \cdot \mu \rangle_\chi^\rho)$ with $\sigma \cdot s \in \mathbf{M}_\omega(\mathbf{S}_0)$, then the s -thread of π consists of big-steps between $\mathbf{S}_0$ states.
- Furthermore, big-steps between $\mathbf{S}_0$ states have no hidden enabledness.

where s -thread of a path, π , whose initial configuration contains s , is the subsequence i_j with $j \in \mathbf{Nat}$ of transitions of π with action focus in a state that has evolved from s .

If $\mathbf{S}_0$ is a big-step marking for ATH , then the *big-step transform* of ATH relative to $\mathbf{S}_0$, $BigStep(ATH, \mathbf{S}_0)$, is the actor theory obtained as follows. The messages and action focus points are those of ATH and the set of states is $\mathbf{S}_0$. The acquaintance, internal actor and active actor functions are inherited from ATH . The rules of $BigStep(ATH, \mathbf{S}_0)$ are the big-step rules for big-steps between $\mathbf{S}_0$ states. The following lemma shows that the state-marking big-step transformation preserves interaction semantics.

Lemma 5. If $\mathbf{S}_0$ is a big-step marking for ATH , K_0 is a configuration with states in $\mathbf{S}_0$, then

$$K_0 : ATH \stackrel{i}{\simeq} K_0 : BigStep(ATH, \mathbf{S}_0)$$

Example 6 (Dispatcher continued). The final transformation in the dispatcher equivalence proof is a big-step transformation. We first note that any computation starting from the initial dispatcher configuration can, by permuting independent steps, be put in a canonical form in which the internal rule applications form the following pattern (ignoring input/output steps)

$$[(\overline{\text{tick}})^*, \overline{\text{task}}, \overline{\text{time}}, \overline{\text{disp}}]^{\{0 \dots \infty\}}, (\overline{\text{tick}})^\infty.$$

Taking $\mathbf{S}_0$ to be states of the form $ST(d, t, w_0, w_1, n, \text{nil}, mp_{st})$ we get a big-step marking. Defining $ATH_{\text{SwTD}} = BigStep(ATH_{\text{SwTi}}, \mathbf{S}_0)$ we have by lemma 5

$$\text{Disp}_{ST}(d, t, w_0, w_1) : ATH_{\text{SwTi}} \stackrel{i}{\simeq} \text{Disp}_{ST}(d, t, w_0, w_1) : ATH_{\text{SwTD}}$$

Note that in ATh_{SwTD} we have the big-step rules

$$\begin{aligned}
 \mathbf{ttd}: \quad & ST(d, t, w_0, w_1, n, \mathbf{nil}, mp_{st}) \xrightarrow[w_j \triangleleft \mathbf{task}(v) @ c]{d \triangleleft \mathbf{task}(v) @ c} ST(d, t, w_0, w_1, n, \mathbf{nil}, mp_{st}) \\
 & \text{where } j = n \bmod 2 \text{ and } \mathbf{ttd} = [\mathbf{dlv}; \mathbf{task}; \mathbf{time}; \mathbf{disp}] \\
 \mathbf{tick}: \quad & ST(d, t, w_0, w_1, n, \mathbf{nil}, t \triangleleft \mathbf{tick} \cdot mp_{st}) \\
 & \longrightarrow ST(d, t, w_0, w_1, n + 1, \mathbf{nil}, t \triangleleft \mathbf{tick} \cdot mp_{st})
 \end{aligned}$$

with the focus of the rule labelled $\mathbf{ttd}$ being d and the focus of rule labelled $\mathbf{tick}$ being $\mathbf{tick}(t, t \triangleleft \mathbf{tick})$. There is also a rule for processing extra $\mathbf{time}$ requests (sent from outside) but in configurations where t is not a receptionist this will not happen.

The final step to establish Dispatcher property (theorem 1) is to show that the dispatcher described by ATh_{SwTD} and that described by ATh_D are interaction equivalent for inputs in $\mathbf{AMP}_D$. The relation between configurations of ATh_{SwTD} and ATh_D defined by

$$\begin{aligned}
 \langle ST(d, t, w_0, w_1, 2n, \mathbf{nil}, t \triangleleft \mathbf{tick}) \cdot \mu \rangle_{w_0, w_1}^d & \sim_C \langle D(d, w_0, w_1) \cdot \mu \rangle_{w_0, w_1}^d \\
 \langle ST(d, t, w_0, w_1, 2n + 1, \mathbf{nil}, t \triangleleft \mathbf{tick}) \cdot \mu \rangle_{w_0, w_1}^d & \sim_C \langle D(d, w_1, w_0) \cdot \mu \rangle_{w_0, w_1}^d
 \end{aligned}$$

for μ a multiset of task messages (to d or one of w_0, w_1) is an interaction simulation using the following correspondence of enabled transition labels:

$$\begin{aligned}
 \phi_C(\mathbf{ttd}(d, d \triangleleft \mathbf{task}(v) @ c)) &= \mathbf{disp}(d, d \triangleleft \mathbf{task}(v) @ c) \\
 \phi_C(\mathbf{tick}(t_{\mathbf{tk}}, \emptyset)) &= \mathbf{togl}(d_t, \emptyset)
 \end{aligned}$$

Thus we have established the desired equivalence.

$$\text{Disp}_{ST}(d, w_0, w_1) : ATh_{\text{SwTD}} \stackrel{i}{\simeq} \mathbf{AMP}_D \text{Disp}_D(d, w_0, w_1) : ATh_D.$$

4 Conclusions and Future Work

In this paper we described a set of relations and operations on actor theories and showed how they can serve as tools to help establish relations between descriptions of actor system components written in different notations or at different levels of abstraction. This is a first step in developing an actor theory toolkit. The long term goal for this toolkit is to support a variety of formal manipulations of actor system descriptions written in a variety of notations using actor theories as the common semantic ground. A mapping from actor theories to rewriting logic is described in [11] and this mapping is being implemented in the rewriting logic language Maude [3]. This will be a basis for defining and executing actor theory transformations.

The inclusion mappings on actor theories become theory mappings in the Maude representation and the summation operation is a very simple example of

a module operation. The work on module algebras [6] and structured theories [7] provides much more advanced structuring operations. The internalization and big-step transformations are similar to program transformation techniques such as fusion and partial-evaluation. As implementation progress it will be useful to explore this connection further and to exploit the techniques used in these communities.

Other approaches to reasoning about distributed object-based systems include process algebras for objects and distributed temporal logics. A process algebra for actors is presented in [8] with axioms for equivalence based on a bisimulation relation, which does not account for fairness requirements. DTL [4] is a temporal logic for reasoning about events in a distributed object system. In [5] proof-theory for a branching time temporal logic is developed and used to axiomatize actor computation. A general rely-guarantee style principle for compositional reasoning is derived. The temporal logic approach is complementary to our equational approach as they focus on reasoning about properties of the computations/interactions of a component, rather than relations between components.

Acknowledgements. The author would like to thank Ian Mason and Scott Smith and the anonymous referees for valuable suggestions for improving previous versions of this paper. This research was partially supported by NSF CCR-9633419, DARPA/Rome Labs AF F30602-96-1-0300, DARPA/NASA NAS2-98073. ONR N00012-99-C-0198.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.
2. H. G. Baker and C. Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, Aug. 1977.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic, 1998. URL: <http://maude.csl.sri.com>.
4. G. Denker and H.-D. Ehrich. Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97), Volume 2, IFIP TC6 WG6.1 Intern. Workshop, 21-23 July, Canterbury, Kent, UK*, pages 89–104. Chapman & Hall, 1997.
5. C. H. C. Duarte. *Proof-theoretic Foundations for the Design of Extensible Software Systems*. PhD thesis, Imperial College, University of London, 1999.
6. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Malaga, 1999.
7. F. Duran and J. Meseguer. Structured theories and institutions. In *Category Theory in Computer Science*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998.
8. M. Gaspari and G. Zavattaro. An algebra of actors. In P. Ciancariani, A. Fantechi, and R. Gorrieri, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 3–18. Kluwer, 1999.
9. C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

10. C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J.-B. Stefani, editors, *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996. Proceedings published in 1997 by Chapman & Hall.
11. C. L. Talcott. Actor theories in rewriting logic, 1999. submitted for publication.

Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic

Christiano de O. Braga^{1,2}, E. Hermann Haeusler², José Meseguer¹, and Peter D. Mosses³

¹ Computer Science Laboratory, SRI International

² Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro

³ BRICS & Department of Computer Science, University of Aarhus

Abstract. Action semantics (AS) is a framework for specifying the semantics of programming languages, in a very modular and readable way. Recently, the operational semantics of action notation (action semantics's specification language) has been rewritten using Modular SOS (MSOS), a new modular approach for specifying operational semantics. The new modular specification of action notation facilitates the creation of extensions to action semantics, to deal with new concepts, such as components. The Maude Action Tool uses the reflective capabilities of rewriting logic, implemented on the Maude system, to create an executable environment for action semantics and its potential extensions. This is achieved by a mapping between the MSOS and rewriting logic formalisms which, when applied to the MSOS semantics of each facet of action notation, yields a corresponding rewrite theory. Such rewrite theories are executed on *action programs*, that is, on the action notation translation of a given program P in a language L , according to L 's action semantics.

1 Introduction

Action semantics (AS) [11,12] is a framework for modular specification of the semantics of programming languages. Action notation (AN) is the specification language of action semantics. Action notation specifications are organized in so-called *facets*, allowing new features to be added to a language without the burden of having to reformulate what has already been specified. AN has another interesting characteristic: it allows the creation of very readable specifications, close to English text, yet remaining completely formal.

AN has a detailed operational semantics. Its original specification [11, Appendix C] was, though precise, monolithic. Recently, Mosses has rewritten it using a new technique called Modular SOS (MSOS) [13]. The MSOS of AN is not only more modular and clearer, without mapping the constructs to a kernel, but it also facilitates the extension of AN to support new semantic structures.

The Maude system [3] is an efficient implementation of rewriting logic, a logic that has been proposed as a logical and semantic framework [10] into which different specification languages can be embedded. Thanks to its reflective capabilities, the Maude system allows the creation of executable environments for the specification languages thus embedded.

This paper reports on a new tool, the Maude Action Tool (MAT): an executable environment for action semantics via the mapping of the MSOS of AN into rewriting logic. MAT allows the execution of programs based on their action semantics. Furthermore, the mapping to rewriting logic opens up the possibility of reasoning formally about the properties of AN definitions, using the formal tools [7, Chapter 5], [4] and model checking [6] capabilities of Maude.

MAT is an action semantics-driven interpreter: given an action semantics description of a programming language L , L_{ASD} , and a program written in L , $PROG_L$, the latter is executed according to L_{ASD} . This is done by first applying a *desugaring* function to L_{ASD} , producing an equational theory, L_{RL} (a Maude functional module) specifying L 's abstract syntax, semantic entities and semantic equations. The *desugaring* function maps action semantics descriptions, which can be viewed as elements of the datatype AS , to equational theories, which in Maude are the elements of the datatype **FModule** of functional modules.

$$desugaring : AS \rightarrow \mathbf{FModule}$$

The action semantics $PROG_{AN}$ of $PROG_L$ is then obtained using the equations in the theory $L_{RL} = desugaring(L_{ASD})$ to replace all L constructs by their associated action denotations. Finally, $PROG_{AN}$ is rewritten according to action notation's rewrite theory, AN_{RL} .

The *semantics* function sends MSOS specifications, which can be viewed as elements of a datatype $MSOS$, to rewrite theories, which in Maude are represented as elements of the datatype **Module** of system modules. Applying the *semantics* function to the MSOS specification of AN, AN_{MSOS} , generates action notation's rewrite theory AN_{RL} .

$$semantics : MSOS \rightarrow \mathbf{Module}$$

The fact that equational and rewrite theories are available in Maude as *data* elements of respective sorts **FModule** and **Module** is part of Maude's reflective capabilities [3,2]. Therefore the *desugaring* and *semantics* functions are meta-level functions. They use the metalevel interface of Maude, which allows the parsing, and pretty-printing of modules and terms, as well as the application of equations and rewrite rules according to user-defined strategies. For a more detailed explanation of Maude's metalevel interface, we refer to [3, Section 2.5].

Plan of the Paper As explained above, the action semantics description of a programming language L , L_{ASD} , together with the Modular SOS of action notation, AN_{MSOS} , constitute the input to MAT. Section 2 presents (very briefly) the MSOS framework, its motivation, and the MSOS of AN, and Section 3 describes in more details the *semantics* mapping, explaining how the MSOS of AN is mapped to rewriting logic using Maude's object-oriented specifications. Section 4 concludes the paper with a discussion of related work, MAT's current state and ongoing work, and a sketch of an action notation extension to support software components.

2 Modularity in SOS and the Modular SOS of Action Notation

Structural Operational Semantic (SOS), sometimes called small-step operational semantics, was first proposed by Plotkin [20], where transitions specify how an abstract machine moves from one state to another. (Big-step or “natural” semantics may formally be regarded as a special case of small-step semantics.)

Partially stimulated by Wansbrough and Hamer’s [22] work on modularity of the semantics of action notation using a monadic approach, Mosses [13] has developed a variant of the SOS framework called Modular SOS (MSOS). The main idea of MSOS is to encapsulate the semantic structures in the labels of the rules. If a new semantic structure is added, the existing rules do not change: the label variables simply range over a larger set.

Section 2.1 illustrates the modularity problem in SOS. Section 2.2 explains how MSOS copes with it and briefly explains how the operational semantics of action notation can be defined using the MSOS approach.

2.1 The SOS Modularity Problem

To illustrate the SOS modularity problem, let us consider the following SOS rules for the evaluation of arithmetical addition expressions.

$$\begin{array}{c}
 e ::= v \mid e_1 + e_2 \\
 v ::= n \quad , \quad n \in \mathbb{N}
 \end{array}$$

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n} \quad \frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2} \quad \frac{e_2 \rightarrow e'_2}{v_1 + e_2 \rightarrow v_1 + e'_2} \quad (1)$$

If one now considers a functional extension¹, the existing rules would have to be modified in order to support the notion of environments, besides the addition of the rule for function application. In the original formulation of the rules, the transition relation only involved expressions as configurations. Now it involves both expressions and an environment, so all the rules have to be reformulated:

$$\begin{array}{c}
 e ::= \dots \mid x \\
 v ::= \dots \mid \lambda x(e) \\
 x \in Var, \quad \rho \in Environment = Var \rightarrow Val
 \end{array}$$

$$\frac{n = n_1 + n_2}{\rho \vdash n_1 + n_2 \rightarrow n} \quad (2)$$

$$\frac{\rho \vdash e_1 \rightarrow e'_1}{\rho \vdash e_1 + e_2 \rightarrow e'_1 + e_2} \quad \frac{\rho \vdash e_2 \rightarrow e'_2}{\rho \vdash v_1 + e_2 \rightarrow v_1 + e'_2} \quad (3)$$

¹ Here we illustrate extension with dynamic bindings, since to specify static bindings in small-step SOS, either auxiliary configurations or a substitution operation would be needed.

$$\frac{\rho \vdash e_1 \rightarrow e'_1}{\rho \vdash e_1 e_2 \rightarrow e'_1 e_2} \quad \frac{\rho \vdash e_2 \rightarrow e'_2}{\rho \vdash v_1 e_2 \rightarrow v_1 e'_2} \quad (4)$$

$$\frac{\rho[x \mapsto v] \vdash e \rightarrow e'}{\rho \vdash \lambda x(e) v \rightarrow e'} \quad \frac{\rho(x) = v}{\rho \vdash x \rightarrow v} \quad (5)$$

A further imperative extension would cause more modifications, adding the notion of stores. The transition relation now involves the syntax, the environment, and the store, and we obtain the following SOS rules:

$$\begin{aligned} e &::= \dots \mid l := v \\ v &::= \dots \mid l \mid () \\ l &\in Loc, \sigma \ \sigma' \in Store = Loc \rightarrow Val \end{aligned}$$

$$\frac{n = n_1 + n_2}{\rho \vdash \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle} \quad (6)$$

$$\frac{\rho \vdash \langle e_1, \sigma \rangle \rightarrow \langle e'_1, \sigma' \rangle}{\rho \vdash \langle e_1 + e_2, \sigma \rangle \rightarrow \langle e'_1 + e_2, \sigma' \rangle} \quad \frac{\rho \vdash \langle e_2, \sigma \rangle \rightarrow \langle e'_2, \sigma' \rangle}{\rho \vdash \langle v_1 + e_2, \sigma \rangle \rightarrow \langle v_1 + e'_2, \sigma' \rangle} \quad (7)$$

$$\frac{\rho \vdash \langle e_1, \sigma \rangle \rightarrow \langle e'_1, \sigma' \rangle}{\rho \vdash \langle e_1 e_2, \sigma \rangle \rightarrow \langle e'_1 e_2, \sigma' \rangle} \quad \frac{\rho \vdash \langle e_2, \sigma \rangle \rightarrow \langle e'_2, \sigma' \rangle}{\rho \vdash \langle v_1 e_2, \sigma \rangle \rightarrow \langle v_1 e'_2, \sigma' \rangle} \quad (8)$$

$$\frac{\rho[x \mapsto v] \vdash \langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle}{\rho \vdash \langle \lambda x(e) v, \sigma \rangle \rightarrow \langle e', \sigma' \rangle} \quad \frac{\rho(x) = v}{\rho \vdash \langle x, \sigma \rangle \rightarrow \langle v, \sigma \rangle} \quad (9)$$

$$\rho \vdash \langle l := v, \sigma \rangle \rightarrow \langle (), \sigma[l \mapsto v] \rangle \quad \frac{\sigma(l) = v}{\rho \vdash \langle l, \sigma \rangle \rightarrow \langle v, \sigma \rangle} \quad (10)$$

2.2 Label Transformers and the Modular SOS of Action Notation

To improve the modularity of SOS specifications, and in particular to avoid the need for such reformulations as illustrated above, Mosses has proposed the use of labels in a labelled transition system to encapsulate the semantic structures. Every semantic structure is a component in the label structure, changed and accessed via *set* and *get* functions, independently of the other components.

In categorical terms, the labels can be seen as the arrows of a category, where the semantic states are the objects. For this reason such a transition system is called an Arrow-Labelled Transition System (ALTS). The source $pre(\alpha)$ of a label α on a transition represents the state before the transition, and its target $post(\alpha)$ represents the state afterwards. Two transitions can be adjacent only when their labels are composable. Identity labels ι have the same source and target, so transitions labelled with them never affect the state.

A label represents the state of the system both before and after a transition, possibly with some more information associated with the transition itself (such as the usual synchronization signals of CCS).

Formally, SOS is based on the concept of labelled transition systems (LTS). An LTS is a structure $(\Gamma, T, A, \longrightarrow)$ where Γ is a set of configurations, $T \subseteq \Gamma$

is the set of terminal configurations, A is the set of labels and $\longrightarrow \subseteq \Gamma \times A \times \Gamma$ is the transition relation.

MSOS is based on the concept of ALTS, which are labelled transition systems where the set of labels A is the set of the arrows of a category. The objects of A , written $|A|$, are the states of the ALTS. The composition of arrows α_1 and α_2 is defined iff $post(\alpha_1) = pre(\alpha_2)$. The identity arrows of A are written I^A , or simply I when A is evident; they are called *silent* or *unobservable* arrows and are used to label transitions that merely reduce configurations. For two transitions $\gamma \xrightarrow{\alpha_1} \gamma', \gamma'' \xrightarrow{\alpha_2} \gamma'''$ to be adjacent, two conditions must be met, namely $\gamma' = \gamma''$, and $post(\alpha_1) = pre(\alpha_2)$. Terminating or infinite sequences of adjacent transitions then define the *computations* of the ALTS.

An ALTS can be reduced straightforwardly to a labelled transition system by moving back the states to the configuration and forgetting that the labels are the arrows of a category.

A key concept in MSOS is the one of *label transformers*. Label transformers assume that the category A comes equipped with additional structure, so that we can think of the arrows in A as having “components”. We can get to each component by using a particular index in a set of indices. That is, it is assumed that there are sets *Index*, of indices, and *Univ*, of information structures, and functions $set : A \times Index \times Univ \rightarrow A$ and $get : A \times Index \rightarrow Univ$, where the intuitive meaning is that $set(\alpha, i, u)$ changes the component of α indexed by i to u , leaving the rest unchanged; and $get(\alpha, i)$ yields the component of α indexed by i .

A label transformer T involves adding a fresh new index, say i , to the original set *Index*, and transforms the original category of labels A to the product category $T(A) = A \times C_T$. The original *set* and *get* functions are extended in the obvious way, that is, by treating the arrows $\beta \in C_T$ as the information stored in the new index i .

Mosses has identified three such label transformers that can be used as basic building blocks to extend the label categories of MSOS specification in a very modular way [13]. They correspond to the following three basic choices of C_T :

- a discrete category E , typically understood as a set of environments, giving rise to the label transformer **ContextInfo**(i, E).
- a cartesian product S^2 , understood as a binary relation, and viewed as a category with set of objects S and arrow composition given by relational composition. S is typically understood as a set of stores. This gives rise to the label transformer **MutableInfo**(i, S).
- a monoid (A, f, τ) , viewed as a one-object category, and is typically understood as a set of actions. The **EmittedInfo**(i, A, f, τ) label transformer rises with this choice.

MSOS specifications have two key ingredients: the labelled transition rules, that specify how configurations change, and the category of labels, that encapsulate the pre and post states and additional semantic information. As already mentioned, the category of labels can be extended by means of label transform-

ers without any need for modifying the previous given transition rules in any such extensions, that is, in a fully modular way.

We can exemplify the modularity of MSOS with the language introduced in Section 2.1. The MSOS labelled transition rules for that language can be specified as follows:

$$\frac{n = n_1 + n_2}{n_1 + n_2 \xrightarrow{\iota} n} \quad \frac{e_1 \xrightarrow{\alpha} e'_1}{e_1 + e_2 \xrightarrow{\alpha} e'_1 + e_2} \quad \frac{e_2 \xrightarrow{\alpha} e'_2}{v_1 + e_2 \xrightarrow{\alpha} v_1 + e'_2} \quad (11)$$

$$\frac{e_1 \xrightarrow{\alpha} e'_1}{e_1 \ e_2 \xrightarrow{\alpha} e'_1 \ e_2} \quad \frac{e_2 \xrightarrow{\alpha} e'_2}{v_1 \ e_2 \xrightarrow{\alpha} v_1 \ e'_2} \quad (12)$$

$$\frac{\alpha' = \text{set}(\alpha, \text{env}, \text{get}(\alpha, \text{env})[x \mapsto v]) \quad e \xrightarrow{\alpha'} e'}{\lambda x(e) \ v \xrightarrow{\alpha} e'} \quad (13)$$

$$\frac{\text{get}(\iota, \text{env})(x) = v}{x \xrightarrow{\iota} v} \quad (14)$$

$$\frac{\alpha = \text{set}_{\text{post}}(\iota, \text{sto}, \text{get}_{\text{pre}}(\iota, \text{sto})[l \mapsto v])}{l := v \xrightarrow{\alpha} ()} \quad (15)$$

$$\frac{\text{get}_{\text{pre}}(\iota, \text{sto}, l) = v}{l \xrightarrow{\iota} v} \quad (16)$$

The label category is the following nested application of label transformers:

$$\mathbf{MutableInfo}(\text{sto}, \text{Store})(\mathbf{ContextInfo}(\text{env}, \text{Environment})(1))$$

That is, $\text{Index} = \{\text{env}, \text{sto}\}$ corresponding to adding two new “components” to the trivial category 1 (consisting of just one object and one arrow), namely the arrows and the objects of the categories formed from the sets *Environment* and *Store* by the label applied transformers. The set_{post} and get_{pre} functions are auxiliary functions involved in the label transformer **MutableInfo**, used in the imperative language extension.

Note that the above rules did not require any change to accommodate the functional and imperative extensions and that they would not have to be changed at all in a further extension. To allow concurrency and message-passing, for instance, one would add a new semantic structure to the labels. The new rules would *set* and *get* data to and from the new index involved in the extension, but the previous rules wouldn’t have to be changed [13].

Action notation has been given an MSOS semantics [14]. Such an MSOS semantics is defined in a modular way for each facet. This involves defining the syntax of actions for each facet, the transition rules based on that syntax (so that a transition $\gamma \xrightarrow{\alpha} \gamma'$ involves syntactic structures γ and γ' defined by the facet’s syntax), and the corresponding category of labels α , which is typically built using one or more label transformers. The key advantage of this style of semantics over the previous SOS semantics for action notation [11] is its modularity, in that the addition of new constructs and facets to AN does not require any changes to the previously given semantic definitions.

3 Mapping Action Notation to Rewriting Logic

Rewriting logic has been proposed as a logical and semantic framework in [10]. Martí-Oliet and Meseguer present several ways to map specification languages (among other formal concepts) to rewriting logic via their operational semantics. In particular, they present two alternatives to map SOS to rewriting logic:

- the terms being rewritten are the SOS transition rules; or
- SOS transition rules are mapped directly to rewriting logic rules.

The latter alternative has the interesting characteristic of generating interpreters for the SOS definitions of the languages being mapped. Our embedding follows this idea. However, the current version of the Maude system does not support the definition of conditional rewrite rules whose conditions may involve other rewrites in addition to purely equational parts of the condition (this feature will be included in the next version of the Maude system). For this reason, we have implemented an extension to Maude, using its metalevel interface, which allows the specification and execution of more general conditional rules, with rewritings in their conditions. These more general rules are introduced with the keyword `rcrl`. They are used in the example of Section 3.3 and Section 3.1 explains the rewriting strategy for `rcrl` of rules.

This section explains how the operational semantics of action notation is mapped to rewriting logic. In other words, the application of the *semantics* function, introduced in Section 1, to AN_{MSOS} yields AN_{RL} .

The *semantics* function transforms an MSOS specification into a corresponding rewriting logic specification. The key idea in MSOS is the encapsulation of semantic information in the labels. For that reason, MSOS specifications are mapped to Maude's object-oriented specifications [3, Section 3.1], so that the semantic information is now encapsulated in objects, where attribute names exactly correspond to the indices in the label category (so that modularly adding new indices can be easily achieved by multiple class inheritance), and where the resulting configurations are formalized as composed actions (or yielders).

As explained in Section 2.2, MSOS has two main concepts: the category of labels and the rules. A rewrite theory consists of an equational specification and a set of rewrite rules. Therefore, to explain how AN_{MSOS} is transformed into AN_{RL} we have to explain how the resulting equational specification and rules are obtained. Section 3.2 describes how the category of labels is mapped to an equational theory described in the form of an object-oriented module with class specifications. This defines the equational part of AN_{RL} . Section 3.3 explains how MSOS rules are handled, producing the rewriting rules of AN_{RL} . But before let us give some more details about the `rcrl` rules.

3.1 Extending Maude with `rcrl` Rules

The Maude extension to support rewritings in the conditions of conditional rules has been implemented as a meta-interpreter to the Maude language. The meta-interpreter is composed of a parser, an evaluation function and a pretty-printer.

The parser and pretty-printer were implemented as extensions to Full-Maude [7], which is itself an extension to the Maude language, and supports object-oriented specifications and parameterized modules by means of an extensible module algebra.

The parser is an extension to the Full-Maude function `parseDecl`. It maps terms in the datatype `RuleDecl`, the grammar datatype for rule declarations, specified in the functional module `RCRL-GRAMMAR`, into terms of the datatype `ERule`, for extended rules which specifies the internal representation for `rcrl` rules, declared in the functional module `RCRL-DATATYPE`.

The pretty-printer function maps terms in the datatype `ERule` to quoted identifiers, terms of sort `Qid`, that are then displayed by the read-eval-print loop.

The evaluation function `meta-eval`, like the engine implemented descent functions [3], evaluates a given meta-term according to the given meta-module. The signature for `meta-eval` is as follows. The parameter of sort `MachineInt` specifies the number of rewrite steps that will be applied.

```
op meta-eval : Module Term MachineInt -> Term .
```

It has a simple implementation. A top-level command (`eval`) triggers the evaluation function which rewrites the given term using the current module's rule and equation sets. First the term is reduced, using the equations. Then, the evaluation function iterates over the set of rules and whenever unconditional rules or conditional rules without rewrites in the conditions are found, Maude's default rewriting strategy is applied via the built-in function `meta-apply` [3]. When a `rcrl` rule is found in the set, the meta-level function `apply-rcrl` is called. It checks whether the current `rcrl` rule may be applied, and if so, it iterates over the condition set of that rule, calling `meta-eval` recursively for every rewrite in the set. The substitutions returned by the evaluation function for a rewrite in the condition are passed to the following condition (so position in the condition set is relevant). The substitution resulting from the check of all the conditions is then applied to the right-hand-side term of the original `rcrl` rule. Finally the right-hand-side term is reduced according to the equation set of the current module and the resulting term is returned.

3.2 Mapping the Category of Labels

This mapping process is performed in the context of each facet of action notation and its corresponding MSOS specification. It associates an equational theory (in the form of a Maude object-oriented module) to the label category of the facet's MSOS specification.

Typically, such a label category is built up by applying several label transformers (see Section 2.2 and [14]). We can illustrate the transformation process by means of the *communicative* facet of AN, whose MSOS specification can be found in [14, Appendix A.6]. Created to allow the specification of concurrent languages, it defines how agents (processes) communicate asynchronously. In

their corresponding Maude specification, such agents are specified as objects, of a class **Agent**.

The agent specification in Maude is given by the following object-oriented module. The messages in the module correspond to the actions, the operators to yielders of the communicative facet, and attributes of **Agent** objects are in one-to-one correspondence with the indices of the label category in the MSOS semantics of the communicative facet.

```
(omod AGENT is
  pr MACHINE-INT .
  pr AGENT-ID .
  pr BUFFER .
  pr ACTION-SORTS .
  pr COMMUNICATING .

  class Agent | contractor : AgentId ,
                buffer : Buffer ,
                serial : MachineInt ,
                communicating : Communicating .

  msg send_ : Yelder -> Action .
  msg remove_ : Yelder -> Action .
  msg offer_ : Yelder -> Action .
  msg patiently_ : Action -> Action .

  op current'buffer : -> Yelder .
  op performing'agent : -> Yelder .
  op contracting'agent : -> Yelder .

  op new-agent : -> AgentId .
endom)
```

The above syntax introduces a Maude object-oriented module called **AGENT**. It imports several auxiliary modules, each with the keyword **pr** (protecting, as in OBJ3 [8]). Objects of class **Agent** are defined in the **class** declaration, listing their attributes and their corresponding sorts. For example, the **contractor** attribute must be of sort **AgentId**. An object of class **Agent** in a given state is then a record-like structure of the form

```
< o : Agent | contractor : A, buffer : B, serial : N,
    communicating : C >
```

where *o* is the agent's name, and where *A*, *B*, *N*, and *C* denote the semantic values stored in the corresponding attributes. After the class declaration, the syntactic forms of several messages of sort **Action** are declared, each with the keyword **msg**. Finally, the syntactic form of several other operators are declared, each with the keyword **op**. This entire specification can be *desugared* into an ordinary equational theory, by a general transformation process explained in detail in [7]. In this way, the transformation of the label category for a facet into an equational theory is accomplished.

A language specification might not use all AN facets. The semantic structure for a particular language is specified as a class which inherits only from the classes associated with the facets used by the language. That is, the modular combination of facets in AN is represented in its object-oriented translation as multiple class inheritance.

If one is specifying an imperative language, perhaps only the imperative and basic facets will be used. The semantic structure for that language will be a class, let us say, **State**, which inherits from the **ImperativeFacet** class. If afterwards, a major extension is desired, for example, adding concurrency to the language, the **State** class will then inherit from both the **CommunicativeFacet** class (the communicative facet's class, which is a specialization of the **Agent** and **BasicFacet** classes) and the **ImperativeFacet** class.

3.3 Mapping MSOS Rules

The second part of an MSOS specification consists of the specification of the rules. Applying the *semantics* function to the MSOS rules of the specification of a facet generates the rule part of the resulting rewrite theory of that facet.

The key idea in this mapping process is to associate a rewrite rule to each MSOS rule, where the configuration of the resulting rewrite rule is composed of the configuration of the original MSOS rule and the semantic information associated with the label of the MSOS rule. That is, the configuration changed by the resulting rewriting rule is composed of syntax (as in the MSOS rule) and the state and emitted information originally encapsulated in the label of the MSOS rule.

To illustrate the process, let us consider the MSOS rule for the **send** action, from the communicative facet. It has the following MSOS rule.

$$\begin{array}{c}
 Y \xrightarrow{\iota} ds \wedge n = \text{get}(\iota, \text{serial}) \wedge \\
 m = ds[\text{from } \text{get}(\iota, \text{performer})][\text{at } n] \wedge \\
 \alpha = \text{set}(\text{set}(\text{set}_{\text{post}}(\iota, \text{serial}, \text{successor}(n)), \text{communicating}, \text{set\_of}(m)), \\
 \text{commitment}, \text{true\_value}) \\
 \hline
 \text{send } Y \xrightarrow{\iota} \text{completed}
 \end{array}$$

Quoting [11, page 123]: the primitive action **send** Y , where Y yields a *sort* of message, initiates the transmission of a message. The usual form of Y is **a message [to Y_1] [containing Y_2]**, where Y_1 and Y_2 are individuals. The sort yielded by Y is implicitly restricted to messages from the performing agent, with the next local serial number, and this should determine an individual message.

Setting the *commitment* index to *true_value* specifies that this action cannot be undone if the subsequent action performance fails (cf. cut in Prolog).

As explained in Section 3.1, Maude has been extended with the so-called **rcrl** rules. They constitute the syntax of the rule produced by the transformation process. After the transformation, the states and emitted information associated with the label are now part of the configuration, and the following **rcrl** rule is generated for the above MSOS rule of the **send** action.

```

var y : Yielder . var aid : AgentId . var s : MachineInt .
var c : Communicating . var tv : TrueValue . var OS : ObjectSet .
var m? : Message? . var ds : DataSort . var ATTS : AttributeSet .

op send_ : Yielder -> Action .

rcrl [send] :
  send y
  { < aid : CommunicativeFacet | serial : s,
    communicating : c ,
    commitment : tv , ATTS > OS }
=>
  completed
  { < aid : CommunicativeFacet | serial : (s + 1),
    communicating : (m?,c) ,
    commitment : true value , ATTS > OS }
if
  ((y { < aid : CommunicativeFacet | serial : s,
    communicating : c ,
    commitment : tv , ATTS > } ) => ds) and
  (generate-message(ds [from aid] [at s]) => m?) and
  (m? /= nothing)
fi .

```

Note that the `CommunicativeFacet` class, as mentioned in Section 3.2, is already a specialization of the `Agent` class and the `BasicFacet` class; the `commitment` attribute, from the basic facet, is present.

The rule specifies that the configuration composed by the action `send y` and the object

```

< aid : CommunicativeFacet | serial : s, communicating : c ,
  commitment : tv >

```

rewrites to the configuration formed by the action `complete` and the object

```

< aid : CommunicativeFacet | serial : (s + 1), communicating : (m?,c) ,
  commitment : true value >

```

where the attributes not mentioned in the rule are captured by the `ATTS` variable, of type `AttributeSet` and any other objects involved in the configuration do not change their state. But the rule only applies if the configuration formed by `y` and the initial object rewrites to `ds2` and `ds` produces a message, when composed with the current object identifier and message number.

4 Concluding Remarks and Current Work

We conclude the paper by placing our proposal in the context of related work, summarizing the current state of the MAT system, and sketching a potential extension of action notation to support the notion of software component.

² Rules for yielders do not have an object pattern in the righthand side, because they do not change the state of an object.

4.1 Related Work

The Actress system [1,15], developed by Watt's group at the University of Glasgow, interprets action notation and compiles it into C. It deals with most of action notation, except the communicative facet.

Abaco [5] is an acronym for algebraic-based action compiler, which produces compilers (coded in C++) from action semantics descriptions. An unified algebras³ [11] translator is responsible for generating dynamic and static libraries, which are then used by the code generator and the generated compiler itself, respectively.

Also related to this work are the ASD [21] tools and the OASIS [18] compiler. The latter provides the most convincing evidence so far of the practical applicability of action-semantics-based compiler generation. It is capable of generating optimizing compilers for simple procedural and functional languages. The former was a set of tools developed by van Deursen and Mosses, which allowed specifying ASDs using a syntax-directed editor (or a plain text one), checking and interpreting them.

The MAT tool contributes to these efforts in the following ways:

- It is formally specified and implemented, following AN's Modular SOS, via the mapping of MSOS to rewriting logic. Such a mapping makes it easy to check the correctness of the embedding and supports formal analysis and verification of the resulting specification.
- It supports the full action notation described in [11], including the communicative facet.
- It preserves the modularity of the MSOS specifications. Specially, the rewrite rules associated to different facets do not have to be modified when those facets are combined. This is achieved by means of the multiple inheritance mechanism of classes in Maude object-oriented specifications.
- It allows experimentation with new semantic structures, through the addition of new facets to action notation, via MSOS. The component extension, sketched in Section 4.3 would be an example of a facet addition.

4.2 Current State of MAT

The Maude Action Tool is currently in a prototype state. We have implemented most of the action notation, including the communicative facet, producing manually the AN_{RL} specification. Once the prototype is finished (i.e., the whole action notation is specified in Maude), it will be able to interpret a program P , written in a language L , according to L 's action semantics. Also, at present the specification of L_{RL} has to be manually produced for each language L .

MAT provides a simplified version of the Data Notation described in [11, Appendix E], which parameterizes AN. Any additional datatypes needed for a particular AS specification can be specified, equationally, as Maude modules.

³ Unified algebras is the algebraic framework used in action semantics type system.

The next step is the encoding of the transformation process for MSOS specifications (the function *semantics*) and the *desugaring* function (see Section 1). The *semantics* function will be crucial for the experiments with extensions to action notation, like the one sketched in Section 4.3, since it will allow the input of MSOS specifications into the system.

4.3 The Component Facet

The original operational semantics of action notation was, though precise, monolithic. In [16], Musicante and Mosses proposed an extension to the communicative facet with shared memory, which required a major change in the operational specification of the kernel of action notation.

Extensions and changes in the new MSOS specification of AN do not imply major changes: they can be made local to a facet. If one decides to adopt a new concurrency model, the other facets will not need to be re-specified, since they are self contained in general.

Stimulated by the new modular semantic description, we plan to extend the framework with a new semantic structure, namely, components. Software components are currently being used as the main technique to achieve interoperability among reusable pieces of software. The work described in [17,19,9] represents just a few of them. However, work on components still lacks formalization.

We propose to formalize the notion of software component in an extension of the current action notation, equipping the framework with a mechanism to allow the description of component-based languages. Components written in a language whose semantics is given in terms of the extended action notation will interoperate by construction. The mechanism will also allow connection with already existing components. The new facets will be as follows:

- *synchronous communication*. This specialization of the communicative facet will add the notion of synchronization events.
- *thread*. This facet adds the notion of lightweight processes, known as threads. The main difference between threads and the agents of the communicative facet is that threads share a memory.
- *service*. A service is a special action, that may be dynamically required, and that must satisfy a given specification of its pre- and post-conditions.
- *component*. Finally, the component facet defines the concept of a component, which is a structure with a well-defined interface of required and provided services. Also, a component is (internally) a multi-threaded unit that may signalize and listen to events.

Acknowledgements Braga would like to thank Francisco Durán for his (numerous) explanations about Maude and Full-Maude; to Alberto Verdejo for his suggestions to improve the `rcr1` strategy and for the partial support from CAPES grant No. CBE/PDEE 0009/1998. This work has been supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by NSF through Grant CCR-9900334, and by

ONR Contract N00012-99-C-0198. In addition, Mosses gratefully acknowledges support from BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark.

References

1. D. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction, Paderborn*, volume 641 of *LNCS*, pages 95–109. Springer-Verlag, 1992.
2. M. Clavel. *Reflection in General Logics and in Rewriting Logic*. PhD thesis, University of Navarre, 1998.
3. M. Clavel, F. Durán, S. Eker, N. Martí-Oliet, P. Lincoln, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, <http://maude.csl.sri.com>, January 1999.
4. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Sumazu, Japan. CafeOBJ Project*, April 1998.
5. L. C. de Sousa Menezes and H. P. de Moura. The Abaco system an algebraic based action compiler. In A. Haeberer, editor, *AMAST'98, Proc. 7th Intl. Conf. on Algebraic Methodology and Software Technology, Amazonia, Brazil*, volume 1548 of *LNCS*, pages 527–529. Springer, 1999.
6. G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DISCEX 2000, Proc. Darpa Information Survivability Conference and Exposition, Hilton Head, South Carolina*, volume 1, pages 251–265. IEEE Computer Society Press, January 2000.
7. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Escuela Técnica Superior de Ingeniería Informática, 1999.
8. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, 1992. To appear in J.A. Goguen and G.R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Kluwer, 2000.
9. R. Ibrahim and C. Szyperski. Formalization of component object model (COM) - the COMEL language. In *ECOOP'98 PhD Workshop*, July 1998.
10. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, August 1993.
11. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
12. P. D. Mosses. Theory and practice of action semantics. In *MFCS'96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.
13. P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. The full version appears as Tech. Report RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus.
14. P. D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.

15. H. Moura and D. A. Watt. Action transformations in the Actress compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *LNCS*, pages 16–30. Springer-Verlag, 1994.
16. M. A. Musicante and P. D. Mosses. Communicative action notation with shared storage. Tech. Mono PB-452, Dept. of Computer Science, Univ. of Aarhus, 1993.
17. Object Management Group. *The Common Object Request Broker Architecture and Specification - Revision 2.0*, July 1995. <http://www.omg.org>.
18. P. Ørbæk. OASIS: An optimizing action-based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *LNCS*, pages 1–15. Springer-Verlag, 1994.
19. J. Penix and P. Alexander. Toward automated component adaptation. In *Ninth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, June 1997. <http://www.ececs.uc.edu/~kbse/pub/ps/seke97.ps>.
20. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN - 19, Computer Science Department, Aarhus University, 1981.
21. A. van Deursen and P. D. Mosses. ASD: The action semantic description tools. In *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 579–582. Springer-Verlag, 1996.
22. K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain Specific Languages*, pages 157–170. The USENIX Association, 1997. <http://www.cl.cam.ac.uk/users/kw217/research/msc/thesis/>.

The Extensibility of Maude's Module Algebra^{*}

Francisco Durán

Universidad de Málaga, Málaga, Spain
duran@lcc.uma.es

Abstract. The reflective capabilities of rewriting logic and their efficient implementation in the Maude language can be exploited to endow a reflective language like Maude with a module algebra in which structured modules can be combined and transformed by means of a rich collection of module operations. We have followed this approach to use the specification of such a module algebra as its implementation, including a user interface and an execution environment for it. The high level at which the specification of the module algebra has been given makes this approach particularly attractive when compared to conventional implementations, because of its shorter development time and the greater flexibility, maintainability, and extensibility that it affords. We explain the general principles of the reflective design of the module algebra, focusing in its extensibility, and illustrate some of the possibilities for defining new module operations.

1 Introduction

Structuring mechanisms for building large specifications in a modular fashion are one of the basic components of most specification languages. Module operations can be seen as functions taking modules as arguments and returning other modules as results, which is in fact the idea behind the approaches followed in languages like Clear [2], ACT-ONE [15], Larch [19], OBJ3 [18], CASL [8], and many others. The structuring mechanisms belong to the *metalevel* of the logic in question. This means that, although module operations, and in particular parameterization, provide certain “higher-order” capabilities [17], such capabilities remain limited in their applicability in most specification languages. In fact, the user has a fixed repertoire of operations available and cannot express them, extend them, or reason about them *within the logic*.

The introduction of the notion of a *reflective logic* [6] can drastically change this situation. Informally, a reflective logic is a logic in which important aspects of its metatheory, including the theories themselves, can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic that can be faithfully interpreted in itself, giving access to its metatheory.

A reflective logic opens up many new possibilities for module composition. Module operations that transform, combine, or manipulate modules can be defined within the logic itself, providing higher-order capabilities and also allowing

^{*} Partially supported by CICYT project TIC98-0445-C03-03.

us to reason about such operations within the logic. A *module algebra* specified in such a way can then be easily modified, thanks to the high level at which the module operations are defined, or extended, by defining new operations in the same way. Furthermore, if the logic is efficiently executable then the formal specification of module composition operations can itself be used as a reasonable implementation. In addition, it is also possible to use such an implementation in order to create a formal environment for the logic, with tools for formal analysis, transformation, and theorem proving.

This is the approach we have followed to define the module operations for the Maude system [3]. Maude is a high-level reflective language and high-performance system (reaching 1.66 million rewrites per second on a 500-MHz Alpha for some applications) supporting both membership equational logic [23] and rewriting logic [21] specification and programming for a wide range of applications. Rewriting logic is reflective [7], and has very good properties as a logical framework, in which many other logics and many semantic formalisms can be naturally represented [20]. Maude has been influenced in important ways by the OBJ3 language [18], which can be regarded as an equational logic sublanguage.

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective, making the metatheory of rewriting logic accessible to the user in a clear and principled way. We have defined an extensible module algebra for rewriting logic, and have implemented such a design using Maude's reflective capabilities. The language resulting from extending Maude with this module algebra is called Full Maude. The semantics of this module algebra has been given at two different levels: A logic-independent categorical semantics, which was presented in [13,11], and a rewriting logic semantics given by this algebraic specification, which was presented in [14,11]. Full Maude and its documentation are part of the Maude language release, which is available free of charge from its web site at <http://maude.csl.sri.com>.

The specification of Full Maude in Maude involves not only a set of module operations. It specifies in fact a complete language, including its syntax and an execution environment for it. The fact that Full Maude has Maude as a "sublanguage" does simplify the representation map between them, but the same methodology can be used for many other languages and logics [14,11]. Once we have an appropriate representation map expressing a given logic in the rewriting logic framework, we can then define an execution environment for such a logic in the same way.

In Maude there are two kinds of *modules*, both with initial semantics: functional modules, which are equational theories, and system modules, which are rewrite theories. Full Maude extends Maude with object-oriented notation and with a number of module operations, namely, module hierarchies, module renaming, parameterization of modules, and transformation of object-oriented modules into system modules¹. Full Maude also provides *theories*, with loose semantics, which can be functional, system, or object-oriented. The module operations

¹ Object-oriented modules are used to specify object-oriented systems but can be desugared into system modules [22,12,11].

available in Full Maude are inspired and mirror quite closely those in OBJ3 [18], although with some significant differences which we do not explain here due to the lack of space. A detailed discussion of the language design of Full Maude can be found in [3,11].

Typically, theory composition operations begin with theories structured in some way and result in an unstructured, or less structured, specification. That is, structured theories are often “flattened” when being composed. There are however good reasons for preserving their structure. Besides the obvious understandability and design documentation reasons, it is often very useful to consider theory-building operations whose results are structured theories. For example, refining a software design can be best understood as refining structured theories [24]. There are also more intrinsic reasons, namely, when the semantics associated to a structured module *essentially depends on its structure*. For example, we often want to associate to the inclusion of a parameter theory into the body of a parameterized specification a *freeness constraint*, requiring that the models of the body are free extensions of the models of the parameter.

Although a number of concepts and techniques have been suggested both at the theoretical and specification language levels to keep and use the necessary amount of structure for specific purposes, the most satisfactory way of addressing the need for preserving structure is to make structured theories *first-class citizens*. Categorical constructions giving semantics to structured theories and to operations on them have been given in [13,11]. The most basic form of structured theory is that of a *hierarchy* of theory inclusions, in the sense that more complex forms of structured theories can often be *normalized* to theory hierarchies [12,11], perhaps keeping some additional information such as freeness constraints. This is one of the central ideas in our module algebra. We do not have explicit constructors for diagrams in the language, as it is the case in systems like SPECWARE [24]. Instead, we represent module inclusions in these normalized structures as modules referring to other modules in a database.

The module algebra that we are proposing can be extended in different ways. We can, for example, define new module combining operations, or new module transforming operations in the style of the transformation from object-oriented modules to system modules, which would allow us to define new types of modules or other types of structuring units for our language. But there are many more possibilities. For example, an important area for defining new module operations is viewing *formal tools* as appropriate module operations that analyze and transform specifications.

In this paper we wish to emphasize the extensibility of the module algebra by defining new module combining and manipulating operations. We illustrate some of the possibilities for extensions of this kind with two examples:

- a module expression for union of modules, that is, the classical sum or union operation taking two modules and returning the result of combining them;
- a module expression $\text{TUPLE}(n)$ such that, given a nonzero natural number n , it generates a parameterized functional module specifying tuples of the corresponding size.

The rest of the paper is structured as follows. In Section 2 we introduce very briefly the notion of reflection in rewriting logic and in Maude, and present the main abstract data types involved in the specification of the module algebra. In Section 3 we present the general schema for the evaluation of module expressions. Section 4 describes the way of extending the specification of the module algebra for dealing with new module operations. We finish with some conclusions and future directions in Section 5.

2 Reflection in Maude and Module Data Types

Rewriting logic is reflective [7] in the precise sense that there is a finitely presented rewrite theory $\mathcal{U}$ which is *universal*, that is, for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence:

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \bar{t}' \rangle,$$

where $\overline{\mathcal{R}}$, $\bar{t}$, and $\bar{t}'$ are terms representing, respectively, $\mathcal{R}$, t , and t' as data elements of $\mathcal{U}$.

Maude's design and implementation [3] have systematically exploited the reflective capabilities of rewriting logic, providing key features of the universal theory $\mathcal{U}$ in its built-in module **META-LEVEL**. In particular, **META-LEVEL** has sorts **Term** and **Module**, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\bar{t}$ of sort **Term** and a term $\overline{\mathcal{R}}$ of sort **Module**. For example, the signature for representing terms is as follows:

```
subsort Qid < Term < TermList .
op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
```

The declaration making **Qid** a subsort of **Term** is used to represent variables by the corresponding quoted identifiers. Thus, for example, the variable **N** is represented by '**N**'. The operator **{_}_** is used for representing constants, with the constant in quoted form as first argument and the sort of the constant, also in quoted form, as second argument. For example, the constant **0** in the module **NAT** below is represented as '{**0**}'**Nat**. The operator **_[_]** corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form, and the second argument the list of its subterms, where list concatenation is denoted **_,_**. For example, the term **0 + s 0** of sort **Nat** in the module **NAT** below is metarepresented as '**+_**[_{'**0**'}**Nat**, '**s**[_{'**0**'}**Nat**]].

The module **META-LEVEL** also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, **meta-apply**, **meta-reduce**, and **meta-rewrite**, and also generic parsing and pretty printing functions **meta-parse** and **meta-pretty-print** [3].

In order to be able to handle the different types of modules, theories, and views, we have extended the module **META-LEVEL** with several data types, and

with several transformation functions that act on a database in which the declarations being entered are stored. Thus, the main data types are: **Unit**, for the different types of theories and modules; **View** for views; and **Database** for the database of module, theory, and view definitions.

2.1 The Abstract Data Type Unit

The different types of modules and theories are metarepresented in a syntax very similar to the original user syntax as terms of sort **Unit**. The abstract data type **Unit** can be seen as an extension of the predefined sort **Module** in **META-LEVEL**. We also have subsorts of **Unit** for the different types of theories and modules. For example, we have sorts **StrOModule** and **StrSModule** for structured object-oriented modules and structured system modules, respectively. There are six different types of units, namely, functional, system, and object-oriented modules and theories, which can be parameterized and can import module expressions. There are constructors for each of them in their respective sorts. Modules and theories of any kind are considered to be elements in specific subsorts of the sort **Unit**. The structure of the hierarchy of unit sorts is depicted in Figure 1.

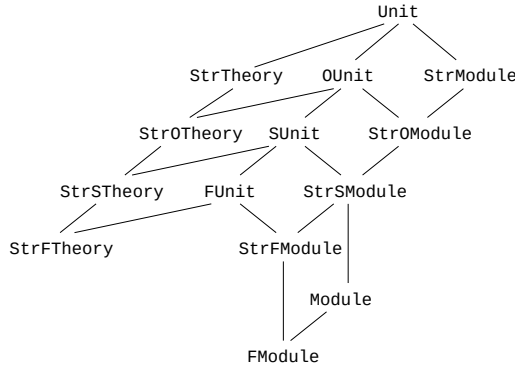


Fig. 1. Hierarchy of Unit Sorts.

Besides sorts for the different types of modules and theories, there are sorts whose terms represent the different items that can appear in a module. We illustrate the syntax for the top-level operators representing the different types of modules and theories by showing the top-level operator for functional modules.

```

op fmod_is_____endfm : ModName ParameterList EImportList
  ESortDecl ESubsortDeclSet EOpDeclSet EVarDeclSet EMembAxSet
  EquationSet -> StrFModule .
  
```

To motivate the general syntax for representing modules, we illustrate it with a simple example, namely, a functional module **NAT** for natural numbers with zero and successor, and with a commutative and associative addition operator with 0 as its identity element.

```

fmod NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+ : Nat Nat -> Nat [comm assoc id: 0] .
  vars N M : Nat .
  eq s N + s M = s s (N + M) .
endfm

```

The representation $\overline{\text{NAT}}$ of NAT is the following term of sort `StrFModule`.

```

fmod 'NAT is
  nilParameterList
  nil
  sorts 'Nat .
  none
  op '0 : nil -> 'Nat [none] .
  op 's_ : 'Nat -> 'Nat [none] .
  op '+_ : 'Nat 'Nat -> 'Nat [comm assoc id({'0}'Nat)] .
  var 'N : 'Nat . var 'M : 'Nat .
  none
  eq '+_['s_['N], 's_['M]] = 's_['s_['+_['N, 'M]]] .
endfm

```

Since NAT has no list of parameters, no list of imported submodules, no list of subsort declarations, and no membership axioms, and since the zero and successor operators have no attributes, those fields are filled with the constants representing empty sets or lists of the appropriate sorts.

2.2 The Abstract Data Type View

Basically, the data elements of sort `View`, representing theory maps, are expressions containing the name of the view, the names of the source and target units, and a set of terms representing the maps asserting how the given target unit is claimed to satisfy the source theory. The syntax for the top-level operator representing views is as follows:

```

op view : ViewExp ModExp ModExp ViewMapSet -> View .

```

There are also constructors for the different types of maps that can be used in a view, all of which generate terms of sort `ViewMap`. Full Maude supports all the maps in OBJ3. For example, there can be maps from operators to derived operators, that is, to terms with variables. Some maps have also been extended, for example, operator maps allow the specification of the operator's arity and coarity. New maps, for example, maps related to object-oriented notions, have also been incorporated.

Let us motivate the general Full Maude syntax for representing views by illustrating it with a simple example. Given the theory `TRIV` of sets below, with just a sort `Elt`, and given the previous module `NAT` of natural numbers, we can define a view `Nat` from `TRIV` to `NAT` as follows.

```

(fth TRIV is
  sort Elt .
endfth)

(view Nat from TRIV to NAT is
  sort Elt to Nat .
endv)

```

The internal metarepresentation $\overline{\text{Nat}}$ of the above view `Nat` is the following term of sort `View`:

```
view('Nat, 'TRIV, 'NAT, sort 'Elt to 'Nat .)
```

This internal representation is stored in the database when the user enters the above view `Nat`. In case it is needed for the evaluation of some module expression, it will be retrieved from the database and then used.

2.3 The Abstract Data Type Database

In order to be able to refer to modules by name, which is extremely useful for module definition purposes at the user level, the evaluation of module expressions takes place in the context of a database, in which we keep information about the modules, theories, and views already introduced in the system, and also about those generated internally. The database is also used to keep structured modules as modules referring to the modules in the database that they include, which may themselves refer to other modules.

The information in the database is stored as a set of elements of sort `UnitInfo` and `ViewInfo`, in which we hold, respectively, the information concerning units and views. In addition to this set of information cells for units and views, we also keep lists with the names of all the units and views in the database, and a list of quoted identifiers in order to be able to simplify the generation of messages.

```
op database : InfoSet ModNameSet ViewExpSet QidList -> Database .
```

3 Operations on Structured Modules

Our module operations take structured modules as arguments and return structured modules as results. For example, the union of two structured modules results in a new structured module in which the structure of the parameter modules is maintained. Note that terms of sort `Unit` are structured modules, where the structure is given by module expressions that refer by name to other modules, or combinations of modules, which are stored in the database. However, we define module operations on *normalized* structures, which allows us to add new operations on structured modules by following a general basic schema. As pointed out in the introduction, these normal forms, terms of sort `Unit` as well, are given by hierarchies of module inclusions, to which more complex forms of structured modules are transformed.

To be able to apply the predefined functions in the module **META-LEVEL** to the modules entered to Full Maude, for example, for execution purposes, we need to map these normalized structures into terms of sort **Module**, which are then compiled into the rewrite engine.

In our approach, transforming a module from its possibly complex structured version to its unstructured form is therefore a two-step process. First, the evaluation of the module expressions results in an intermediate form, in which only simple inclusion relationships appear among the modules. This first step can be seen as the reduction of a structured specification to its *structured normal form*, which is carried out by the **normalization** function. It takes place in the database, and yields as result a possibly modified version of the database state. Then, in a second step, this structured normal form is flattened into an unstructured specification, resulting in a term of sort **Module**.

The evaluation of structured modules and theories is accomplished by the function **evalUnit**, which takes a unit and a database state and returns the database after including the resulting module or theory in it:

```
op evalUnit : Unit Database -> Database .
```

3.1 The General Schema for the Evaluation of Units

The schema followed for evaluating module expressions is very simple and can be presented in a generic way. In our design, the evaluation of a unit consists in first evaluating any module expression in it, which may result in the generation of new modules with such module expressions as *names*, and then reducing its structure to a canonical form in which only unit inclusions appear, that is, to a unit hierarchy.

We illustrate this process with a very simple example, namely, the importation of a parameterized module instantiated by a view. Let us consider the following functional module **LIST** of lists, which is parameterized by the functional theory **TRIV** given in Section 2.2.

```
(fmod LIST[X :: TRIV] is
  sort List[X] .
  subsort Elt.X < List[X] .
  op nil : -> List[X] .
  op _ : List[X] List[X] -> List[X] [assoc id: nil] .
endfm)
```

As in OBJ3 [18], the theories appearing in the interface of a parameterized module must be labeled. This label is then used to qualify all occurrences of the sorts from the parameters being used in the body of the parameterized module. The structured module **LIST** is understood as the inclusion of the parameter **X :: TRIV** into the top module **LIST**. **X :: TRIV** is a copy of **TRIV** in which each of its sorts is qualified by the label **X**, that is, in this case its only sort **Elt** has been renamed to **Elt.X**. This amounts to a module transformation denoted by ϕ in Figure 2.

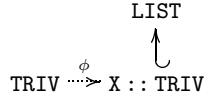


Fig. 2. Structure of the parameterized module LIST.

In Full Maude, we follow the convention of parameterizing the names of the sorts declared in parameterized modules with the labels of the parameter theories, so that when a module is instantiated the parameterized sorts in it become parameterized by the views used in the instantiation. Thus, for example, if the module LIST is instantiated with the view Nat above, the sort List[X] becomes List[Nat].

Let us consider now the following module F00, which just imports the module expression LIST[Nat], that is, imports the module LIST instantiated by the view Nat given in Section 2.2.

```
(fmod F00 is protecting LIST[Nat] . endfm)
```

The instantiating process is summarized in Figure 3. Basically, the evaluation of the expression LIST[Nat] results in the creation of a module, with such an expression as its name, resulting from binding the names of sorts, operators, etc. in the actual parameter TRIV to the corresponding sorts, operators (or expressions), etc. in the target NAT as indicated by the view Nat. Then, the new module LIST[Nat] is included as a submodule of F00. The structured module resulting from the evaluation of the module expression LIST[Nat] is as follows:

```
fmod LIST[Nat] is
  including NAT .
  sorts List[Nat] .
  subsort Nat < List[Nat] .
  op nil : -> List[Nat] .
  op __ : List[Nat] List[Nat] -> List[Nat] [id: nil assoc] .
endfm
```

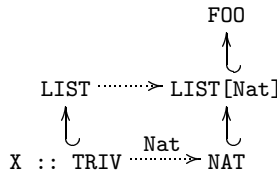


Fig. 3. Structure of module F00.

The flattening of the normalized structure is accomplished following the tradition of the Clear/OBJ family of languages, in which specification structuring is based on the categorical concept of *colimit* [2,10]. However, instead of considering the category of specifications and specification morphisms [2], in our semantics flattening is understood as a colimit in the category of specifications and inclusions of specifications. The colimit of a diagram in this category coincides with the set-theoretic union of the theories in the diagram.

3.2 The Evaluation of Module Expressions

The evaluation of a module expression in a database may produce the creation of a new module, whose name is given by the module expression itself, as well as the evaluation of other module expressions contained in the modules involved in the process. This is the case, for example, for the renaming of modules, in which not only the top module is renamed but, perhaps, some of its submodules as well; it is also the case for the instantiation of parameterized modules, where the module being instantiated may contain submodules which are parameterized by some of the parameter theories of the parameterized module in which they are imported. The modules generated as the result of the evaluation of a module expression are introduced in the database, which is then returned. For example, the evaluation of the module expression `LIST[Nat]` above in a given database state begins by retrieving the module `LIST` and the view `Nat` from the database, and builds a module with name `LIST[Nat]`, which is then entered into the database.

Module expressions are evaluated using the `evalModExp` function, which, given the sort `ModExp` for representing module expressions, is declared as follows.

```
op evalModExp : ModExp Database -> Database .
```

There are equations giving semantics to `evalModExp` on each of the constructors of module expressions. In addition to the appropriate declarations, the introduction of a new module expression operator requires new equations defining the behavior of `evalModExp` on such an operator.

4 Extending Full Maude

We illustrate in this section some of the ways in which the module algebra of Full Maude can be extended by defining new module expressions and integrating them with the rest of the system. As an example of module-combining operations we present the classical sum, or union, of module expressions; and as another example illustrating the possibility of defining completely new module operations we present a module expression such that, given any nonzero natural number, it generates a parameterized module specifying tuples of the corresponding size. The evaluation of these new module expressions follows the general schema for the evaluation of module expressions discussed in Section 3.2, that is, a new module with such module expression as name is created, giving to the expression the intended semantics.

In order to add any new module expressions the first thing we must do is to extend the signature of Full Maude with the declarations for the syntax of the new module expressions. Moreover, the declaration of any new kind of module expression must come together with the definition of several functions on the new module operator. We shall focus here on the specification of the `evalModExp` function for these new module expression operators. See [11] for a detailed discussion on the rest of definitions required.

4.1 The Union Module Expression

The syntax used for the union of module expressions is

```
op _+_ : ModExp ModExp -> ModExp [assoc comm] .
```

The evaluation of a union module expression results in the creation of a new unit, with such a module expression as its name, which imports the two module expressions being combined. Note, however, that the unit being created has to be of the right type. The new unit will be generated having one type or another, depending on the types of the arguments of the union module expression.

The function `rightEmptyUnit` generates an empty unit of the highest of the sorts of its two arguments. For example, the union of an object-oriented module and a functional module will be an object-oriented module. If one of the two module expressions corresponds to a theory, then a theory is generated.

Therefore, assuming variables `ME` and `ME'` of sort `ModExp` and a variable `DB` of sort `Database`, and with `addImportList`, `setName`, and `getUnit` functions to, respectively, add a list of importations to a module, set the name of a module, and retrieve a unit with a particular name from a database state, the equation defining the semantics of `evalModExp` for the union module expression is as follows.

```
ceq evalModExp(ME + ME', DB)
  = evalUnit(
    addImportList((including ME .) (including ME' .)),
    setName(
      rightEmptyUnit(
        getUnit(ME, evalModExp(ME, evalModExp(ME', DB))),
        ME + ME')),
    evalModExp(ME, evalModExp(ME', DB)))
  if not unitInDatabase(ME + ME', DB) .
```

4.2 The n -Tuple Module Expression

The syntax used for the n -tuple module expression is as follows:

```
op TUPLE(_) : NzNat -> ModExp .
```

Its evaluation consists in the generation of a parameterized functional module with the number of `TRIV` parameters specified by the argument. A sort for tuples of such size, and the corresponding constructor and selector operators, are also defined. For example, the module expression `TUPLE(2)` produces as its result the following module.

```
fmod TUPLE(2)[C1 :: TRIV, C2 :: TRIV] is
  sorts Tuple[C1, C2] .
  op (_,_) : Elt.C1 Elt.C2 -> Tuple[C1, C2] .
  op p1_ : Tuple[C1, C2] -> Elt.C1 .
  op p2_ : Tuple[C1, C2] -> Elt.C2 .
  var E1 : Elt.C1 . var E2 : Elt.C2 .
  eq p1(E1, E2) = E1 .
  eq p2(E1, E2) = E2 .
endfm
```

Even though the n -tuple module expression is in principle of a completely different nature than the usual Clear/OBJ module operations, the way of handling it is the same as the way of handling any other module expression. Its evaluation produces a new unit, a parameterized functional module in this case, with the module expression as name.

The equation for the `evalModExp` is reduced to the creation of a module as indicated above. Given variables `N` and `DB` of sorts `NzNat` and `Database`, respectively, and given some auxiliary functions to generate the different declarations in the module, the equation defining the semantics of `evalModExp` for the n -tuple module expression is as follows.

```
ceq evalModExp(TUPLE(N), DB)
  = evalUnit(
    fmod TUPLE(N) is tupleParameterList(N)
      tupleImportList(N)
      sorts tupleSortSet(N) .
      none
      tupleOpDeclSet(N)
      tupleVarDeclSet(N)
      none
      tupleEquationSet(N)
    endfm,
    DB)
  if not unitInDatabase(TUPLE(N), DB) .
```

4.3 Other Extensions

We can extend Full Maude in different ways. Indeed, we plan a number of such extensions for the module algebra, for example, the addition of parameterized theories and views, module operations to hide sorts and operators, and operations to redefine and remove messages in object-oriented modules.

An important area for defining new module operations is viewing *formal tools* as appropriate module operations that analyze and transform specifications. Substantial examples of this kind are the inductive theorem prover developed by Clavel *et al.* [4] and the Church-Rosser checker presented in [11,4]. Both tools can be used to prove properties of equational specifications in Maude². They are written entirely in Maude and are in fact *executable specifications* in rewriting logic of the formal inference systems that they implement.

Both tools have a *reflective* design. Indeed, the fact that rewriting logic is a reflective logic and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in both tools [5,4]. The reflective capabilities of Maude have also been used for building user interfaces and complete execution environments for these tools. In fact, the execution environment of Full Maude has been extended so that the Church-Rosser checker tool has been completely integrated within it [11].

5 Conclusions and Future Directions

We have described how the reflective capabilities of rewriting logic and their efficient implementation in the Maude language can be exploited to endow a reflective language like Maude with an extensible module algebra in which structured modules can be combined and transformed by a rich collection of module operations. This has been the approach followed in order to use the specification of such a module algebra as its implementation.

The Full Maude system includes a user interface and an execution environment for it, which have also been implemented using the reflective capabilities of Maude. The reasonable efficiency of the resulting system has allowed us to include it as part of the distribution package of Maude.

The design of Full Maude is just the result of using the general methodology presented in [14,11] to the case of Maude, but the same approach could be followed for defining module algebras and execution environments for other languages and logics, once we have the appropriate representation map. The high level at which the specification of the module algebra has been given makes this approach particularly attractive when compared to conventional implementations, because of its shorter development time and the greater flexibility, maintainability, and extensibility that it affords.

With this specification we have endowed Full Maude with a module algebra in the Clear/OBJ tradition, including module hierarchies, parameterized modules, views, and module expressions. In addition, Full Maude also allows the use of modules with object-oriented notation [22,12,11].

The module algebra thus obtained is easily *extensible*: A user may easily define new module transformations and module operations; that is, the user can not only do metaprogramming by using the module operations provided in

² Thanks to the efforts of our colleagues in Japan, both tools have been integrated within the overall Cafe environment [16] and can be used to prove formal properties of order-sorted equational specifications in CafeOBJ [9].

the module algebra of the language, but can also do *meta-metaprogramming* by defining new ones. We have illustrated some of the possibilities for extending the module algebra by defining some new module expressions. Furthermore, given the reflective design of Maude, it is even possible to change both the module operations provided in the basic syntax of the language and that very syntax itself.

The Full Maude specification described here is part of an overall reflective and extensible design for the Maude system, in which key components of the system are written in Maude itself and can be modified or extended with relative ease. In particular, we plan a number of such extensions for the module algebra, including the addition of parameterized theories and views, and the development of other module composition and module transformation operations. We also plan to consider module operations to hide sorts and operators, and operations to redefine and remove messages in object-oriented modules.

Another research direction we plan to pursue is formal reasoning about the properties of module algebra operations. Some promising first steps in this direction are the proofs of metalogical properties carried out in the Maude inductive theorem prover by Basin, Clavel, and Meseguer [1]. It would be very interesting to exploit such techniques for proving module algebra properties, and in particular to relate in a systematic way the categorical semantics given in [13] with the Maude specification of the corresponding module operations.

Acknowledgments

The author is grateful to José Meseguer for his advice and collaboration in the long-term joint project of which this work is a part. We are very thankful for the many fruitful discussions on Full Maude with the other members of the Maude team, Manuel Clavel, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, and José Quesada, with whom we have designed and built the Maude language and system.

References

1. D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. In *Proc. of Workshop on Logical Frameworks and Meta-languages*, 1999. Available at <http://www.cs.bell-labs.com/~felty/LFM99/>.
2. R. Burstall and J. Goguen. The semantics of Clear, a specification language. In D. Bjørner, ed., *Proc. of 1979 Copenhagen Winter School on Abstract Software Specification*, vol. 86 of *Lecture Notes in Computer Science*, pp. 292–332. Springer-Verlag, 1980.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Manuscript, SRI International, January 1999. Available at <http://maude.csl.sri.com>.
4. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational logic tools by reflection in rewriting logic. In *Proc. of CafeOBJ Symposium'98*. CafeOBJ Project, 1998.

5. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, eds., *Formal Methods (Vol. II)*, vol. 1709 of *Lecture Notes in Computer Science*, pp. 1684–1704. Springer-Verlag, 1999.
6. M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, ed., *Proc. of Reflection'96*, pp. 263–288, 1996.
7. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, ed., *Proc. of Workshop on Rewriting Logic and its Applications*, vol. 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at <http://www.elsevier.nl/locate/entcs/volume4.html>.
8. CoFI Task Group on Language Design. CASL—The common algebraic specification language, v. 1.0. Available at <http://www.brics.dk/Projects/CoFI>, 1998.
9. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.
10. R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In G. Huet, G. Plotkin, and C. Jones, eds., *Proc. of Workshop on Logical Frameworks*, pp. 83–130. Cambridge University Press, 1991.
11. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, June 1999.
12. F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner, eds., *Proc. of Workshop on Rewriting Logic and its Applications*, vol. 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at <http://www.elsevier.nl/locate/entcs/volume15.html>.
13. F. Durán and J. Meseguer. Structured theories and institutions. In M. Hofmann, G. Rosolini, and D. Pavlović, eds., *Proc. of Conference on Category Theory and Computer Science*, vol. 29 of *Electronic Notes in Theoretical Computer Science*, pp. 71–90. Elsevier, 1999. Available at <http://www.elsevier.nl/locate/entcs/-volume29.html>.
14. F. Durán and J. Meseguer. Structured theories and Maude's module algebra. In *Proc. of Workshop on Algebraic Development Techniques (WADT'99)*, 1999.
15. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. Springer-Verlag, 1985.
16. K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium*, 1994.
17. J. Goguen. Higher-order functions considered unnecessary for higher order programming. In D. Turner, ed., *Proc. of University of Texas Year of Programming, Institute on Declarative Programming*. Addison-Wesley, 1988.
18. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, March 1992.
19. J. Guttag and J. Horning, eds. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
20. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. To appear in D.M. Gabbay, ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers. Short version in J. Meseguer, ed., *Proc. of Workshop on Rewriting Logic and its Applications*, vol. 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. Available at <http://www.elsevier.nl/locate/entcs/volume4.html>.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

22. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Object-Based Concurrency*, pp. 314–390. The MIT Press, 1993.
23. J. Meseguer. Membership algebra as a semantic framework for equational specification. In F. Parisi-Presicce, ed., *Recent Trends in Algebraic Development Techniques (WADT'97)*, vol. 1376 of *Lecture Notes in Computer Science*, pp. 18–61. Springer-Verlag, 1998.
24. Y. Srinivas and R. Jüllig. SPECWARE: Formal support for composing software. In B. Moeller, ed., *Proc. of Conference on Mathematics of Program Construction*, vol. 947 of *Lecture Notes in Computer Science*, pp. 399–422. Springer-Verlag, 1995.

A Reuse-Oriented Framework for Hierarchical Specifications

Sophie Coudert¹ and Pascale Le Gall^{2*}

¹ LSR, 681 rue de la passerelle, BP. 72 38702 Saint Martin d'Heres Cedex France

² L.a.M.I., Université d'Évry, Cours Monseigneur Roméro, 91025 Évry, France
Sophie.Coudert@imag.fr legall@lami.univ-evry.fr

Abstract. In order to facilitate the reuse of possibly complex hierarchical specification components, we propose a unified view of them as (generalised) open terms generated by constructors : the atomic modules (for example enrichments or presentations). Thus, all kinds of pieces of specifications are handled in a uniform way. Moreover, they are autonomous in the sense that they are well defined independently from the context of their design. We present an equational axiomatisation of the structure, providing the class of hierarchical specifications with two combination operators. We show on the example of proofs how thanks to this approach, an attribute for a specification may be systematically inherited from the ones of its modules. The so obtained attributes are naturally structured following the specifications.

Keywords: formal structured specification, reuse, proof system, structured inference, typed equational logic, monoid, language theory.

1 Introduction

Reuse is very promising for increasing both productivity and quality of software. Reused components may be of different nature as parts of code or specifications. Since the specification step precedes the other ones (detailed design, coding, proving, testing, ...) in software development, reusing specification components is particularly worth applying because it implies free reusing of these costly and dependent others activities. To facilitate it, two complementary directions should be studied. The first one is the direct design of reusable components. Such components must have good properties in terms of genericity, reliability, ... and are intended to be used as they are, without modifications. The second direction concerns the design of complex softwares from existing components. For that, some expertise is needed to adjust the components to the new requirements. But, above all, architecture description languages should be useful to facilitate the setting of component connection.

These considerations lead to a unified presentation of reusable components: a common format allows the specifier to put components in a library and to handle

* This work was partly supported by the ESPRIT-IV Working Group 22704 ASPIRE and by the ESPRIT-IV Working Group 23531 FIREworks

them uniformly through a dedicated language. For example, such a format may contain strict indications on the component interface form. More generally, it should attach to a component all the useful informations, and especially those issued from the context of its design. Indeed, to be really reusable, a component should be self contained, i.e. autonomous. Autonomy also means that the informations associated to the reused components allow to deduce all the pertinent informations about the complex system resulting from their connection. The underlying idea lies in the transmission of local informations to the global system through the architecture description language: properties are inherited from the ones of the reused components by combining them according to the connection scheme of the system. We present such an approach for hierarchical structuring which emphasizes the preservation of the modules behaviour when building complex systems, and thus facilitates autonomy purposes. Previous works [NOS95, BST99] about the so-called “modular specification” are all based on a similar structure, without cycles and mutual dependencies. It is typically the structure induced by algebraic primitives ([BG80, SW83, Wir94, Mos97, BST99]) but it can also be found in functional languages, where adding modules does not modify the behaviour of previous ones.

In this paper we rigorously characterise the common shape of all these hierarchical structures. For this, we specify the structure itself, giving a unified view for all kinds of hierarchical modules or primitives. The result is a both formal and flexible framework with a generic notion of component, which can be instantiated by the previous works about modularity. We can then envisage the joint reuse of different kinds of modules, by looking at them from a general point of view, while locally taking benefit of the specific properties offered by each specific approach. This work could be a base toward a reuse-oriented environment for structured specifications, managing libraries of components. The theoretical foundations we propose use well-known concepts (terms, monoids, equational specifications, ...) and give us an intuitive grasp of structured specifications: they are built from basic modules with two operators (a “vertical” composition and an “horizontal” concatenation). The result is very close to a simple intuitive representation of the specifications as walls, where the bricks are the atomic modules (the reusable components). We also show how the proposed framework allows us to provide structured specifications with structured semantics and proofs, with an emphasis on proofs.

2 Reuse Purpose and Hierarchical Specifications

2.1 Specification Formalism

A formal specification is generally described at least by a signature and a set of sentences, corresponding intuitively to the expected interface and properties about the system under specification. These formal elements are often issued from a logical formalism providing coherent semantic and inference counterparts. Such formalisms are often given as particular instances of some meta-formalism. There exist many variants (e.g. [CM97, Bor97]) which are all evolutions of the

one of institutions [GB92]. As we work towards reuse purpose, we encompass all these previous definitions of meta-formalisms by a very loose presentation of specification formalisms likely to accept any present or future exotic formalism.

Definition 1:

We consider as given a class $\mathcal{S}$ of signatures, a class $\mathcal{Sen}$ of sentences, a class $\mathcal{Mod}$ of models, and the four functions defined on¹ $\mathcal{S}$, $sen : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{Sen})$, $mod : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{Mod})$, $\models : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{Mod} \times \mathcal{Sen})$, $\vdash : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{Sen}) \times \mathcal{Sen})$, where for Σ in $\mathcal{S}$,

- $\models(\Sigma)$, denoted $\models_\Sigma$, is called the satisfaction relation and verifies $\forall \Sigma \in \mathcal{S}, \models(\Sigma) \in \mathcal{P}(mod(\Sigma) \times sen(\Sigma))$
- $\vdash(\Sigma)$, denoted by $\vdash_\Sigma$, is called the flat inference relation and verifies $\forall \Sigma \in \mathcal{S}, \vdash(\Sigma) \in \mathcal{P}(\mathcal{P}(sen(\Sigma)) \times sen(\Sigma))$
- The flat inference is sound, i.e.² $\forall \Sigma \in \mathcal{S}, \forall (\Phi, \Psi) \in sen(\Sigma),$

$$\Phi \vdash_\Sigma \Psi \Rightarrow (\forall M \in mod(\Sigma), M \models_\Sigma \Phi \Rightarrow M \vdash_\Sigma \Psi)$$

It is reflexive, monotonic and transitive, i.e. respectively,

$$\forall \Sigma \in \mathcal{S}, \forall \varphi \in sen(\Sigma), \forall \Gamma \subseteq sen(\Sigma), \forall \Gamma' \subseteq sen(\Sigma),$$

- $\{\varphi\} \vdash_\Sigma \varphi$
- $\Gamma \vdash_\Sigma \varphi$ and $\Gamma \subseteq \Gamma' \implies \Gamma' \vdash_\Sigma \varphi$
- $\Gamma \vdash_\Sigma \Gamma'$ and $\Gamma \cup \Gamma' \vdash_\Sigma \varphi \implies \Gamma \vdash_\Sigma \varphi$

Only the elements useful in order to express reuse features are incorporated within our definition. In particular, our reference set $\mathcal{S}$ of signatures may come from several existing dedicated formalisms (if it is defined as the disjoint union of all possible signatures) in order to build heterogeneous specifications [CBLG98]. In the sequel, these signatures will generically be denoted by $\Sigma, \Sigma_i, \Sigma', \dots$

2.2 Modules for Structured Algebraic Specifications

Specification structuring has become a crucial stake for dealing with large systems. It gives a support to decompose a problem into subproblems. Existing algebraic languages or formalisms brought out typical specification components. Structuring primitives or building operators as in [Wir93, HST94] offer a good coverage of algebraic structure. We record their existence, aiming to reuse them as soon as they fit in our reused-oriented work and we take our inspiration from their common shape to propose a unified presentation of reusable modules. They all share a hierarchical point of view: a specification is inductively defined as a module applied on existing subspecifications. We quickly recall some existing structuring primitives. The simplest modules are *basic* specifications defined by a signature and a set of axioms on it. An *enrichment module* $(\Delta\Sigma, \Delta Ax)$

¹ For technical reasons, we require that for any couple of distinct signatures Σ and Σ' of $\mathcal{S}$, the sets $sen(\Sigma)$ and $sen(\Sigma')$ are disjoint.

² We denote $M \models_\Sigma \psi$ for $(M, \phi) \in \models(\Sigma)$, $\Phi \vdash_\Sigma \Psi$ for $(\Phi, \Psi) \in \vdash(\Sigma)$ and if Φ and Ψ are sentence sets, $M \models \Phi$ stands for $\forall \phi \in \Phi, M \models \phi$ and $\Phi \vdash \Psi$ stands for $\forall \psi \in \Psi, \Phi \vdash \psi$. Moreover, the Σ index is omitted when there is no ambiguity.

adds the functionalities of $\Delta\Sigma$ and the associated properties ΔAx to an existing specification SP . The exported global signature is the union of $\Delta\Sigma$ and the one exported by SP . The *sum* $SP + SP'$ may be seen similarly but without any hierarchical order between SP and SP' . Two others classical modules are the ones used to hide or rename some functionalities of a specification.

A common point of these modules is that they know both the signature they export and the signatures of the specifications they are supposed to import. So we can see a module as an object provided with a list of imported signature and an exported signature. Intuitively, the coincidence of signatures (interfaces) is the minimal requirement to connect modules. So, our reuse-oriented approach will be based on modules modelled as functional constructors typed by signatures.

2.3 Structured Semantics and Inference Systems

With structuration comes modularity: the possibility of implementing modules separately, without caring about how imported or importing specifications are implemented. This view leads to give functions as semantics to modules, from imported models (on imported signatures) to models on the exported signature ([BST99]). For example, the semantics of an enrichment module are functions from models of the imported specification to models on the exported signature satisfying the axioms of the enrichment. Let us remark that some primitives do not satisfy this property, like sums whose semantics are the models satisfying the two imported specifications. Such a definition fits into a top-down characterisation while reuse mainly involves a bottom-up approach with available modules as starting point, and some new modules to adjust them with the new requirements. So, we will rather focus on bottom-up defined modules. Module reusing implies implementation reusing and one can deduce the semantics of a complex specification by composing module semantics in the sense of the usual composition of functions. One recognised difficulty is to ensure true sharing of implementations for any shared subspecification. We describe an implementation as a partial function from the modules to the functions of their semantics. Thus we associate an unique implementation to each module, whatever the number of its occurrences within a given specification: the module names directly serve us as non ambiguous identifiers and avoid us to use some explicit mechanism of naming involved for example in [BST99] through unit declarations.

Regarding inference systems, previous works have already pointed out the need for structured mechanisms to efficiently search for proofs, taking into account the sub-modules introducing the implicated elements. So, the structure of the specification naturally arises in structural inference rules. For example, enrichment modules can introduce the two following rules, for axiom introduction and preservation of imported properties (encapsulation rule in [BCLG96]):

$$\frac{}{(\Delta\Sigma, \Delta Ax) + SP \vdash ax} (ax \in \Delta Ax) \qquad \frac{SP \vdash \varphi}{(\Delta\Sigma, \Delta Ax) + SP \vdash \varphi}$$

In the sequel, we will systematically extend these inference schemes to all kind of complex pieces of specifications. Let us point out that by aiming at component

reusing, we advocate for pure structured inference systems, and not for reusing flat inference systems through some general results of specification equivalence between structured specifications and flat specifications [BHK90, Wir93]. We want to take benefit of any structural information and not to forget it.

2.4 Autonomy of Modules and Incomplete Specifications

A weakness of algebraic specification for reuse purpose is the lack of module autonomy: modules are too often dependent to their using context. We have already seen the case of the sum $SP + SP'$ where the semantics of SP are constrained by SP' ([Wir93]). The sharing of common subspecifications is ensured to the detriment of a true autonomy for SP . Another problem is that modules (like enrichments $(\Delta\Sigma, \Delta Ax)$ over SP) are generally designed to import a unique (list of) fully already known specification(s). The only noteworthy exceptions are the generic modules [DGS93] which specify the minimal properties expected from the imported specifications. In a certain sense, we will apply this to all our reusable modules. Indeed, we will provide them with a domain: a set of specifications defining the minimal constraints on allowed imported specifications. It may be seen as a weak version of genericity since it does not include instantiation mechanisms by means of signature renaming. There, we only require the soundness of inferences for models respecting the domains. So, we succeed in defining a local property of soundness which will be extended to all combinations of modules leading to a complete or incomplete specification (see [Cou98]).

3 Hierarchical Specifications as Terms

We first rigorously define a general shape for components and we then intuitively present the attributes associated to these specifications (semantics, proofs and application domains) showing how for autonomy purpose, each basic component has its own set of attributes from which we can deduce the attributes of complex specifications. The transmission mechanism related to the proof part is developed in section 5. (see [Cou98] a detailed presentation.)

3.1 The Common Shape of Hierarchical Modules

Here, we are specifically interested in the syntactic characteristics of specifications. As exposed in section 2, many primitives can be characterised by modules which import a signature sequence (those of the potentially imported specifications) and which export another signature. Under the name of *profile*, we define the adequate notion of types for these modules, generalised to multiple outputs and provided with some operators, in order to later extend them to complex specifications.

Reminder: For any class C , $(C^*, \bullet, \varepsilon)$ is the free monoid on C , i.e. the class of all finite sequences of elements of C , provided with the binary operator “ $\bullet$ ” (concatenation), with neutral element “ ε ”: the empty sequence. C^+ denotes $C^* \setminus \{\varepsilon\}$.

Definition 2:

We denote by $\mathcal{S}$ the class $(\mathcal{S}^*)^2$ of profiles on $\mathcal{S}$, provided with the total operator $\bullet : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ and the partial operator $\circ : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ defined for all (w_1, w_2) and (w_3, w_4) in $\mathcal{S}$ by

- $(w_1, w_2) \bullet (w_3, w_4) = (w_1 \bullet w_3, w_2 \bullet w_4)$
- $(w_1, w_2) \circ (w_3, w_4) = (w_1, w_4)$ if $w_2 = w_3$ and is undefined otherwise.

The elements of $\mathcal{S}^*$ are called semi-profiles.

With a profile in $\mathcal{S}$, atomic modules will have typed inputs and outputs, constraining their possible connections. They look like functional generators and complex specifications will look like terms. Thus, a hierarchical specification framework will be characterised by a class of *constructors*: the atomic modules on which specifications are built. We will call such a class a *metasignature* in order to avoid confusion with the specification signatures of $\mathcal{S}$.

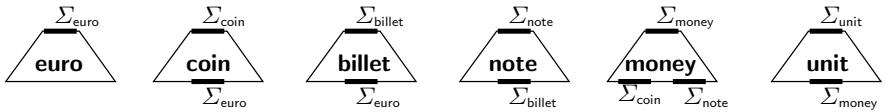
Definition 3:

A *metasignature* over $\mathcal{S}$ is a class Θ of constructors $\mathbf{f}$, each provided with a profile (ω, ω') over $\mathcal{S}$. Such a constructor $\mathbf{f}$ will often be denoted by $\mathbf{f} : \omega' \rightarrow \omega$, ω by $\text{In}(\mathbf{f})$, and ω' by $\text{Out}(\mathbf{f})$.

A *metasignature* is pointed if each element $\mathbf{f}$ verifies $\text{Out}(\mathbf{f}) \in \mathcal{S}$.

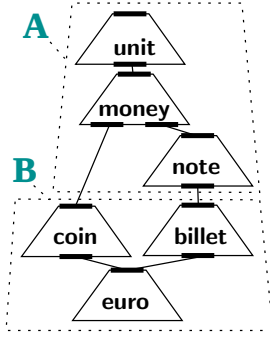
We can intuitively illustrate these definitions with the classical algebraic primitives. To specify European money³, we could have the following decomposition:

- A basic module $\mathbf{euro}^{\text{BAS}} : \Sigma_{\text{euro}} \rightarrow \Sigma_{\text{euro}}$ describing the euro and its subdivisions.
- Two enrichment modules $\mathbf{coin}^{\text{ENR}} : \Sigma_{\text{euro}} \rightarrow \Sigma_{\text{coin}}$ $\mathbf{billet}^{\text{ENR}} : \Sigma_{\text{euro}} \rightarrow \Sigma_{\text{billet}}$, describing coins and notes, the second one being written in French.
- renaming module $\mathbf{note}^{\text{REN}} : \Sigma_{\text{billet}} \rightarrow \Sigma_{\text{note}}$ based on an interpretation function from English names to French names.
- a sum module $\mathbf{money}^{\text{SUM}} : \Sigma_{\text{note}} \bullet \Sigma_{\text{coin}} \rightarrow \Sigma_{\text{money}}$, characterising the money as the union of coins and notes.
- A restriction module $\mathbf{unit}^{\text{RES}} : \Sigma_{\text{money}} \rightarrow \Sigma_{\text{unit}}$, hiding non integer values and based on the inclusion of Σ_{unit} in Σ_{money} .



If we see signature carriers as sets, we would have for example $\Sigma_{\text{unit}} \subseteq \Sigma_{\text{money}}$ or $\Sigma_{\text{money}} = \Sigma_{\text{coin}} \cup \Sigma_{\text{note}}$. Notice that we do not constrain the relation between imported and exported signatures. Its characterisation is delegated to the in-

³ For a more complete and rigorous presentation of an example, see [Cou98, CBLG98].



stances and thus to the particular primitives. Our generic description leads us to propose a primitive-independent description of module combination, allowing their joint reuse.

With this approach, we naturally see structured specifications as terms. So, the specification built on the previous modules can be seen on the figure below and described by $\text{unit}[\text{money}[\text{coin}[\text{euro}], \text{note}[\text{billet}[\text{euro}]]]]$. In this denotation two equal subterms implicitly refer to a shared subspecification. In the same way, any complete hierarchical specification can be denoted by

a term. However, it does not suffice for incomplete specification. For example, in order to denote the piece of specification A on the figure by a term like $\text{unit}[\text{money}[\text{?}, \text{note}[\text{?}]]]$, we need something well defined for the “?”. For this we introduce typed holes which can be compared to the anonymous variable of some logic programming languages. We also want to consider pieces like B on the figure, which can be seen as $[\text{coin}[\text{euro}], \text{billet}[\text{euro}]]$. For this, our terms also encompass tuples of terms so that we can deal with all kind of piece of specification in a uniform way. More technically, we will use the free monoid concatenation: its associativity spares us the management of parentheses. A Σ -typed hole will be denoted by $\tilde{\Sigma}$. And each term will naturally have a profile, describing the sequences of its inputs and outputs, from left to right.

Definition 4:

Let Θ be a pointed metasignature over $\mathcal{S}$. We define by induction the class $\mathcal{T}\Theta = \bigcup_{w \in \mathcal{S}^*} \mathcal{T}\Theta_w$ of the metaterms over Θ , each metaterm τ being

provided with a profile $\text{Pro}(\tau) = (\text{Out}(\tau), \text{In}(\tau))$ in $\mathcal{S}$:

1. $\forall \Sigma \in \mathcal{S}, \tilde{\Sigma} \in \mathcal{T}\Theta_{\Sigma}$, with $\text{Pro}(\tilde{\Sigma}) = (\Sigma, \Sigma)$
2. $\forall \mathbf{f} \in \Theta, \forall \tau \in \mathcal{T}\Theta_{\text{In}(\mathbf{f})}, \mathbf{f}[\tau] \in \mathcal{T}\Theta_{\text{Out}(\mathbf{f})}$,
with $\text{Pro}(\mathbf{f}[\tau]) = \text{Pro}(\mathbf{f}) \circ \text{Pro}(\tau)$.
3. $\varepsilon \in \mathcal{T}\Theta_{\varepsilon}$, with $\text{Pro}(\varepsilon) = (\varepsilon, \varepsilon)$
4. $\forall \omega, \omega' \in \mathcal{S}^*, \forall \tau \in \mathcal{T}\Theta_{\omega}, \forall \tau' \in \mathcal{T}\Theta_{\omega'}, \tau \bullet \tau' \in \mathcal{T}\Theta_{\omega \bullet \omega'}$,
with $\text{Pro}(\tau \bullet \tau') = \text{Pro}(\tau) \bullet \text{Pro}(\tau')$

We will denote such a metaterm by $\tau : \text{In}(\tau) \rightarrow \text{Out}(\tau)$. $\text{In}(\tau)$ is called the input profile of τ and $\text{Out}(\tau)$ its output profile. Moreover, we will denote recursively by $\tilde{\Sigma} \bullet u$ the metaterm $\tilde{\Sigma} \bullet \tilde{u}$, with $\tilde{\varepsilon} = \varepsilon$.

Notice that associating the metaterm $\mathbf{f}[\widetilde{\text{In}(\mathbf{f})}]$ to each constructor $\mathbf{f}$ defines a canonical injection from Θ to $\mathcal{T}\Theta$. Moreover, metaterms have a normal form: the class $\mathcal{T}\Theta$ verifies some foreseeable properties, as the fact that any non neutral metaterm may be uniquely decomposed as a sequence of pointed metaterms:

Properties 5:

Let Θ be a pointed metasignature over $\mathcal{S}$.

1. $\mathcal{T}\Theta$ is well defined: for each τ in $\mathcal{T}\Theta$, $\text{In}(\tau)$ and $\text{Out}(\tau)$ are unique.

2. $(\mathcal{T}\Theta, \bullet, \varepsilon)$ is a monoid isomorphic to the free monoid $(\widehat{\mathcal{T}}^*, \bullet, \varepsilon)$, with $\widehat{\mathcal{T}} = \bigcup_{\Sigma \in \mathcal{S}} \mathcal{T}\Theta_\Sigma$, the class of pointed terms.
3. For all u and v in $\mathcal{S}^*$ and for all τ in $\mathcal{T}\Theta_{u \bullet v}$, there is a unique decomposition $\tau = \tau_1 \bullet \tau_2$ such that $\text{Out}(\tau_1) = u$ and $\text{Out}(\tau_2) = v$.

Definition 4 allows us to “horizontally” combine terms using “ $\bullet$ ”. Property 5 allows us to inductively define a “vertical” composition “ $\diamond$ ”, typed by the signatures, extending the apposition of a constructor over metaterms. Our specification of the money can be then seen as $SP = A \diamond B$.

Definition 6:

Let Θ be a pointed metasignature over $\mathcal{S}$. The term composition is a partial binary operator $\diamond : \mathcal{T}\Theta \times \mathcal{T}\Theta \rightarrow \mathcal{T}\Theta$ defined for $\{(\tau_1, \tau_2) \in \mathcal{T}\Theta \times \mathcal{T}\Theta \mid \text{In}(\tau_1) = \text{Out}(\tau_2)\}$ by:

For each τ_1 and τ_2 in $\mathcal{T}\Theta$ such that $\text{In}(\tau_1) = \text{Out}(\tau_2) = \omega$,

1. if τ_1 is $\widetilde{\Sigma} : \Sigma \rightarrow \Sigma$, then $\tau_1 \diamond \tau_2 = \tau_2$
2. if τ_1 is $\mathbf{f}[\tau]$, then $\tau_1 \diamond \tau_2 = \mathbf{f}[\tau \diamond \tau_2]$
3. if τ_1 is ε , then τ_2 is ε and $\tau_1 \diamond \tau_2$ is ε
4. if τ_1 is $\tau_1^1 \bullet \tau_1^2$, then $\tau_1 \diamond \tau_2 = (\tau_1^1 \diamond \tau_2^1) \bullet (\tau_1^2 \diamond \tau_2^2)$, where (τ_2^1, τ_2^2) is the unique decomposition of τ_2 such that $\text{Out}(\tau_2^1) = \text{In}(\tau_1^1)$ and $\text{Out}(\tau_2^2) = \text{In}(\tau_1^2)$ and $\tau_2^1 \bullet \tau_2^2 = \tau_2$.

Thus, with this modelling of hierarchical structure, all kinds of specification term (closed or open, atomic or complex) can be simply combined with two operators and managed in a uniform way. It respects the intuition: the two operators describe the connections between modules and the order in which the modules are combined doesn’t matter, as ensured by the following properties:

Property 7:

Let Θ be a pointed metasignature over $\mathcal{S}$.

- The term composition on $\mathcal{T}\Theta$ is associative, i.e. if τ_1, τ_2 and τ_3 are three metaterms verifying $\text{In}(\tau_1) = \text{Out}(\tau_2)$ and $\text{In}(\tau_2) = \text{Out}(\tau_3)$, then $(\tau_1 \diamond \tau_2) \diamond \tau_3 = \tau_1 \diamond (\tau_2 \diamond \tau_3)$.
- The term composition and the concatenation on $\mathcal{T}\Theta$ commute, i.e. if τ_1, τ_2, τ'_1 , and τ'_2 are in $\mathcal{T}\Theta$ are such that $\text{In}(\tau_1) = \text{Out}(\tau'_1)$ and $\text{In}(\tau_2) = \text{Out}(\tau'_2)$, then $(\tau_1 \bullet \tau_2) \diamond (\tau'_1 \bullet \tau'_2) = (\tau_1 \diamond \tau'_1) \bullet (\tau_2 \diamond \tau'_2)$.

3.2 Autonomy for Components: The Rôle of Attributes

The previously presented specification terms describe structured texts. Here we present their attributes, i.e. semantics and proofs which are present in a lot of formalisms, and application domain which ensures autonomy for reuse purpose. Notice that not all the existing structuring primitives are reuse-oriented (see 2 for the case of the sum primitive). Thus some of them would need to be adapted for our oriented abstract unifying description. We consider five attributes:

- a *profile* : already presented inventory of the module inputs and outputs.
- *semantics*: mathematical models abstracting the possible implementations.
- an *inference*: proof mechanism to establish properties about the module.
- an *application domain*: the required properties about the imported specifications are defined by means of some canonical specifications.

Moreover the definition of an attribute for a module does not depend on imported or importing specifications and the attributes of a complex specification only depend on those of its components and on its structure.

We intuitively present attributes on the example of an enrichment module $\Delta : \Sigma \rightarrow \Sigma'$ (with $\Sigma \subset \Sigma'$), where Σ is the signature of the imported specification SP , and Σ' is the signature of the compound specification $\Delta[SP]$. Δ should accept several implementations, each of them producing an implementation of $\Delta[SP]$ from any implementation of SP . Thus, Δ 's semantics are functions from $mod(\Sigma)$ to $mod(\Sigma')$. The properties of $\Delta[SP]$ can originate in Δ or SP . Thus, Δ 's inference will be a binary relation between sentences on Σ (the premises) and sentences on Σ' (the consequences), with empty premises in the first case (Δ 's axioms introduction) and premises as consequences of SP in the second case (simple transmission). The flat inference $\vdash_{\Sigma'}$ can then be used to establish other properties about $\Delta[SP]$. Finally, a module can require some properties about what it imports. Previous approaches often presupposed a closed and known imported specification. Here the inputs are constrained by Δ 's domain, a class of specifications describing the minimal prerequisites. It is only a reference: any specification whose semantics respects those of the domain will be accepted and ensures a correct behaviour of Δ 's inference and semantics. Indeed, an inference is correct for a semantic function μ when the validity of premises in any imported model M implies the validity of the corresponding consequences in $\mu(M)$ and this constraint only holds for imported models issued from the domain.

Any attribute is automatically extended from components to compounds following a same mechanism: the attributes of two combined terms are combined with an operator corresponding to the operator used to combine the terms. Thus, for each attribute we have defined two operators which are briefly presented here using black points for concatenations and white points for compositions. To each constructor **f** in a metasignature Θ we associate:

- a profile $Pro(\mathbf{f})$ in $\mathcal{S}$. $\circ$ and $\bullet$ have already been defined (cf. def. 3).
- semantics $Sem(\mathbf{f}) \subseteq \{function\ f \mid f : mod(In(\mathbf{f})) \rightarrow mod(Out(\mathbf{f}))\}$, where $mod(\Sigma_1 \bullet \dots \bullet \Sigma_n)$ denotes⁴ $mod(\Sigma_1) \bullet \dots \bullet mod(\Sigma_n)$.
 “ $\bullet$ ” extends the following intuition: $(f \bullet g)(x \bullet y) = f(x) \bullet g(y)$
 while “ $\circ$ ” extends the classical composition of functions.
- an inference $\Vdash(\mathbf{f}) \subseteq sen(In(\mathbf{f})) \times sen(Out(\mathbf{f}))$, where $sen(\Sigma_1 \bullet \dots \bullet \Sigma_n)$ denotes $\mathcal{P}(sen(\Sigma_1)) \times \dots \times \mathcal{P}(sen(\Sigma_n))$.
 “ $\hat{\bullet}$ ” extends the following intuition: $(x \bullet x')(R \hat{\bullet} S)(y \bullet y') \Leftrightarrow (xRy) \wedge (x'Sy')$
 while “ $\hat{\circ}$ ” extends the classical composition of relations.

⁴ roughly speaking: the concatenation between classes is an associative Cartesian product.

- a domain $Dom(\mathbf{f}) \subseteq \mathcal{T}^{\Theta}_{In(\mathbf{f})} \times \{\mathbf{f}\}$, a class of terms exporting $In(\mathbf{f})$.
- “ $\bullet$ ” extends the concatenation to sets of terms
- while “ $\diamond$ ” is intuitively given by: $(D_1, \tau_1) \diamond (D_2, \tau_2) = (D, \tau_1 \diamond \tau_2)$, where the terms τ of D are those of D_2 which verify $\tau_2 \diamond \tau \in D_1$.

To complete this, notice also that we ensure a unique implementation choice for several occurrences of a constructor in a complex specification (see also section 2). The attributes of complex pieces of specification are built in the following way, which will naturally be modeled by morphisms in the more abstract presentation of section 4:

- $Pro(\tau \bullet \tau') = Pro(\tau) \bullet Pro(\tau')$ and $Pro(\tau \diamond \tau') = Pro(\tau) \circ Pro(\tau')$,
- $Sem(\tau \bullet \tau') = Sem(\tau) \bar{\bullet} Sem(\tau')$ and $Sem(\tau \diamond \tau') = Sem(\tau) \bar{\diamond} Sem(\tau')$,
- $\Vdash (\tau \bullet \tau') = \Vdash (\tau) \bar{\bullet} \Vdash (\tau')$ and $\Vdash (\tau \diamond \tau') = \Vdash (\tau) \bar{\diamond} \Vdash (\tau')$,
- $Dom(\tau \bullet \tau') = Dom(\tau) \bar{\bullet} Dom(\tau')$ and $Dom(\tau \diamond \tau') = Dom(\tau) \bar{\diamond} Dom(\tau')$.

4 Bimonoids: An Axiomatisation of Hierarchical Structure

We have seen how structured specifications can be seen as a class of metaterms provided with two combination operators satisfying some properties. Here, we give an axiomatisation in the typed equational logic (TEQL [EM85]) of such a structure, called *bimonoid* (term composition can be seen as a typed concatenation, with neutral holes):

Definition 8:

The bimonoids on $\mathcal{S}$ are the models (with respect to TEQL [EM85]) of the specification $SP_B(\mathcal{S}) = (\Sigma_B(\mathcal{S}), Ax_B(\mathcal{S}))$, where

- $\Sigma_B(\mathcal{S}) = (\mathcal{S}, F_B(\mathcal{S}))$ where $F_B(\mathcal{S})$ is the following set of function symbols, each provided with a typed (TEQL) arity:

1. $\epsilon : \rightarrow (\varepsilon, \varepsilon)$
2. $\omega : \rightarrow (\omega, \omega)$, for all ω in $\mathcal{S}^*$;
3. $\odot_{s,s'} : s \times s' \rightarrow (s \bullet s')$ for all (s, s') in $\mathcal{S} \times \mathcal{S}$;
4. $\odot_{s,s'} : s \times s' \rightarrow (s \circ s')$ for all (s, s') in $\mathcal{S} \times \mathcal{S}$ verifying $s|_2 = s'|_1$.

In the sequel, $\odot$ et $\odot$ will be used infix and indexes will be omitted.

Thus the axioms in the next point are in fact axiom schemas.

- $Ax_B(\mathcal{S})$ denotes the following set of axioms:

1. $(x \odot y) \odot z = x \odot (y \odot z)$
2. $\epsilon \odot x = x \odot \epsilon = x$
3. $\neg \omega_1 \odot \neg \omega_2 = \neg(\omega_1 \bullet \omega_2)$
4. $(x \odot x') \odot x'' = x \odot (x' \odot x'')$
5. $\neg \omega \odot x = x$
6. $x \odot \neg \omega' = x$
7. $(x \odot y) \odot (x' \odot y') = (x \odot x') \odot (y \odot y')$

where x, y, z, x', y' et x'' are adequately typed variables.

For example $\mathcal{S}$ may be provided with a structure of bimonoid on $\mathcal{S}$ by interpreting ϵ , ω , $\odot$ and $\odot$ respectively by $(\varepsilon, \varepsilon)$, (ω, ω) , $\bullet$ and $\circ$. The class $\mathcal{T}^{\Theta}$ of specification metaterms is a bimonoid too, with a particularity: it is built on the constructors of a signature Θ . In order to take benefit of this, we introduce Θ -parametrized bimonoids, where Θ is added to the bimonoid signature:

Definition 9:

Let Θ be a metasignature over $\mathcal{S}$. The Θ -bimonoids are the (TEQL) models of the specification $SP_B^\Theta(\mathcal{S}) = (\Sigma_B^\Theta(\mathcal{S}), Ax_B^\Theta(\mathcal{S}))$, where

- $\Sigma_B^\Theta(\mathcal{S}) = (\mathcal{S}, F_B(\mathcal{S}) \cup \{f : \rightarrow (v, u) \mid (f : u \rightarrow v) \in \Theta\})$
- $Ax_B^\Theta(\mathcal{S}) = \sigma(Ax_B(\mathcal{S}))$, with $\sigma : \Sigma_B(\mathcal{S}) \rightarrow \Sigma_B^\Theta(\mathcal{S})$ the signature inclusion morphism.

Morphisms between bimonoids are called morphisms of bimonoids.

The constructors $\mathbf{f} : \omega \rightarrow \omega'$ of Θ are seen here as constant symbols with the TEQL arity $\rightarrow (\omega', \omega)$. If Θ is a pointed metasignature, the class $\mathcal{T}\Theta$ may be seen as a Θ -bimonoid by interpreting each constant symbol $\mathbf{f} : \rightarrow (v, u)$ of $\Sigma_B(\mathcal{S})$ by the metaterm $\mathbf{f}[\tilde{u}]$, and $\epsilon, \neg u, \odot$ and $\otimes$, respectively by $\epsilon, \tilde{u}, \bullet$ and $\diamond$. Classical algebraic results ensure that a Θ -bimonoid B can be seen as a simple bimonoid by forgetting the additional operators of Θ and that the class of Θ -bimonoids is not empty. We thus get the advantage of the classical following property:

Property 10:

We denote the initial model of $SP_B^\Theta(\mathcal{S})$ (up to isomorphism) by $\Theta^\blacksquare$ and call it the free bimonoid generated by Θ . For each bimonoid B' and each total function h from Θ to B' preserving the profiles, there is a unique morphism of bimonoid $h^\blacksquare : \Theta^\blacksquare \rightarrow B'$, extending h , i.e such that $\forall x \in \Theta, h(x) = h^\blacksquare(x)$.

For a pointed metasignature Θ , $SP_B^\Theta(\mathcal{S})$ minimally describes $\mathcal{T}\Theta$ since $\mathcal{T}\Theta$ does not satisfy any extra property that would not be implied by $SP_B^\Theta(\mathcal{S})$:

Theorem 11:

Let Θ be a pointed metasignature over $\mathcal{S}$. The Θ -bimonoid $\mathcal{T}\Theta$ of the metaterms over Θ is isomorphic to the initial Θ -bimonoid $\Theta^\blacksquare$.

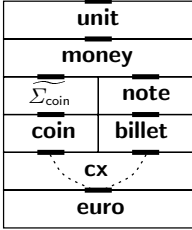
Proof: Not in this paper by lack of place, but can be found in [Cou98]

The free bimonoid characterises the class of the metaterms (i.e. structured specifications) for all metasignatures Θ . Thus, thanks to this algebraic approach, we can take benefit of Property 10 in the general case of hierarchical specification: any function $att : \Theta \rightarrow Att$ providing constructors with a given attribute can be systematically extended to metaterms by $att^\blacksquare$ defined on $\Theta^\blacksquare$, as soon as Att has a structure of bimonoid over Θ . Defining autonomous attributes amounts to ensure bimonoid structures to their classes. That's what we have briefly presented in section 3.2 about profiles, semantics, proofs and domains. About profile, let us remark that the function $\mathbf{Pro} : \mathcal{T}\Theta \rightarrow \mathcal{S}$ (def. 4) is in fact $(\mathbf{Pro} : \Theta \rightarrow \Theta)^\blacksquare$ ($\mathbf{Pro}$ in def. 3) and inferences are fully presented in section 5. So defined autonomous attributes are combined with respect to the specification structure without caring about the building order thanks to term equivalence classes induced by the specification $SP_B^\Theta(\mathcal{S})$. Finally, the attribute associated to the combination of two metaterms can be computed from those of the two pieces with the adequate operator on attributes, without caring about the internal structure of the two pieces.

4.1 Generalised Constructors

With bimonoids, we can also have not-pointed constructors . It allows to see a complex module as atomic, hiding its internal structure. It can be useful for local optimisations, with interdependent implementations of components. The complex module itself can be autonomous and used as basic component. More generally, it is then possible to structure specifications without requiring structured implementations: we allow one-module programs which just have to behave, from an external point of view, as a structured one would do.

Generalised constructors can also model new kind of components, such as connection modules which can be used to link inputs. The specification of the European Money could then be those on the inserted figure, where the module **cx** is such a new linking component. Pointed metaterms already modeled specification sharing: semantics ensure a unique implementation choice for all occurrences of a same constructor inside a specification. From a purely abstract point of view, it may suffice. But if we want to copy programming language where type compatibility often requires a unique compiled unit for a module



which is imported through two different ways, then we should be able to offer such a hard sharing. The shared models are not only identical, but also issued from exactly the same design decision. We offer a rigorous foundation for an intuitive language allowing the description of concrete architecture for functional systems, where specifications can be seen as walls (metaterms) of bricks (constructors): connection modules (as **cx**) allow to transmit properties related to a unique occurrence of a subspecification. Finally we can see on the figure that the neutral holes, allowing signature coincidence, behave like simple vertical transmission. At the semantics level, they are interpreted by signature-typed identities.

5 Structured Inferences for Hierarchical Specifications

The inference associated to a constructor is a relation between sentences on its input profile and sentences on its output profile . Intuitively, it allows the deduction of properties about the built global specification from properties about the imported ones. We first extend the notion of sentence on a signature. A sentence on a semi-profile is a sequence of sets of sentences respecting it:

$$\text{for all } \Sigma_1 \bullet \dots \bullet \Sigma_n \text{ in } \mathcal{S}^*, \quad \text{sen}(\Sigma_1 \bullet \dots \bullet \Sigma_n) = \{ \Phi_1 \bullet \dots \bullet \Phi_n \mid \Phi_1 \subseteq \text{sen}(\Sigma_1) \text{ and } \dots \text{ and } \Phi_n \subseteq \text{sen}(\Sigma_n) \}$$

And for our needs, we can also extend the notion of set union:

$$\text{for all } w \text{ in } \mathcal{S}^* \text{ and all } \Phi_1 \bullet \dots \bullet \Phi_n \text{ and } \Phi'_1 \bullet \dots \bullet \Phi'_n \text{ in } \text{sen}(w), \\ (\Phi_1 \bullet \dots \bullet \Phi_n) \cup (\Phi'_1 \bullet \dots \bullet \Phi'_n) = (\Phi_1 \cup \Phi'_1) \bullet \dots \bullet (\Phi_n \cup \Phi'_n)$$

Thus, a *hierarchical inference* for a constructor is a binary relation between sentences on its input profile and sentences on its output profile . So, hierarchical inferences are defined with respect to profiles:

Definitions 12:

- A hierarchical inference on a profile $w \rightarrow w'$ in $\mathcal{S}$ is a binary relation $\|\sim \subseteq \text{sen}(w) \times \text{sen}(w')$.
We denote **Infer** the class of all hierarchical inferences on all profiles in $\mathcal{S}$ and⁵ **Pro**($\|\sim$) the profile of $\|\sim$.
- A proof system for Θ is a function $\|\sim : \Theta \rightarrow \mathbf{Infer}$ associating to each constructor $\mathbf{f}$ a hierarchical inference on its profile verifying
for all φ in $\text{sen}(\text{In}(\mathbf{f}))$ and all ψ and ψ' in $\text{sen}(\text{Out}(\mathbf{f}))$,
if $(\varphi, \psi) \in \|\sim(\mathbf{f})$ and $(\varphi, \psi') \in \|\sim(\mathbf{f})$ then $(\varphi, \psi \cup \psi') \in \|\sim(\mathbf{f})$
As usual, $\|\sim(\mathbf{f})$ will be denoted by $\|\sim_{\mathbf{f}}$ and used in an infix way.

These inferences are structural ones: they allow the inheritance of properties through the structure of specifications. They can be extended to metaterms by using Property 10. For this we have to provide **Infer** with a structure of bimonoid by defining a concatenation and a composition between structural inferences:

Definition 13:

- The inference concatenation $\hat{\bullet} : \mathbf{Infer} \times \mathbf{Infer} \rightarrow \mathbf{Infer}$ is the binary function defined by: for all $\|\sim$ and $\|\sim'$ in **Infer**
 $\|\sim \hat{\bullet} \|\sim' = \{(\varphi \bullet \varphi', \psi \bullet \psi') \mid \varphi \|\sim \psi \text{ and } \varphi' \|\sim \psi'\}$
- The inference composition $\hat{\circ} : \mathbf{Infer} \times \mathbf{Infer} \rightarrow \mathbf{Infer}$ is the partial binary function defined by: for all $\|\sim$ and $\|\sim'$ in **Infer**,
 $(\text{In}(\|\sim) = \text{Out}(\|\sim')) \Leftrightarrow \|\sim \hat{\circ} \|\sim'$ is defined, and then,
 $\|\sim \hat{\circ} \|\sim' = \{(\Psi, \Phi) \in \text{sen}(\text{In}(\|\sim')) \times \text{sen}(\text{Out}(\|\sim)) \mid$
 $\exists \Lambda \in \text{sen}(\text{Out}(\|\sim')), \Psi \|\sim \Lambda \text{ and } \Lambda \|\sim' \Phi)\}$

We have then the expected property:

Property 14:

For all w in $\mathcal{S}^*$, let Id_w be $\text{Id}_w = \{(\Phi, \Phi) \mid \Phi \in \text{sen}(w)\}$.
Infer is provided with a structure of bimonoid on $\mathcal{S}$ by interpreting ϵ , $\neg_w$, $\odot$ and $\odot$ respectively by $\{(\epsilon, \epsilon)\}$, Id_w , $\hat{\bullet}$ and $\hat{\circ}$.

Thus, a proof system $\|\sim : \Theta \rightarrow \mathbf{Infer}$ is automatically extended to complex modules: $\|\sim^{\blacksquare} : \Theta^{\blacksquare} \rightarrow \mathbf{Infer}$ associates a hierarchical inference to each metaterm on Θ (Property 10). These structural inferences allow to raise properties inherited from modules through the structure, to the top level of the specification. But with such a system we don't take benefit of the flat inferences $\vdash$ associated to the signatures. We would like to allow a proof of the form " $\Phi \|\sim_{\mathbf{f} \circ_{\mathbf{g}}} \Psi$ " when there exist two sentences Λ and Λ' over the intermediate semi-profile $\text{Out}(\mathbf{g}) = \text{In}(\mathbf{f})$ such that $\Phi \|\sim_{\mathbf{g}} \Lambda$, $\Lambda \vdash \Lambda'$ and $\Lambda' \|\sim_{\mathbf{f}} \Psi$. Indeed, these flat inferences are sound with respect to the models exported by the corresponding components and then, such an extension would increase the proof power in a well-founded way. Flat inference is extended to semi-profiles:

⁵ the profile of an inference is unique because the sets of sentences over two different signatures are disjoint.

Notation: For all $\{\Sigma_i\}_{1 \leq i \leq n} \subseteq \mathcal{S}$, we can see $\vdash_{\Sigma_i}$ as a hierarchical inference $\vdash(\Sigma_i) : \Sigma_i \rightarrow \Sigma_i$, and then $\vdash(\Sigma_1 \bullet \dots \bullet \Sigma_n) = \vdash(\Sigma_1) \hat{\bullet} \dots \hat{\bullet} \vdash(\Sigma_n)$ is a hierarchical inference on $\Sigma_1 \bullet \dots \bullet \Sigma_n \rightarrow \Sigma_1 \bullet \dots \bullet \Sigma_n$

Properties 15:

Let $\text{clot} : \mathbf{Infer} \rightarrow \mathbf{Infer}$ and $\text{clot} : \mathbf{Infer} \rightarrow \text{clot}(\mathbf{Infer})$ be defined by

$$\forall \|\sim \in \mathbf{Infer}, \\ \text{clot}(\|\sim) = \|\sim_{\text{Out}(\|\sim)} \hat{\circ} \|\sim \hat{\circ} \|\sim_{\text{In}(\|\sim)} \quad \text{and} \quad \text{clot}(\|\sim) = \text{clot}(\|\sim).$$

- clot is a closure (i.e $\text{clot} \circ \text{clot} = \text{clot}$) and thus, for all w in $\mathcal{S}^*$, $\vdash_w$ is neutral for the inference composition in⁶ $\text{clot}(\mathbf{Infer})$.
- $\hat{\bullet}$ and $\hat{\circ}$ are stable in $\text{clot}(\mathbf{Infer})$.
- $\{(\varepsilon, \varepsilon)\}$ is in $\text{clot}(\mathbf{Infer})$.
- $\text{clot}(\mathbf{Infer})$ is provided with a bimonoid structure by respectively interpreting ε , $\neg_w$, $\odot$ and $\odot$ by $\{(\varepsilon, \varepsilon)\}$, $\{\vdash_w\}_{w \in \mathcal{S}^*}$, $\hat{\bullet}$ and $\hat{\circ}$.

Since $\text{clot}(\mathbf{Infer})$ is a bimonoid, we still use Property 10 and associate to our specifications proof mechanisms enriched by flat inferences:

Definition 16:

Let $\|\sim : \Theta \rightarrow \mathbf{Infer}$ be a proof system for Θ its canonical closed extension is $(\text{clot} \circ \|\sim)^\blacksquare$.

Notice that each metaterm τ is equal to $\widetilde{\text{Out}(\tau)} \diamond \tau \diamond \widetilde{\text{In}(\tau)}$ and that with the closure, it imports and exports sets of properties which can be saturated by the adequate flat inference systems. Thus here, the flat inferences are naturally hierarchical inferences associated to the holes, which are neutral for composition. It allows to enrich premises and consequences of structural inferences by deduction which are sound in the concerned models⁷. We can then envisage to reuse already done pieces of proof and to insert them in specification-like structured proofs, where the purely structural inferences have been added.

6 Conclusion

We have characterized pieces of hierarchical specifications as generalised open terms (encompassing tuples of terms) generated by constructors which are the atomic modules. This approach has good properties for reuse purpose: we can uniformly handle enrichment module for example (as “one function symbol” terms), and incomplete specifications; moreover, modules are autonomous: their definition is “context independent”. For this, we have specified the structure (called *bimonoid*) with the typed equational logic in such a way that the initial model of the specification exactly describes the class of hierarchical specifications. As a result, associating an attribute (semantics, proofs, domain) to each

⁶ notice that $\text{Id}_w (= \{(\Phi, \Phi) \mid \Phi \in \text{sen}(w)\})$ is not in $\text{clot}(\mathbf{Infer})$ (except for $w = \varepsilon$).

⁷ A detailed example of a heterogeneous structured proof is given in [Cou98, CBLG98].

constructor suffices to automatically provide all complex specifications with it, and these attributes are correspondingly structured. This mechanism has been rigorously developed about inference systems which reuse the flat inferences associated to signatures by formalisms. It has been intuitively presented about other attributes (see [Cou98] for a detailed presentation).

This work gives foundation for a simple language with two combination operators for specifications: a vertical composition and a horizontal concatenation. Intuitively, hierarchical specifications can be seen as walls, whose bricks are the atomic modules. And finally, as signatures or modules can belong to different formalisms, our specifications are potentially heterogeneous. This idea is developed in ([BCLG96, CBLG98, Cou98]), with the introduction of specific translation modules.

References

- [BCLG96] G. Bernot, S. Coudert, and P. Le Gall. Towards heterogeneous formal specifications. In *AMAST'96, Munich*, pages 458–472. LNCS 1101, 1996.
- [BG80] R. Burstall and J. Goguen. The semantics of CLEAR, a specification language. In *Proc. Advanced Course on Abstract Software Specifications, Berlin*, pages 292–332. Springer, 1980.
- [BHK90] J.A. Bergstra, R. Heering, and P. Klint. Module algebra. *Journal of the Association for Computer Machinery*, 37(2):335–372, 1990.
- [Bor97] T. Borzyszkowski. Correctness of a logical system for structured specifications. In *WADT'97, Tarquinia, Italy*, pages 107–121. LNCS 1376, 1997.
- [BST99] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specification in CASL. In *AMAST'98, Amazonia-Manaus*, pages 263–277, LNCS 1548, 1999.
- [CBLG98] S. Coudert, G. Bernot, and P. Le Gall. Hierarchical heterogeneous specifications. In *WADT'98, Lisbon, Portugal*. LNCS 1589, p. 107–121, 1998.
- [CM97] M. Cerioli and J. Meseguer. May I borrow your logic? Transporting Logical Structures along Maps. *TCS*, 173(2):311–347, 1997.
- [Cou98] S. Coudert. Cadre de spécifications hétérogènes. PhDThesis, Université d'Évry, France, 1998.
- [DGS93] R. Diaconescu, J. Goguen, and P. Stefaneas. Logical support for modularisation. In G. Huet and G. Plotkin, editors, *Proc. Workshop on Types and Logical Frameworks*, p. 83–130, 1993.
- [EM85] H. Ehrig and B. Mahr. Fundamentals of algebraic specification 1. equations and initial semantics. *EATCS Monographs on Theoretical Computer Science*, 6, 1985.
- [GB92] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of ACM*, 39(1):95–146, 1992.
- [HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured theory presentations and logic representations. *Annals of Pure and Applied Logic*, 67, 1994.
- [Mos97] P. Mosses. CoFI: The common framework initiative for algebraic specification and development. In Springer, editor, *Proc. TAPSOFT'97, Theory and Practice of Software Development*, volume 1214, pages 115–137, 1997.
- [NOS95] M. Navarro, F. Orejas, and A. Sanchez. On the correctness of modular systems. *Theoretical Computer Science*, 140:139–177, 1995.

- [SW83] D. Sannella and M. Wirsing. A kernel language for algebraic specifications and implementation. In *Proc. FCT'83*. LNCS 158, p. 413-427, 1983.
- [Wir93] M. Wirsing. Structured specifications: syntax, semantics and proof calculus. In Brauer W. Bauer F. and Schwichtenberg H., editors, *Logic and Algebra of Specification*, pages 411–442. Springer, 1993.
- [Wir94] M. Wirsing. Algebraic specification languages: An overview. In *WADT'94, S. Margherita, Italy*, volume 906, pages 81–116. LNCS, 1994.

MIX(FL): A Kernel Language of Mixin Modules^{*}

Davide Ancona

Dipartimento di Informatica e Scienze dell'Informazione
Via Dodecaneso, 35 - 16146 Genova (Italy)
davide@disi.unige.it

Abstract. We define the language of mixin modules *MIX(FL)* with the aim of providing foundations for the design of module systems supporting mixins. Several working examples are presented showing the benefits of the use of mixins and overriding in module systems. The language is strongly typed and supports separate compilation. The denotational semantics of the language is based on an algebraic approach and is parametric in the semantics of the underlying core language. Hence, even though the language is defined on top of a specific core language, other kinds of core languages could be considered as well.

Introduction

The notion of mixin originated from the object-oriented community (see [17,15]) for denoting *parametric heir classes*, that is, classes which can be applied to different parent classes. Recently, this notion has been recognized to be independent of the object-oriented paradigm; indeed, it turns out to be a useful feature for promoting software reuse in module languages (see [7,10,13,3,4]).

We say that a module language supports *mixin modules* (or simply *mixins*) if it provides the following features:

- modules can be *parametric*, that is, their behavior depends on components, usually called *deferred*, whose definition is not known in the module but must be provided from the outside;
- the basic mechanism for making modules *concrete* (i.e., with no deferred components) is a binary operator between modules M_1 and M_2 that allows the user to associate to a deferred component of M_1 (resp. M_2) the definition of some component defined in M_2 (resp. M_1), in such a way that *mutual recursion* between module boundaries is supported;
- there exist operations allowing the user to change the definition of a component f in a module M in such a way that all other components of M whose behavior depends on f change as well, according to the new definition of f ; we say that the definition of f has been *overridden* and f is called a *virtual component*.

^{*} Partially supported by Murst - Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software and APPLIED SEMantics - Esprit Working Group 26142.

More briefly, we can say that a language of mixins supports parametric modules, mutual recursion between modules and virtual components.

According to the definition given above, in *MIX(FL)* a mixin is a module which contains deferred components (functions and types), i.e. components to be imported from another module; a binary and commutative operator of merge allows to associate deferred components of one mixin with the corresponding definitions (if any) of the other. In this way, recursive definitions are allowed to span module boundaries. Furthermore, *MIX(FL)* supports virtual components. From the linguistic point of view, *MIX(FL)* is a very simple typed module language, supporting separate compilation and providing five operators for mixins manipulation (merge, freeze, restrict, hiding and renaming).

The design of the language *MIX(FL)* is based on two principles.

First, we want to concentrate on the notion of mixin module by defining a simple (but expressive) language for mixins having an algebraic model. The simplicity of the language allows to define in a rather compact way its static correctness and its denotational semantics.

Second, we want to show that the language can be defined in a modular way. Indeed the language is made up of two independent layers: the core language, which has no modularity features and the module language, which is parametric in the core language. The module language can be “applied to” a variety of core languages satisfying a minimal set of requirements. This is suggested by the notation *MIX(FL)*, where the core language *FL* is a simple typed functional language which could be replaced by other core languages. For sake of simplicity, we have chosen *FL* to be a simple functional language, but it should be clear that the language could be replaced by other more complex languages.

The sequel of the section is organized as follows: in Sect.1 the language *MIX(FL)* is introduced by means of some working examples. Sect.2 is devoted to an overview of the algebraic model used for defining the semantics of *MIX(FL)*. In Sect.3 the syntax and the typing rules of the language are provided, whereas the denotational semantics has been omitted for lack of space (see [1]). Finally, in Sect.4 we draw some conclusion.

1 Defining Mixins in *MIX(FL)*

In this section we introduce the language *MIX(FL)* by means of some working examples. All main features of the module language and of the core language *FL* are illustrated.

1.1 Mixins

Consider the following example of definition of mixin in *MIX(FL)*:

```
mixin GenericIntTree =
mix
  deferred
    type forest
```

```

fn emptyForest():forest
  addForest(tree,forest):forest
  eqForest(forest,forest):bool
defined
  type tree = int×forest
fn singleNode(i:int):tree = (i,emptyForest())
  addChild(c,t:tree):tree = (t.fst,addForest(c,t.scd))
  eqTree(t1,t2:tree):bool = t1.fst=t2.fst and eqForest(t1.scd,t2.scd)
end

```

The mixin **GenericIntTree** defines an implementation of (the data-type of) integer-labeled trees, where each tree is represented by a pair consisting of the integer labeling its root and the forest of its subtrees. The mixin is not concrete, since the definitions for the data-type of forests are deferred. The usefulness of this decision is two-fold; first, there is not a unique “right” implementation for it: lists, sets and multi-sets are three reasonable possibilities (among others) and the choice depends on which kind of tree equality we want to define; second, even in the case we have in mind a “standard” data-type, say lists, we are not forced to implement it, but we can simply use a compiled mixin module for generic lists provided by the library in order to instantiate **GenericIntTree**.

Note that we could further increase the level of genericity of the module by “abstracting” also on the type of the labels of the trees and on the corresponding equality test, but here we have avoided this for sake of simplicity.

Let us now analyze the structure of a mixin definition in *MIX(FL)*; a mixin consists of a *deferred* and a *defined* part, introduced by the keywords **deferred** and **defined**, respectively. The only entities *MIX(FL)* deals with are *types* and *functions*.

In the deferred part the types and functions that must be provided from the outside are listed. We call them *deferred types* and *deferred functions*, respectively. Note that deferred components are also types. That allows to effectively exploit the expressive power of mixin modules even in the case of strongly typed languages. However, we will see that, when dealing with types, one has to be more careful in defining the semantics of the mixin operators.

Deferred types are simply listed and for each deferred function the corresponding signature is specified. Note that the signature of a deferred function may also contain a type defined in the mixin, as it happens for **addForest** in **GenericIntTree**.

The types and functions contained in the definition part are called *defined types* and *defined functions*, respectively. All the definitions contained in the definition part are exported, therefore are visible from the outside.

Type definitions may be mutually recursive and may refer also to deferred and predefined types. For sake of simplicity we consider only three predefined types: the type **unit**, containing a unique value denoted by the constant *****, and the types **bool** and **int** of booleans and integers, respectively, with the usual constants and operations.

Type constructors provided by the language are *disjoint sum* (+) and *product* (×). A set of predefined primitives is associated with each type constructor. The

type definition $s = t \times u$ implicitly introduces the primitives $(_,\_):t \times u \rightarrow s$, $\_fst:s \rightarrow t$ and $\_scd:s \rightarrow u$, whereas the type definition $s = t+u$ implicitly introduces the primitives $\_fst?:s \rightarrow \text{bool}$, $\_scd?:s \rightarrow \text{bool}$, $<\_,\_>:t \rightarrow s$, $<-,\_>:u \rightarrow s$, $\_fst:s \rightarrow t$ and $\_scd:s \rightarrow u$, with the usual meaning; we refer to [1] for more explanations on these primitives.

Function definitions may be mutually recursive and may refer also to deferred functions, the primitives introduced by the type definitions and the predefined operations.

The meaning of `singleNode(i)`, `addChild(c,t)` and `eqTree(t1,t2)` is the following: the first expression returns a tree having only one node labeled by `i`, the second adds the tree `c` to the forest of subtrees of `t`, and the third performs the equality test between `t1` and `t2` and depends on the function `eqForest` which is expected to perform in turn the equality test on forests.

Finally, note that `GenericIntTree` is not simply a parametric module, since the interface which it depends on (corresponding with its deferred part) contains the type `tree` defined in the mixin. For instance, in SML [16] the signature (i.e., the interface) of a formal parameter of a functor (i.e., a parametric module) can not contain a type defined in the body of the functor.

1.2 Merging Mixins

In order to effectively use the mixin `GenericIntTree`, we must provide a definition for the type `forest` and for the functions `emptyForest`, `addForest` and `eqForest`. Assume we want to implement forests as lists of trees and let us consider the following mixin containing a standard implementation of generic lists with equality test.

```
mixin GenericList =
mix
  deferred
    type elem
    fn eqElem(elem,elem):bool
  defined
    type list = unit+(elem×list)
    fn emptyList():list = <*,->
      addList(e:elem,l:list):list = <-, (e,l)>
      eqList(l1,l2:list):bool =
        if l1.fst? then l2.fst?
        else if l2.fst? then false
        else eqElem(l1.scd.fst,l2.scd.fst) and eqList(l1.scd.scd,l2.scd.scd)
end
```

The mixin `GenericList` defines the data-type of lists over elements of a generic type `elem` with equality `eqElem`. Note that in this case `GenericList` actually corresponds to a parametric module. The type `list` is defined by means of the $\times$ and $+$ type constructors.

Now we would like to merge `GenericIntTree` with `GenericList` in such a way that the following bindings are performed: the type `forest` and the functions

`emptyForest`, `addForest` and `eqForest` deferred in `GenericIntTree` are associated, respectively, with the type `list` and the functions `emptyList`, `addList` and `eqList` defined in `GenericList`; conversely, the type `elem` and the function `eqElem` deferred in `GenericList` are associated, respectively, with the type `tree` and the function `eqTree` defined in `GenericIntTree`. We can merge the two mixins together and accomplish the desired bindings by means of the renaming operator.

```
GenericIntTree ⊕
(rename type elem by tree, list by forest
  fn eqElem by eqTree, emptyList by emptyForest,
    addList by addForest, eqList by eqForest
in GenericList)
```

Note that the mixin obtained in this way contains in the defined part also the functions `emptyForest`, `addForest` and `eqForest`, which now can be hidden from the outside¹. To this end, we can define

```
mixin IntTree =
  hide emptyForest, addForest, eqForest in
  (GenericIntTree ⊕
    (rename type elem by tree, list by forest
      fn eqElem by eqTree, emptyList by emptyForest,
        addList by addForest, eqList by eqForest
in GenericList))
```

The mixin `IntTree` contains a concrete implementation of integer-labeled trees (hence, we call it a *concrete* mixin) and exports the type `tree` and `forest` and the functions `singleNode`, `addChild` and `eqTree`.

1.3 Overriding Definitions

The language provides operators for redefining components and supports the notions of virtual and frozen component. As an example, consider the following mixin definition:

```
mixin DefaultIntTree =
mix
defined
  type tree = int × forest
    forest = unit + (tree × forest)
  fn virt emptyForest(): forest = <*, ->
  virt addForest(t: tree, l: forest): forest = <-, (t, l)>
  virt eqForest(l1, l2: forest): bool =
    if l1.fst? then l2.fst?
```

¹ In fact, in this case we could also hide the type `forest`, but for sake of simplicity, the language allows to hide only functions.

```

    else if l2.fst? then false
    else eqTree(l1.scd.fst,l2.scd.fst) and
        eqForest(l1.scd.scd,l2.scd.scd)
singleNode(i:int):tree = (i,emptyForest())
addChild(c,t:tree):tree = (t.fst,addForest(c,t.scd))
eqTree(t1,t2:tree):bool = t1.fst=t2.fst and eqForest(t1.scd,t2.scd)
end

```

The mixin `DefaultIntTree` is concrete (indeed, it has no deferred part), and, at first sight, it may seem equivalent to `IntTree`; however, the functions `emptyForest`, `addForest` and `eqForest` are declared virtual and, therefore, can be redefined in order to obtain a different implementation of the data-type of forests². All the other components which are not explicitly declared virtual are considered frozen.

For instance, assume that we need a different implementation of integer-labeled trees, where the equality on trees does not take into account the order and the possible repetitions of subtrees.

```

mixin ListModifier =
  hide fn included,isIn in
  mixin
    deferred
    type list,elem
    fn head(list):elem
    tail(list):list
    isEmptyList(list):bool
    eqElem(elem elem):bool
  defined
  fn eqList(l1,l2:list):bool = included(l1,l2) and included(l2,l1)
  included(l1,l2:list):bool =
    if isEmptyList(l1) then true
    else isIn(head(l1),l2) and included(tail(l1),l2)
  isIn(e:elem l:list):bool =
    if isEmptyList(l) then false
    else eqElem(e,head(l)) or isIn(e,tail(l))
end

```

The mixin `ListModifier` is a modifier for mixins implementing lists. Since we do not want to deal with type modifications, the type `list` must be provided from the outside; hence, the only way to manipulate elements of type `list` inside `ListModifier` is by means of primitives that must be provided from the outside, too.

The redefinition of `eqForest` can be performed in the following way: first, `DefaultIntTree` is restricted in order to eliminate the definition of `eqForest` and then it is merged with the new definition provided by `ListModifier`.

² Note that, however, the definition of the type `forest` can not be changed, as explained in the sequel.

```
(restrict eqForest in DefaultIntTree)  $\oplus$ 
(rename type elem by tree, list by forest
  fn eqElem by eqTree, eqList by eqForest
in ListModifier)
```

Note that for this last example to work, the mixin `DefaultIntTree` should contain also the definitions of the primitives corresponding to `head`, `tail` and `isEmptyList`.

Finally, $MIX(FL)$ provides a freeze operation which allows to transform virtual into frozen functions. For example, the expression

```
freeze emptyForest, addForest, eqForest in DefaultIntTree
```

denotes a mixin equivalent to `DefaultIntTree` except that `emptyForest`, `addForest` and `eqForest` are now frozen functions. See [4] for more examples and explanations on the freeze operator.

2 Overview of the Algebraic Model

In this section we define the framework that we will use for defining the static and dynamic semantics of $MIX(FL)$.

For reasons of space we only give an overview of the model; however, the reader can find the main definitions and more explanations and technical details in [1].

The algebraic model of $MIX(FL)$ must satisfy two main properties: it must be parametric in the model of the core language, so that the semantics of mixins is independent of the language FL and the module language can be easily instantiated over other core languages; it must be able to model parametric modules, virtual components and mutual recursion.

The first point can be achieved by using the notion of model part which is directly derived from that of institution [14]. This amounts to say that the algebraic model at the core level can be represented essentially by a pair $\langle \mathbf{Sig}, Mod \rangle$ where $\mathbf{Sig}$ is a category of signatures (which is required to have all finite colimits, in order to be able to combine signatures) and $Mod: \mathbf{Sig}^{op} \rightarrow \mathbf{Class}$ is a contravariant functor (which is required to preserve all finite colimits in order to be able to combine models).

The second point requires that mixin models are functions from $Mod(\Sigma^{in})$ into $Mod(\Sigma^{out})$, where Σ^{in} is the signature of the components from which the mixin depends on (that is, its deferred and virtual components) and Σ^{out} is the signature of the components defined in the mixin (that is, its virtual and frozen component). In other words, a mixin signature can be formalized by a pair of signatures $\langle \Sigma^{in}, \Sigma^{out} \rangle$ where $\Sigma^{in} = \Sigma^{def} + \Sigma^{vir}$, $\Sigma^{out} = \Sigma^{vir} + \Sigma^{fro}$, with Σ^{def} , Σ^{vir} , Σ^{fro} signatures corresponding to the deferred, frozen and virtual components, respectively. That expresses in an abstract way that the only relation between the input and the output signatures is given by the virtual

components, which are shared; in particular, if there are no virtual components, then Σ^{in} and Σ^{out} can be considered as two independent signatures.

Basically, following the idea firstly developed in [8] and [18] for the denotational semantics of classes and inheritance, mixin have two different semantics: an *open* semantics (that is, mixins as functions) suitable for combining modules and a *closed* semantics (that is, mixins as models over a given signature) needed for dealing with modules at the core level. Note that the closed semantics is expected to be obtained from the open semantics via application of a fixed point operator.

Another technical “ingredient” required by the second point is the existence (of a family) of fixed point operators to be applied to mixin models, since even a concrete mixin is represented by a function, if it has some virtual component, and, as already discussed, we both need its open and closed semantics; furthermore, fixed point operators are at the basis of the freeze operator presented in Sect.1.

Let us now come back to the mixin **GenericIntTree** defined in Sect.1. We have already noted that the deferred operation **addForest** contains the defined type **tree**, hence the interface of **GenericIntTree** cannot be divided in two independent signatures corresponding to deferred and defined components, respectively. Since in general we allow signatures of deferred components to contain defined types, and conversely, it is clear that the model outlined above is too abstract for modeling mixins in *MIX(FL)*: intuitively, we would like to express the property that the input and the output signatures share types. Hence, we have to consider signatures in which there is an explicit notion of sort modeling type components. A mixin signature can be defined as a 4-tuple $\langle S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out} \rangle$ where S^{in} and S^{out} are two disjoint sets of symbols corresponding to deferred and defined type components, respectively, and $\Sigma^{in}, \Sigma^{out}$ are two signatures over $S^{in} \cup S^{out}$ which have the same meaning as explained above.

The assumption that S^{in} and S^{out} are disjoint sets models the fact that type components cannot be virtual. Indeed, in Sect.1.3 we have already noted that the type **forest** cannot be safely redefined in **DefaultIntTree**. Hence, type components in signatures must be explicitly distinguished from other components also for a second reason, that is that they are components which cannot be redefined since it is not true that other definitions make sense for (formally, are parameterized by) any possible definition of them.

Correspondingly, at the semantic level, we need to refine our notion of a mixin model as a pair $\langle H, F \rangle$. The component H is a function giving the semantic counterpart of type definitions in a module, in terms of deferred types. For instance, in the mixin **GenericIntTree**, this function gives an interpretation of the type **tree** for each given interpretation of the type **forest**.

We assume that the denotation of a type component is the set of its possible values. Hence, H is more precisely a function

$$H: SSet(S^{in}) \rightarrow SSet(S^{out}).$$

Moreover, H must be continuous (in the usual sense, see [1]); indeed, referring to the mixins **GenericIntTree** and **GenericList** of Sect.1, when we merge them,

we get a mixin with no deferred type components ($S^{in} = \emptyset$) and two defined type components whose interpretation is the least fixed point of the following recursive definition:

```
tree = int×forest
forest = unit+(elem×forest)
```

The component F is a family of (total) functions indexed over possible interpretations Y of deferred types (formally, $Y \in SSet(S^{in})$), s.t., for any Y , F_Y gives the semantic counterpart of other (non-type) definitions in a module, in terms of input components, for a fixed interpretation of deferred (Y) and defined ($H(Y)$) types.

Formally, denoting by $Mod_X(\Sigma)$ the class of the Σ -models with carrier X (i.e., where the interpretation of sorts is fixed to be as in X),

$$F_Y: Mod_{Y+H(Y)}(\Sigma^{in}) \rightarrow Mod_{Y+H(Y)}(\Sigma^{out})$$

For instance, in the mixin **GenericIntTree**, any Y gives an interpretation (set of values) for the type **forest**, and, correspondingly, F_Y gives an interpretation of the functions **singleNode**, **addChild** and **eqTree** for each given interpretation of the functions **emptyForest**, **addForest** and **eqForest**; the domains and codomains of all these functions are fixed for each index Y .

From the discussion above it is clear that the algebraic framework that must be provided at the core level (what we call the *core framework*) is richer than a model part. We refer to [1] for a formal definition of core framework.

3 Syntax and Typing Rules

In this section we introduce the formal syntax of the language $MIX(FL)$.

First, the abstract syntax of the untyped version of the core and the module language is presented; then, the typed version is introduced by means of a set of typing rules.

As already stated, the semantics of the language is driven by the formal framework briefly sketched in Sect.2. This is reflected by the terminology used in the sequel: types at the level of the core language are also called sorts.

3.1 The Untyped Languages

We define the untyped version of the core and the module language. The abstract syntax of the untyped core language is given in Fig.1.

The meta-variables t, t_i range over the set of user-defined sort symbols, while s, s_i range over its superset of all sort symbols containing also the set $S_P = \{unit, bool, int\}$ of predefined sorts. The meta-variables f, f_i range over the set of user-defined operation symbols, while g ranges over its superset of all operation symbols containing also the predefined operations of the language. The set of predefined operations contains all the operations associated with the predefined

types and the primitives implicitly introduced by type constructors. The meta-variables x, x_i range over the set of variables.

We also use the standard notation $\overline{me}$ or $me_i^{i=1..n}$ for tuples $me_1, \dots, me_n$.

Finally, we have the following additional assumptions on the syntax of the core language. Declarations and definitions of sort and operation symbols do not contain repetitions; this implies that overloading is not allowed. The sets of user-defined sort symbols, predefined sort symbols, user-defined operation symbols, predefined operation symbols and variable symbols are pairwise disjoint.

Type declarations (*tdec*) are sequences of user-defined sort symbols. Function declarations (*fdec*) are sequences of assignments of function signatures to user-defined operation symbols. Type definitions (*tdef*) are sequences of assignments of type expressions to sort symbols. Type expressions are formed by means of sorts and the two type constructors $\times$ (product) and $+$ (disjoint sum). For sake of simplicity we consider only elementary (not structured) type expressions; this is not a limitation since more complex type expressions may be defined by introducing auxiliary sort symbols. Function definitions (*fdef*) are sequences of assignments of function signatures and expressions to user-defined operation symbols. Finally, an expression is either a variable or a function application or a conditional expression.

$$\begin{array}{ll}
 tdec ::= t_i^{i=1..n} & fdef ::= f_i(\bar{x}_i:\bar{s}_i):s_i = e_i^{i=1..n} \\
 fdec ::= f_i(\bar{s}_i):s_i^{i=1..n} & e ::= x \mid g(e_i^{i=1..n}) \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
 tdef ::= t_i = s_i op_i s_i'^{i=1..n} & op ::= \times \mid +
 \end{array}$$

Fig. 1. Abstract syntax of the untyped core language

The abstract syntax of the untyped module language is given in Fig.2. The freeze and the restrict operations have effect only on functions, while hiding and renaming are significant also for types. We assume that in the sequences of types and functions there are no repetitions. In particular, for the case of renaming we assume that $\{t_i^{i=1..n}\} \cap \{t'_i{}^{i=1..n}\} = \emptyset$ and $\{f_i^{i=1..p}\} \cap \{f'_j{}^{j=1..p}\} = \emptyset$ and for the case of mixin definitions (last production) the set of sort symbols defined in *tdef* is disjoint from the set of sort symbol defined in *tdec*. Analogously, *fdef_f*, *fdef_v* and *fdec* define different function symbols; *fdef_f* and *fdef_v* represent the definitions of the frozen and virtual function, respectively.

$$\begin{array}{l}
 M ::= M_1 \oplus M_2 \mid \text{freeze } f_i^{i=1..n} \text{ in } M \mid \text{restrict } f_i^{i=1..n} \text{ in } M \mid \text{hide } f_i^{i=1..p} \text{ in } M \\
 \mid \text{rename type } t_i \text{ by } t'_i{}^{i=1..n} \text{ fn } f_j \text{ by } f'_j{}^{j=1..p} \\
 \mid \text{mix deferred type } tdec \text{ fn } fdec \text{ defined type } tdef \text{ fn } fdef_f \text{ virt } fdef_v \text{ end}
 \end{array}$$

Fig. 2. Abstract syntax of the module language

3.2 The Typed Languages

We give the typing rules for both the core and the module language. As already stated, we model mixin interfaces by mixin signatures instantiated over *Palg*.

Before giving the typing rules, we introduce some auxiliary functions used in the typing rules (and also in the semantic clauses).

- Let Σ be a signature; then $Opns(\Sigma)$ denotes the set of the operation symbols of Σ . For any pair of signatures Σ_1 and Σ_2 , we say that Σ_1 and Σ_2 are *weakly disjoint* and write $\Sigma_1 \parallel \Sigma_2$ iff $Opns(\Sigma_1) \cap Opns(\Sigma_2) = \emptyset$. Note that $\Sigma_1 \parallel \Sigma_2$ does not imply $\Sigma_1 \cap \Sigma_2 = \emptyset$, since the two signatures may contain common sorts.
- Let $ren = t_i \text{ by } t'_i{}^{i=1..n}, f_j \text{ by } f'_j{}^{j=1..p}$ be a renaming. Then, for any signature Σ over S s.t. $\{t_i{}^{i=1..n}\} \subseteq S$, $\{t'_i{}^{i=1..n}\} \cap S = \emptyset$, $\{f_j{}^{j=1..p}\} \subseteq Opns(\Sigma)$, $\{f'_j{}^{j=1..p}\} \cap Opns(\Sigma) = \emptyset$, $Morph_\Sigma(ren)$ denotes the signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ defined as follows. The signature Σ' is over $\{t'_i{}^{i=1..n}\} \cup (S \setminus \{t_i{}^{i=1..n}\})$ and $Opns(\Sigma') = \{f'_j{}^{j=1..p}\} \cup (Opns(\Sigma) \setminus \{f_j{}^{j=1..p}\})$. For any $t \in S$, if $t = t_i$ for some $i \in \{1, \dots, n\}$ then $\sigma(t) = t'_i$ else $\sigma(t) = t$, and analogously for the operations.
- Let $tdec$ and $tdef$ be a type declaration and definition, respectively. Then $Sorts(tdec)$ and $Sorts(tdef)$ denote the set of sorts declared in $tdec$ and defined in $tdef$, respectively. Analogously, if $fdec$ and $fdef$ are a function declaration and definition, respectively, then $Sig_S(fdec)$ and $Sig_S(fdef)$ denote the signature over S corresponding to the operations declared in $fdec$ and defined in $fdef$, respectively.

Typing rules for the core language There are four kinds of judgments for the language *FL* with the following meaning:

- $tdec \vdash_T tdef$ means that $tdef$ is a statically correct type definition w.r.t. the type declaration $tdec$.
- $\langle tdec; tdef \rangle \vdash_{FDC} fdec$ means that $fdec$ is a statically correct function declaration w.r.t. the type declaration and definition $tdec$ and $tdef$.
- $\Gamma \vdash_{FDF} fdef$, where $\Gamma = \langle tdec; tdef; fdec \rangle$, means that $fdef$ is a statically correct function definition w.r.t. the type declaration and definition $tdec$ and $tdef$ and the function declaration $fdec$. We denote by $Sig(\langle tdec; tdef; fdec \rangle)$ the expression $Sig_S(fdec)$, with $S = Sorts(tdec) \cup Sorts(tdef)$.
- $\Gamma, \bar{x}:\bar{s} \vdash_E e:s$ means that e is a statically correct expression of sort s w.r.t. Γ and the variable declaration $\bar{x}:\bar{s}$.

The typing rules for the language *FL* are straightforward, hence they have been omitted for lack of space (see [1]).

Typing rules for the module language. The judgment for module expressions has the form $\vdash_M M: \langle S^{in}, \Sigma^{in} \rangle \rightarrow \langle S^{out}, \Sigma^{out} \rangle$ and means that M is a statically correct mixin expression over the mixin signature $\langle S^{in}, \Sigma^{in}, S^{out}, \Sigma^{out} \rangle$. The typing rules for mixin expressions are given in Fig.3 and Fig.4. (see [1] for

$$\begin{aligned}
(\text{mrg}) \quad & \frac{\vdash_M M_i : \langle S_i^{\text{in}}, \Sigma_i^{\text{in}} \rangle \rightarrow \langle S_i^{\text{out}}, \Sigma_i^{\text{out}} \rangle, i = 1, 2}{\vdash_M M_1 \oplus M_2 : \langle S^{\text{in}} \setminus S^{\text{fr}}, \Sigma^{\text{in}} \setminus \Sigma^{\text{fr}} \rangle \rightarrow \langle S^{\text{out}}, \Sigma^{\text{out}} \rangle} \\
& \text{where } S_1^{\text{out}} \cap S_2^{\text{out}} = \emptyset, \Sigma_1^{\text{out}} \parallel \Sigma_2^{\text{out}}, S^{\text{in}} = S_1^{\text{in}} \cup S_2^{\text{in}}, \Sigma^{\text{in}} = \Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}} \\
& S^{\text{fr}} = S_1^{\text{out}} \cup S_2^{\text{out}}, \Sigma^{\text{fr}} = (\Sigma_1^{\text{out}} \setminus \Sigma_1^{\text{in}}) \cup (\Sigma_2^{\text{out}} \setminus \Sigma_2^{\text{in}}) \\
& S^{\text{out}} = S_1^{\text{out}} \cup S_2^{\text{out}}, \Sigma^{\text{out}} = \Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}} \\
\\
(\text{frz}) \quad & \frac{\vdash_M M : \langle S^{\text{in}}, \Sigma^{\text{in}} \rangle \rightarrow \langle S^{\text{out}}, \Sigma^{\text{out}} \rangle}{\vdash_M \text{freeze } f_i^{i=1..n} \text{ in } M : \langle S^{\text{in}}, \Sigma^{\text{in}} \setminus \Sigma^{\text{fr}} \rangle \rightarrow \langle S^{\text{out}}, \Sigma^{\text{out}} \rangle} \\
& \text{where } \text{Opns}(\Sigma^{\text{fr}}) = \{f_i^{i=1..n}\}, \Sigma^{\text{fr}} \subseteq \Sigma^{\text{in}} \cap \Sigma^{\text{out}} \\
\\
(\text{rst}) \quad & \frac{\vdash_M M : \langle S^{\text{in}}, \Sigma^{\text{in}} \rangle \rightarrow \langle S^{\text{out}}, \Sigma^{\text{out}} \rangle}{\vdash_M \text{restrict } f_i^{i=1..n} \text{ in } M : \langle S^{\text{in}}, \Sigma^{\text{in}} \rangle \rightarrow \langle S^{\text{out}}, \Sigma^{\text{out}} \setminus \Sigma^{\text{rs}} \rangle} \\
& \text{where } \text{Opns}(\Sigma^{\text{rs}}) = \{f_i^{i=1..n}\}, \Sigma^{\text{rs}} \subseteq \Sigma^{\text{in}} \cap \Sigma^{\text{out}}
\end{aligned}$$

Fig. 3. Typing rules for the module language (first part)

the definitions of inclusion, union, intersection, and difference of many sorted signatures).

The merge operation (mrg) is well-defined whenever there are no conflicting type ($S_1^{\text{out}} \cap S_2^{\text{out}} = \emptyset$) or function ($\Sigma_1^{\text{out}} \parallel \Sigma_2^{\text{out}}$) definitions. Note that the condition $\Sigma_1^{\text{out}} \parallel \Sigma_2^{\text{out}}$ ensures that the union $\Sigma_1^{\text{out}} \cup \Sigma_2^{\text{out}}$ is not an overloaded signature. The set of output sorts S^{out} and the output signature Σ^{out} of the resulting mixin are defined by union, corresponding to the intuition that the defined types and functions of $M_1 \oplus M_2$ are the defined types and functions of M_1 and M_2 . The set of input sorts S^{in} is given by $(S_1^{\text{in}} \cup S_2^{\text{in}}) \setminus (S_1^{\text{out}} \cup S_2^{\text{out}})$ since, by definition, S^{in} and S^{out} must be disjoint. This corresponds to the intuition that the types which remain deferred in $M_1 \oplus M_2$ are those which have not a corresponding definition in the other mixin. Since no requirement exists on $\Sigma^{\text{in}} \cap \Sigma^{\text{out}}$, we could define Σ^{in} as the union $\Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}}$. However, in this way we would obtain the undesired effect that a function f deferred in one mixin and defined as frozen in the other becomes virtual in $M_1 \oplus M_2$; in order to avoid this, the frozen functions of M_i , $i = 1, 2$ (corresponding to $\Sigma_i^{\text{in}} \setminus \Sigma_i^{\text{out}}$, $i = 1, 2$) have to be deleted from $\Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}}$, hence, $\Sigma^{\text{in}} = (\Sigma_1^{\text{in}} \cup \Sigma_2^{\text{in}}) \setminus ((\Sigma_1^{\text{out}} \setminus \Sigma_1^{\text{in}}) \cup (\Sigma_2^{\text{out}} \setminus \Sigma_2^{\text{in}}))$.

The freeze operation (frz) is defined only for virtual functions (side condition $\Sigma^{\text{fr}} \subseteq \Sigma^{\text{in}} \cap \Sigma^{\text{out}}$). The effect on the signature is the deletion of the function symbols $f_i^{i=1..n}$ from the input signature, corresponding to the fact that the virtual functions $f_i^{i=1..n}$ become frozen in the resulting mixin. Note that the signatures of the functions $f_i^{i=1..n}$ in Σ^{fr} are uniquely determined by the side condition $\Sigma^{\text{fr}} \subseteq \Sigma^{\text{in}} \cap \Sigma^{\text{out}}$ and the assumption that signatures are not overloaded.

$$(hd) \quad \frac{\vdash_M M: \langle S^{in}, \Sigma^{in} \rangle \rightarrow \langle S^{out}, \Sigma^{out} \rangle}{\vdash_M \mathbf{hide}_{f_i^{i=1..p}} \mathbf{in} M: \langle S^{in}, \Sigma^{in} \setminus \Sigma^{hd} \rangle \rightarrow \langle S^{out}, \Sigma^{out} \setminus \Sigma^{hd} \rangle}$$

where $Opns(\Sigma^{hd}) = \{f_i^{i=1..p}\}$, $\Sigma^{hd} \subseteq \Sigma^{out}$

$$(rm) \quad \frac{\vdash_M M: \langle S^{in}, \Sigma^{in} \rangle \rightarrow \langle S^{out}, \Sigma^{out} \rangle}{\vdash_M \mathbf{rename\ type\ } t_i \mathbf{ by\ } t'_i{}^{i=1..n} \mathbf{ fn\ } f_j \mathbf{ by\ } f'_j{}^{j=1..p} \mathbf{ in\ } M: \langle \sigma(S^{in}), \sigma(\Sigma^{in}) \rangle \rightarrow \langle \sigma(S^{out}), \sigma(\Sigma^{out}) \rangle}$$

where $\{t_i^{i=1..n}\} \subseteq S^{full}$, $\{t'_i{}^{i=1..n}\} \cap S^{full} = \emptyset$, $\{f_j^{j=1..p}\} \subseteq Opns(\Sigma^{full})$
 $\{f'_j{}^{j=1..p}\} \cap Opns(\Sigma^{full}) = \emptyset$, $\sigma = Morph_{\Sigma^{full}}(t_i \mathbf{ by\ } t'_i{}^{i=1..n}, f_j \mathbf{ by\ } f'_j{}^{j=1..p})$

$$(mxn) \quad \frac{\langle tdec; tdef; fdec \rangle \vdash_{FDF} fdef_f; fdef_v}{\vdash_M \mathbf{mix\ deferred\ type\ } tdec \mathbf{ fn\ } fdec \mathbf{ defined\ type\ } tdef \mathbf{ fn\ } fdef_f \mathbf{ virt\ } fdef_v \mathbf{ end:} \langle S^{in}, \Sigma^{in} \rangle \rightarrow \langle S^{out}, \Sigma^{out} \rangle}$$

where $S^{in} = Sorts(tdec)$, $\Sigma^{in} = Sig_{S^{full}}(fdec) \cup Sig_{S^{full}}(fdef_v)$
 $S^{out} = Sorts(tdef)$, $\Sigma^{out} = Sig_{S^{full}}(fdef_f; fdef_v)$

Fig. 4. Typing rules for the module language (second part)

The restrict operation (rst), like freeze, is defined only for virtual functions. The effect on the signature is the deletion of the function symbols $f_i^{i=1..n}$ from the output signature, corresponding to the fact that the virtual functions $f_i^{i=1..n}$ become deferred in the resulting mixin.

The hiding operation (hd) is meaningful for both virtual and frozen functions (side condition $\Sigma^{hd} \subseteq \Sigma^{out}$). Frozen functions have to be deleted from the output signature whereas virtual functions have to be removed also from the input signature. As already noted, there are cases where types can be hidden, too. Unfortunately, the framework presented here does not allow one to express the semantics of such an hiding operator purely in terms of primitive operators (as happens in [4]; roughly speaking, we miss a primitive operators for deleting types in the input signature). However, when the framework is instantiated over *PAlg*, an ad hoc semantic definition can be given. Finally, a definition independent from the instantiation of the framework could be given by considering a more general (and complicated) definition of mixin signature and model.

The renaming operation (rm) is defined for any kind of components. We assume that renamings are bijective and introduce new names (side conditions $\{t'_i{}^{i=1..n}\} \cap S^{full} = \emptyset$ and $\{f'_j{}^{j=1..p}\} \cap Opns(\Sigma^{full}) = \emptyset$). The signature of the resulting mixin is obtained in a straightforward way by replacing in the signature of M each sort and operation symbol as specified by $t_i \mathbf{ by\ } t'_i{}^{i=1..n}$ and $f_j \mathbf{ by\ } f'_j{}^{j=1..p}$, respectively. The side conditions $\{t_i^{i=1..n}\} \subseteq S^{full}$ and $\{f_j^{j=1..p}\} \subseteq Opns(\Sigma^{full})$ ensure that the renaming is well-defined.

A mixin definition (mxn) is well-typed whenever the type definition $tdef$ is correct w.r.t. $tdec$, the function declaration $fdec$ is correct w.r.t. $\langle tdec; tdef \rangle$ and the function definition $fdef_f; fdef_v$ is correct w.r.t. $\langle tdec; tdef; fdec \rangle$. These conditions are all ensured by the premise $\langle tdec; tdef; fdec \rangle \vdash_{FDF} fdef_f; fdef_v$. Note that the input signature contains both the deferred functions ($Sig_{Sfull}(fdec)$) and the virtual functions ($Sig_{Sfull}(fdef_v)$) of the mixin.

4 Conclusion

We have presented a kernel language of mixin modules, whose definition is based on the algebraic framework defined in [3,4].

We have given several working examples where the use of mixins and of the related operators offered by the language enhances software reusability and extensibility.

Both the static and dynamic semantics of the language have been considered. In particular, we have investigated the possibility of defining mixin modules with deferred types, allowing the combination of mutually recursive type definitions.

For what concerns related work, [10] contain proposals for extending Standard ML with mutually recursive modules³ and the type-theoretical analysis in [9] deals with the problem of recursive type definitions spanning module boundaries in the context of the phase distinction formalism [16].

Other relevant work concerning the design of languages with mixin modules, we cite [7] and the subsequent work in [6]; more recently, we mention the methodological paper [11] and the proposals for mixin-based extensions of Java in [13,2].

Finally, a proposal which looks similar to our work is that of *units* for MzScheme [12].

In comparison to these related papers, the novelty of our approach stands both in the definition of an algebraic framework for modeling mixin languages and in the introduction of type components.

In particular, since the framework is parametric in the core language, it can be used with minimal adjustments for defining the semantics of several different mixin languages.

Concerning type components, our analysis has pointed out the existence of problems related with type definitions: types are components which cannot be handled in the same way as function components, but which need more care.

Finally, the attempt at defining a language of mixins with type components is a good starting point for studying an extension of the calculus for mixin modules defined in [5] with generic type components.

Acknowledgment: the author is grateful to Elena Zucca who encouraged him to write this paper and suggested a number of improvements in earlier drafts.

³ The authors use the name *mixins* for their mutually recursive modules; we prefer to reserve this name to modules which support both mutual recursion and virtual components.

References

1. D. Ancona. *Modular Formal Frameworks for Module Systems*. PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1998.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam - a smooth extension of Java with mixins. Technical Report DISI-TR-99-15, DISI, University of Genova, 1999. Submitted for publication.
3. D. Ancona and E. Zucca. An algebra of mixin modules. In F. Parisi-Presicce, editor, *Proc. 12th International Workshop on Algebraic Development Techniques, WADT '97, Tarquinia, Italy, June 1997. Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 92–106, Berlin, 1998. Springer Verlag.
4. D. Ancona and E. Zucca. A theory of mixin modules: basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
5. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, 1999*, number 1702 in *Lecture Notes in Computer Science*, pages 62–79. Springer Verlag, 1999.
6. G. Banavar and G. Lindstrom. An application framework for module composition tools. In *Proc. of European Conference on Object-Oriented Programming*, number 1098 in *Lecture Notes in Computer Science*, pages 91–113. Springer Verlag, July 1996.
7. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, April 1992. IEEE Computer Society.
8. W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Dept. of Computer Science, Brown University, 1989.
9. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *PLDI'99 - ACM Conf. on Programming Language Design and Implementation*, 1999.
10. D. Duggan and C. Sourelis. Parameterized modules, recursive modules, and mixin modules. In *1998 ACM SIGPLAN Workshop on ML*, pages 87–96, Baltimore, Maryland, September 1998. ACM Press.
11. R.B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Intl. Conf. on Functional Programming 1998*, September 1998.
12. M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI'98 - ACM Conf. on Programming Language Design and Implementation*, pages 236–248, 1998.
13. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183, January 1998.
14. J. A. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journ. ACM*, 39(1):95–146, 1992.
15. S. E. Keene. *Object-Oriented Programming in Common Lisp*. Addison Wesley, 1989.
16. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.
17. D.A. Moon. Object oriented programming with Flavors. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1986*, pages 1–8. ACM Press, November 1986. SIGPLAN Notices, volume 21, number 11.
18. U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 289–297, 1988.

Behavioural Subtyping Relations for Object-Oriented Formalisms^{*}

Clemens Fischer and Heike Wehrheim

Universität Oldenburg, Fachbereich Informatik
Postfach 2503, D-26111 Oldenburg, Germany
{fischer, wehrheim}@informatik.uni-oldenburg.de

Abstract. In this paper we investigate the object-oriented notion of *subtyping* in the context of behavioural formalisms. Subtyping in OO-formalisms is closely related to the concept of inheritance. The central issue in the choice of subtyping relations among classes is the *principle of substitutability*: an instance of the subtype should be usable wherever an instance of the supertype was expected. Depending on the interpretation of “usable”, we obtain a variety of subtyping relations: stronger subtyping relations, allowing one to *share* the subtype instance among different clients without any change compared with the supertype, and weaker relations, restricting the possibilities of interference of different clients on the subtype instance. The subtyping relations are taxonomically ordered in a hierarchy. The concept of “usability” is formalised via *testing scenarios*, which provide alternative characterisations for the subtyping relations.

1 Introduction

The concept of inheritance is one of the key features for the success of object oriented programming languages and design methods. Inheritance supports reusability and extendibility of software, and allows for the design of well structured systems. The purpose of this concept is rather diverse, e.g. Meyer [Mey97] lists 10 different kinds of use of inheritance. The common aspect of all is the conceptual relationship between the superclass and its subclasses: every subclass object “is a” superclass object. A particular strong form of this is-a relationship is *subtyping*. Subtyping in general is a technique used to enhance flexibility of typing in programming languages by allowing to use a value of a subtype wherever a value of a supertype was expected. In object-oriented languages this notion of subtype is lifted to objects: Subtyping requires objects of the sub- and supertype to fulfil the *principle of type substitutability* [WZ88, Lis88]: an instance of the subtype should be usable wherever an instance of the supertype was expected, *without* a client being able to tell the difference. Usually subtyping is defined on the basis of signatures of methods, requiring methods of the subtype to agree with the corresponding methods of the supertype concerning the types of arguments and results (contravariance of arguments and covariance of results). In the context of

^{*} This work was partially funded by the Leibniz Programme of the German Research Council under grant Ol 98/1-1.

distributed systems, however, this is not sufficient for ensuring type substitutability. Additionally to comparing signatures, the *behaviour* of a subtype has to be compared with the behaviour of the supertype. Thus the term “difference” in the definition of type substitutability has a behavioural meaning: clients requesting services from the subtype should not be able to see any difference to the behaviour of the supertype as long as methods from the supertype are used. This allows to replace every supertype object by a subtype object without the former clients having to change their protocol of interaction with the object.

As a practical example consider a scenario with an old-fashioned printer connected to an Ethernet which is jointly used by some people, among others two clients *C* and *E*. An interface to the printer fixes its services and the protocols to be run when using it. When exchanging the printer by a new one, possibly with lots of additional features, the new interface is derived from the old interface by means of inheritance. Client *C* has however not noticed that the printer has been exchanged (or does not care about all the new features), whereas client *E* is immediately busy with trying out all new possibilities. Type substitutability in this context means that client *C* should be able to use the new printer as before without noticing any differences, *even* if client *E* at the same time uses the additional features of the new printer. As an example, a request of client *C* for a black-and-white print should still be accepted even if the printer is currently serving client *E*’s request for a colour print. Inheritance alone does not guarantee this to hold, rather an appropriate notion of subtyping has to be developed and then checked on the interfaces.

Scenarios like this are common in SUN’s Jini technology [Wal99]. It provides a simple standardised way to encapsulate interfaces for all kind of devices based on the Java programming language. It is explicitly designed for heterogeneous systems whose components should be easy to change, extend or replace. Subtyping that solely relies on signatures is not sufficient for expressing restrictions on the behaviour of the subtype needed for a safe replacement.

Several authors have already investigated a notion of *behavioural subtyping* [Ame91, LW94, LW95, Cus91, Nie95, BD99, BBS97, vdAB97, BCP99]. While the definitions of subtyping relations in the first three papers are tailored towards the preservation of invariant and history properties on states (achieved through rules on methods pre- and postconditions plus sometimes so-called constraint rules), the rest of the above cited work base their subtyping relations on some process algebra correctness relation. Process algebra correctness relations directly compare processes via their observable behaviour. The most popular one as a choice for a subtyping relation in object-oriented formalisms is *extension* [BSS87], used in the theory around the process algebra LOTOS [BB87]. The advantage of extension is that it, as the name says, allows *extension of functionality*, which means that the subtype may have new methods in addition to the ones of the supertype. Extension of functionality is clearly a prerequisite for an object oriented subtyping relation. However, the extension relation only weakly fulfils the requirement of type substitutability (as we will see later): the subtype exposes the same behaviour as the supertype only as long as a sequence of services is requested which is a valid trace for the supertype. A badly behaving environment knowing about the

new services of the subtype might interfere with the old services such that a client may indeed see the difference.

In this paper, we propose four behavioural subtyping relations, which vary in the degree in which they fulfil the principle of type substitutability. To be more precise, they all do fulfil this principle, however, under different assumptions on interferences by the environment, ranging from no interference at all to any possible interference. For all these relations, we give an alternative testing characterisation (similar to the testing scenarios of De Nicola and Hennessy [DNH84]), which precisely formalises the idea of type substitutability and thus provides a means for validating that the subtyping relations indeed capture the intended meaning. The basis of our subtyping relations is the correctness relation *failures refinement* from the process algebra CSP [Hoa85]. To allow extension of functionality we modify failures refinement in four different ways, using hiding, restriction and substitution of the new methods. We show that all four relations collapse into failures refinement when the subtype does not add new methods to the supertype. The subtyping relations are taxonomically ordered in a hierarchy which is shown to be strict and has failures refinement as its top element.

We do not consider some concrete object oriented specification language in this paper, rather our work is based on general labelled transitions system (LTS) which are a common framework for defining behavioural semantics. Thus our theory is applicable to a large number of object oriented formal languages. Labelled transition systems are used to describe the behaviour of objects (instances of classes), thus they act as the *behavioural type* of an object. The actions occurring in the LTS may stand for methods called *on* the object as well as methods called *by* the object; i.e. it is not only the objects interface (methods supplied to clients) which makes up the alphabet of the LTS, rather we consider LTS' that capture every observable behaviour of an object.

The rest of this paper is organised as follows. We start in Section 2 with the relevant definitions of transition systems. In Section 3 we present the four new subtyping relations and compare them with each other. Section 4 gives the testing characterisations. In the conclusion we compare the new subtyping relations with the ones already proposed in the literature. A full version of this paper with all proofs is available as [FW99].

2 Definitions

We start with the definition of the relevant concepts underlying our subtyping relation: labelled transition systems, failures of LTS (which are the basis for the definition of failures refinement) and some process algebra operations on LTS. We assume Σ to be a set of actions, τ to be a special *invisible actions*, $\tau \notin \Sigma$ and $\Sigma_\tau = \Sigma \cup \{\tau\}$. The symbol τ plays the usual role of an invisible silent action in process algebras.

Definition 1. A labelled transition system (LTS) is a tuple $T = (Q, \rightarrow, q_0)$ such that

- Q is a set of states,
- $\rightarrow \subseteq Q \times \Sigma_\tau \times Q$ is a transition relation and
- $q_0 \in Q$ is the initial state.

A labelled transition system of a class describes the possible states of an object of this class (identified by particular values of its attributes), the transitions between states

(which methods are enabled in a state and how the state changes with an execution of a method) and the initial state.

We write $q \xrightarrow{a} q'$ if $(q, a, q') \in \rightarrow$ and $T \xrightarrow{a} T'$ if $T = (Q, \rightarrow, q_0)$ and $T' = (Q, \rightarrow, q'_0)$ such that $q_0 \xrightarrow{a} q'_0$. Let $\sigma \in \Sigma_\tau^*$ be a trace and $A \subseteq \Sigma_\tau$ a set of actions. The *projection of σ on A* , $\sigma \upharpoonright A$, is the trace where all occurrences from events not in A are removed.

From the transition relation $\rightarrow$, we derive various other notations, used in the sequel.

Definition 2.

Let $T = (Q, \rightarrow, q_0)$ be a labelled transition system, $q, q' \in Q$, $a_i \in \Sigma_\tau$ and $\sigma \in \Sigma^*$.

- $q \xrightarrow{a_1 \dots a_n} q'$ iff there are states $q_0, q_1, \dots, q_n$ such that $q = q_0$, $q_i \xrightarrow{a_{i+1}} q_{i+1}$ and $q_n = q'$.
- $q \xRightarrow{\sigma} q'$ iff there is a trace $t \in \Sigma_\tau^*$ such that $q \xrightarrow{t} q'$ and $\sigma = t \upharpoonright \Sigma$.
- The set of traces of T is $\text{traces}(T) := \{\sigma \in \Sigma^* \mid \exists q \in Q \bullet q_0 \xRightarrow{\sigma} q\}$.
- A state is stable if no τ transitions are possible: q stable if $q \not\xrightarrow{\tau}$.

The set of enabled actions of a stable state $q \in Q$ is

$$\text{next}(q) := \{a \in \Sigma \mid \exists q' \in Q \bullet q \xrightarrow{a} q'\},$$

its maximal refusals are $\text{refusals}(q) := \Sigma \setminus \text{next}(q)$.

- The set of failures of T is

$$\mathcal{F}(T) := \{(\sigma, X) \in \Sigma^* \times 2^\Sigma \mid \exists q \in Q : q_0 \xRightarrow{\sigma} q \wedge q \text{ stable} \\ \wedge X \subseteq \text{refusals}(q)\}$$

The traces of a transition system describe all possible execution sequences of a system. The failures record the set of actions that can be refused after some trace has been executed. For a transition system T , we let $\alpha(T)$ denote the set of actions used as labels in *reachable* transitions of T . We write $T \xRightarrow{\sigma} \text{dead}$ iff there is some q such that $q_0 \xRightarrow{\sigma} q$ and $\forall a \in \Sigma_\tau : q \not\xrightarrow{a}$. For an LTS T , we let $\text{next}(T)$ stand for $\text{next}(q_0)$.

In this paper we will not be concerned with divergence, i.e. our subtyping relations do not take divergence into account. We expect our theory to carry over to a treatment of divergence; however, to make the principal concepts of subtyping clearer we will resign from it in this paper.

Three operations on transition systems will play a role in the sequel: restriction and hiding, used for defining subtyping relations, and parallel composition, used for testing characterisations. Restriction of an action a from an LTS T removes all transitions labelled with a from the transition relation, thus making all a -steps impossible. Hiding a in T renames all a transitions into τ transitions. Hiding introduces new non-stable states, from which the refusal sets are not taken into account for the failure semantics. Parallel composition with synchronisation over some set A combines two transitions systems in parallel, synchronising transitions labelled with actions in A and interleaving the other transitions.

Definition 3. Let $T = (Q, \rightarrow, q_0)$ be a labelled transition system and $A \subseteq \Sigma$.

1. The restriction of A in T , $T \setminus_\tau A$, is defined as $(Q, \rightarrow', q_0)$ with

$$\rightarrow' = \{(q, a, q') \in \rightarrow \mid a \notin A\}.$$

2. The hiding of A in T , $T \setminus_{\tau} A$, is defined as $(Q, \rightarrow', q_0)$ with

$$\rightarrow' = \{(q, a, q') \in \rightarrow \mid a \notin A\} \cup \{(q, \tau, q') \mid \exists a \in A \bullet (q, a, q') \in \rightarrow\}.$$

3. Let $T_i = (Q_i, \rightarrow_i, q_{0i})$, $i = 1, 2$, be transition systems. The parallel composition of T_1 and T_2 , $T_1 \parallel_A T_2$, is $(Q, \rightarrow, q_0)$ with $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, and

$$\begin{aligned} \rightarrow' = \{ & ((q_1, q_2), a, (q'_1, q'_2)) \mid a \in A \wedge q_i \xrightarrow{a}_i q'_i, i = 1, 2, \text{ or} \\ & a \notin A, q_i \xrightarrow{a} q'_i, q'_{3-i} = q_{3-i}, i \in \{1, 2\} \} \end{aligned}$$

We let the operator $\parallel$ stand for $\parallel_{\emptyset}$. The following proposition follows directly from the definitions.

Proposition 4 (Failures of the operators).

1. $\mathcal{F}(T \setminus_{\tau} A) = \{(\sigma, X) \mid \exists Y \bullet (\sigma, Y) \in \mathcal{F}(T), \sigma \in (\Sigma \setminus A)^*, X \subseteq Y \cup A\}$.
2. $\mathcal{F}(T \setminus_{\tau} A) = \{(\sigma, X) \mid \exists(\sigma', Y) \in \mathcal{F}(T), \sigma = \sigma' \upharpoonright (\Sigma \setminus A) \wedge X \subseteq Y \wedge A \subseteq Y\}$.
3. $\mathcal{F}(T_1 \parallel_A T_2) = \{(\sigma, X) \mid \exists(\sigma_1, X_1) \in \mathcal{F}(T_1), (\sigma_2, X_2) \in \mathcal{F}(T_2),$
 $\sigma \in \sigma_1 \parallel_A \sigma_2, X_1 \setminus A = X_2 \setminus A \wedge X = X_1 \cup X_2\}$.

The operator $\parallel_A$ on traces computes the set of parallel compositions of traces, interleaving the non-synchronised actions and sharing the synchronised actions.

In classical CSP-hiding, only the pairs (σ, Y) from $\mathcal{F}(T)$ are used for which $A \subseteq Y$, i.e. the actions to be hidden are already refused and thus this pair may not give rise to a new non-stable state. For our subtyping relations, we need another kind of hiding operator, which does look at the new non-stable states when computing failures.

Definition 5. Let T be a transition system, A a set of actions. The weak hiding of A in T , $T \setminus_{wh} A$, is defined by the failure semantics

$$\mathcal{F}(T \setminus_{wh} A) = \{(\sigma, X) \mid \exists(\sigma', Y) \in \mathcal{F}(T), \sigma = \sigma' \upharpoonright (\Sigma \setminus A) \wedge X \subseteq Y \cup A\}.$$

We do not give a definition for weak hiding on transition systems here. The difference to classical hiding is that we now also look at non-stable states introduced by hiding some actions. The intention behind weak hiding is that the hidden actions correspond to actions of another user of an object, invisible to a specific client, but *not* under control of the object alone. Thus the invisible actions here play a different role than τ transitions usually do. For this weak hiding operation and for transition systems in general, we have the following connection between deadlocks and failures:

Proposition 6. Let T be a labelled transition system and $\sigma \in \Sigma^*$.

$$(T \xrightarrow{\sigma} \text{dead}) \iff (\sigma, \Sigma) \in \mathcal{F}(T).$$

We will therefore sometimes use the transition system and sometimes just the failure semantics when reasoning about deadlocks. Note that – in contrast to hiding – weak hiding may introduce new deadlocks.

3 Subtyping

Next, we present our subtyping relations on LTS. We use the letter O (for the German word “Oberklasse”) for a superclass and U (“Unterklasse”) for one of its subclasses; our relations define when U can be said to be a subtype of O .

As we have already explained in the introduction, any instance of a subtype should work correctly even if the client believes it is using the supertype. For this reason we make the general assumption from now on that the subclass does not remove any method that was used in the superclass; i. e. we assume $\alpha(O) \subseteq \alpha(U)$. Let $N = \alpha(U) \setminus \alpha(O)$ be the set of new methods of the subclass U .

Failures Refinement. The traditional implementation relation known from the process algebra CSP is failures refinement [BHR84].

Definition 7 (Failures refinement). *A labelled transition system U is a failures refinement of an LTS O (denoted $O \sqsubseteq_{\mathcal{F}} U$) if $\mathcal{F}(U) \subseteq \mathcal{F}(O)$.*

The subclass U might be more deterministic than O ; resolving some of the nondeterministic choices O has. Failures refinement precisely captures the ‘exchange transparent to users’ idea. Therefore all subtyping relations we present in the following are derived from failures refinement. However, failures refinement itself is not suitable as a subtyping relation in object-oriented formalisms, since failures refinement does not allow to introduce new methods (no extension of functionality). From $\mathcal{F}(U) \subseteq \mathcal{F}(O)$ we can deduce $\alpha(U) \subseteq \alpha(O)$ and thus by our assumption $\alpha(U) = \alpha(O)$. For this reason we modify failures refinement in the following to allow extension of functionality, obtaining four different kinds of subtyping relations.

Subtyping relations. For the definition of subtyping relations based on failures refinement, we now have to decide how to treat the new methods. For a comparison of the subclass with the superclass they have to be ignored somehow. There are basically three possibilities: we *restrict* their execution, we *hide* them or we *substitute* them by old methods. Since we have two hiding operations available, this will give rise to four different subtyping relations, which are - witnessed by the testing characterisations of Section 4 - not only definable but also make practical sense.

We start with the first possibility, called *weak subtyping*. The choice of names for this relation (and the ones to follow) will become clear later, latest when we give the testing characterisation. Weak subtyping is a straightforward formalisation of the subtyping idea: As long as we (and any other user of the subclass U) does not call any of the new methods (thus no action from N occurs) U should have the same behaviour as O . But if some action from N is executed, U can do whatever it wants to do. There has to be no relationship to the behaviour of O anymore.

Definition 8 (Weak subtyping). *Let U, O be LTS’ and $N \subseteq \Sigma$. U is a weak subtype of O (denoted $O \sqsubseteq_{\tau}^N U$) iff $O \sqsubseteq_{\mathcal{F}} U \setminus_{\tau} N$.*

A common example for weak subtyping is mode change. As an example, consider a class O describing a simple calculator. We can derive a more sophisticated scientific calculator U from O such that U works in two modes, a stack based mode (where you type 3, 4, + to add 3 and 4) and a normal mode (where you type 3, +, 4). As long as

we do not change the mode we can happily use U as if it were O , but when we want to explore the new functionality we must use U in a completely different way. The name weak behavioural subtyping also appears in the literature on state-based subtyping relations [DL96]; it is in fact quite close to the definition here in that it also compares sub- and supertype by ignoring all computations which invoke new methods.

For the last example we assumed to have exclusive control over U : No one else can change the mode. But this view is often too restrictive. To give an example, let O be a class implementing the methods *push* and *pop* of a shared stack. For performance reasons, the subclass U offers also the method *push2* to push two elements with one step onto the stack. Weak subtyping is not appropriate to relate U and O as it would allow anything to happen after *push2*, rather we expect *push2* to be the same as double *push*.

This is captured by the notion of *safe subtyping*, which is based on some kind of substitution. The idea is that every new action may only lead to some state that was also reachable with a sequence of old actions. Thus every new action can be substituted by a sequence of old actions. For the formal definition we need the following substitution operator on traces.

Definition 9 (Substitutability). Let $N \subseteq \Sigma$. A trace $\sigma' \in (\Sigma \setminus N)^*$ may arise from $\sigma = a_0 a_1 \dots a_n \in \Sigma^*$ by substitution of new methods N ($\sigma' \triangleleft^N \sigma$) iff there exists some function $f : \{0..n\} \rightarrow (\Sigma \setminus N)^*$ such that

1. $\forall i, 0 \leq i \leq n, a_i \in (\Sigma \setminus N) : f(i) = a_i$,
2. $\sigma' = f(0) \wedge f(1) \wedge \dots \wedge f(n)$.

We will use $\triangleleft$ as follows: The trace σ is some trace of the subclass U that is substituted by the trace σ' which is from O . The function f is the mapping from new actions to sequences of old actions. The trace σ' is constructed by applying f on σ (condition 2). Condition 1 guarantees that f does not change actions not in N . The function f is called substitution function.

Proposition 10 (Transitivity of substitutability). Let A, B be sets of actions with $B \subseteq A$. Then $\sigma_1 \triangleleft^A \sigma_2 \triangleleft^B \sigma_3 \Rightarrow \sigma_1 \triangleleft^A \sigma_3$.

Definition 11 (Safe subtyping). Let U, O be LTS' and $N \subseteq \Sigma$. U is a safe subtype of O (denoted $O \sqsubseteq_s^N U$) iff for all $(\sigma, X) \in \mathcal{F}(U)$ there exists some $(\sigma', X') \in \mathcal{F}(O)$ such that $\sigma' \triangleleft^N \sigma$ and $X' = X \cup N$.

Safe subtyping matches most closely the idea of Liskov and Wing's "extension maps" [LW94], which require that any new method has to be explained in terms of the old methods. Thus no invariants can be invalidated.

However, safe subtyping is still too liberal in some cases. Think of a two player game where opponents do alternating moves m_1 and m_2 . Players would clearly dislike a subclass U which introduces the new method m_{212} allowing player 2 to do two of its own moves together with a move of player one in a single step. But this is allowed by safe subtyping as m_{212} could be replaced by m_2, m_1, m_2 . Such undesirable effects of subtypes can be avoided if instead optimal subtyping is used as a subtyping relation.

Definition 12 (Optimal subtyping). Let U, O be LTS' and $N \subseteq \Sigma$. U is an optimal subtype of O (denoted $O \sqsubseteq_{wh}^N U$) iff $O \sqsubseteq_{\mathcal{F}} U \setminus_{wh} N$.

Optimal subtyping is a restricted version of safe subtyping where we can substitute new actions only with the empty trace. It requires a very strong relationship between the superclass and its subclass: we may have new methods, but the execution of these new methods may not alter the behaviour of the superclass at all. That is why weak hiding is used for the definition: the behaviour of the subclass has to be compared with the behaviour of the superclass also in the states in which new methods are executable. We therefore cannot use classical CSP-hiding which ignores refusals at non-stable states.

The last subtyping relation we present is optimistic subtyping. It is derived from normal hiding in the same way as optimal subtyping is derived from weak hiding.

Definition 13 (Optimistic subtyping). Let U, O be LTS' and $N \subseteq \Sigma$. U is an optimistic subtype of O (denoted $O \sqsubseteq_{\tau}^N U$) iff $O \sqsubseteq_{\mathcal{F}} U \setminus_{\tau} N$.

In contrast to optimal subtyping, optimistic subtyping makes assumptions about the behaviour of other users. E. g. let O again be a two player game. The subclass U offers two more methods, *stop* and *continue*. After *stop* no other action can happen before the game is continued. Optimistic subtyping allows this kind of subtyping. It assumes that the environment (the other user) is “friendly” in the sense that it does not delay the game infinitely (hence the term “optimistic”). Technically, this is achieved by not looking at the states of the subclass in which new methods are executable; now the non-stable states are ignored in the comparison. Note that U is not an optimal subtype of O in this example: After the new action *stop* we observe a new refusal that is not visible in O .

Optimistic subtyping is more based on an idea of *implementation* than of *extension*; the new methods are used to implement the behaviour of the superclass and we assume that the environment is willing to fulfil all requests for this new methods.

All subtyping relations are transitive which is a must for a subtyping relation.

Theorem 14 (Transitivity). Let C_1, C_2, C_3 be LTS' such that $\alpha(C_1) \subseteq \alpha(C_2) \subseteq \alpha(C_3)$ and $N_1 = \alpha(C_2) \setminus \alpha(C_1)$ and $N_2 = \alpha(C_3) \setminus \alpha(C_2)$. Then

$$C_1 \sqsubseteq_{\star}^{N_1} C_2 \sqsubseteq_{\star}^{N_2} C_3 \Rightarrow C_1 \sqsubseteq_{\star}^{N_1 \cup N_2} C_3$$

for $\star \in \{r, s, wh, \tau\}$.

Hierarchy. From now on, we will omit the index N of our subtyping relations. When the subclass does not introduce new methods ($\alpha(U) = \alpha(O)$), all refinement relations collapse into failures refinement.

Proposition 15. If $\alpha(U) = \alpha(O)$ then $\sqsubseteq_{\mathcal{F}} = \sqsubseteq_{\tau} = \sqsubseteq_s = \sqsubseteq_r = \sqsubseteq_{wh}$.

The proof is a straightforward application of the definitions.

Figure 1 gives the hierarchy of subtyping relations.

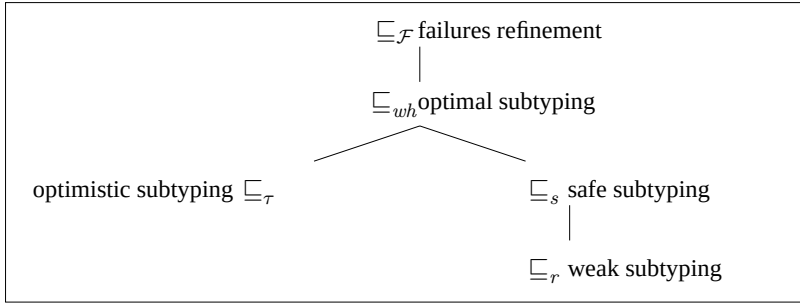


Fig. 1. The hierarchy of subtyping relations

Theorem 16 (Hierarchy of subtyping relations). *The subtyping relations form a hierarchy: $\sqsubseteq_{\mathcal{F}} \subseteq \sqsubseteq_{wh} \subseteq \sqsubseteq_s \subseteq \sqsubseteq_r$ and $\sqsubseteq_{wh} \subseteq \sqsubseteq_{\tau}$.*

Proof: We let N stand for $\alpha(U) \setminus \alpha(O)$.

$\sqsubseteq_{\mathcal{F}} \subseteq \sqsubseteq_{wh}$: Follows from Proposition 15 and the fact that for $O \sqsubseteq_{\mathcal{F}} U$ to hold, the set of reachable actions in U must be a subset of $\alpha(O)$.

$\sqsubseteq_{wh} \subseteq \sqsubseteq_s$: Let $O \sqsubseteq_{wh} U$ and $(\sigma, X) \in \mathcal{F}(U)$. By definition of weak hiding, $(\sigma \upharpoonright (\Sigma \setminus N), Y) \in \mathcal{F}(U \setminus_{wh} N)$ for all $Y \subseteq X \cup N$. By $\sqsubseteq_{wh}$ we get $(\sigma \upharpoonright (\Sigma \setminus N), Y) \in \mathcal{F}(O)$. Moreover $\sigma \upharpoonright (\Sigma \setminus N) \triangleleft \sigma$: function f defined as $f(i) = \sigma[i]$ if $\sigma[i] \in \alpha(O)$, and $f(i) = \varepsilon$ otherwise.

$\sqsubseteq_{wh} \subseteq \sqsubseteq_{\tau}$: Let $O \sqsubseteq_{wh} U$ and assume $(\sigma', X) \in \mathcal{F}(U \setminus_{\tau} N)$. By definition of hiding there must be some $(\sigma, Y) \in \mathcal{F}(U)$ such that $\sigma' = \sigma \upharpoonright (\Sigma \setminus N)$ and $N \subseteq Y$, $X \subseteq N$. By definition of $\sqsubseteq_{wh}$, $(\sigma', X) \in \mathcal{F}(O)$.

$\sqsubseteq_s \subseteq \sqsubseteq_r$: Let $O \sqsubseteq_s U$ and $(\sigma, X) \in \mathcal{F}(U \setminus_r N)$. Then $\sigma \in \alpha(O)^*$. By definition of restriction there is some Y such that $(\sigma, Y) \in \mathcal{F}(U)$ and $X \subseteq Y \cup N$. By definition of $\sqsubseteq_s$ we get $(\sigma, Y \cup N) \in \mathcal{F}(O)$ (note that σ is already in $\alpha(O)^*$) and by subset closure of failure sets we get $(\sigma, X) \in \mathcal{F}(O)$. $\square$

The reverse inclusions do not hold as can be seen by the the informal examples we gave in section 2. Further examples for this fact are given in [FW99].

Theorem 17 (Strictness). *The hierarchy is strict: $\sqsubseteq_{\mathcal{F}} \subsetneq \sqsubseteq_{wh} \subsetneq \sqsubseteq_s \subsetneq \sqsubseteq_r$ and $\sqsubseteq_{wh} \subsetneq \sqsubseteq_{\tau}$ and $\sqsubseteq_{\tau} \subsetneq \sqsubseteq_s \subsetneq \sqsubseteq_r$.*

4 Testing Characterisations

In this section we formalise the principle of type substitutability by a testing scenario. The basic idea is the following: we have a client using the services of classes who acts as an observer for differences between U and O ; a class U is type substitutable for O if every request for a service that O satisfies is also satisfied by U . Formulated the other way round: whenever U composed in parallel with the client (client requesting services of O from U) runs into a deadlock, so will O together with the client. We can vary this

principle by adding an environment which also interacts with U and O . The degree of possible interference of the environment with the services the client requests determines the strength of the principle of type substitutability. The testing characterisations will also make the choice of names for the subtyping relations clearer. First of all, we can divide the relations into two classes: subtyping relations suitable for *shared* mutable objects (more than one client using an object) or for *single access* objects (a single client only). In the first class we find the relations optimal, safe and optimistic, in the latter weak subtyping. The class of subtyping relations for shared objects can further be divided: optimistic subtyping is only suitable if we have a “friendly” environment (hence the name optimistic), whereas safe and optimal subtyping are based on “pessimistic” assumptions on the environment; the environment may interfere with the services of the client. Among the latter, optimal subtyping is the best one, no environment can disturb a service for a client on a subclass that was possible in the superclass.

As a common assumption for all testing characterisations we use the following: the client which plays the part of the observer is denoted C , the environment also accessing U and O is denoted E . We assume that $\alpha(C) \subseteq \alpha(O)$ (client C wants to use the services that O provides) and $\alpha(E) \subseteq \alpha(U)$. Again we use $N = \alpha(U) \setminus \alpha(O)$. The testing scenarios are all of the form: if $U \parallel_S (C \parallel E) \xrightarrow{\sigma} \mathbf{dead}$ then also $O \parallel_{S'} (C \parallel E') \xrightarrow{\sigma'} \mathbf{dead}$. They vary in the allowed environment E (sometimes also empty) and in the synchronisation set S (U may or may not perform actions alone). The idea behind this testing scenario is the following: whenever the request for a specific service can fail on the subclass (subclass, client and environment deadlock), it already could fail on the superclass. Compared with De Nicola and Hennessy testing [DNH84], we have split the testers here into two parts, corresponding to a particular client and the environment, (and the testing so far only captures deadlocks, not divergence¹). This splitting allows us to formally capture the notion of shared and non-shared objects.

For every subtyping relation, we give an equivalent testing characterisation, which describes exactly what kind of type substitutability is captured by its definition. We give the characterisation in the order of the hierarchy, starting at the top.

The testing characterisation for failures refinement is well known from process algebras. We have adapted it here to our particular setting with a client and an environment.

Theorem 18 (Testing characterisation for failures refinement).

Let O, U be transition systems, $\alpha(O) \subseteq \alpha(U)$. The following are equivalent:

1. $O \sqsubseteq_{\mathcal{F}} U$.
2. $\forall C$ such that $\alpha(C) \subseteq \alpha(O)$, $\forall E, \forall \sigma \in \alpha(U)^*$:
 $U \parallel_{\alpha(U)} (C \parallel E) \xrightarrow{\sigma} \mathbf{dead} \Rightarrow O \parallel_{\alpha(U)} (C \parallel E) \xrightarrow{\sigma} \mathbf{dead}$.

Failures refinement fulfils the principle of type substitutability in a very strong form. Whatever is possible with the subclass is also possible with the superclass. However, it can also be seen from the testing characterisation that the subclass U may not have additional methods (it is otherwise straightforward to construct C and E such that the implication does not hold anymore).

¹ If the subtyping relations would be based on failure-divergence refinement instead of failures refinement, we would additionally need a scenario for possible divergences.

The strongest subtyping relation allowing extension of functionality is optimal subtyping, it has the most liberal condition on the environment: the subclass can take the part of the superclass in any possible environment. Addition of new methods is now allowed: whenever the subclass deadlocks when used by a client and a possibly hostile environment, the superclass already deadlocks when used by the same client together with the “same” environment (leaving out the requests for new methods).

Theorem 19 (Testing characterisation for optimal subtyping). *Let O, U be transition systems, $\alpha(O) \subseteq \alpha(U)$. The following are equivalent:*

1. $O \sqsubseteq_{wh} U$.
2. $\forall C$ such that $\alpha(C) \subseteq \alpha(O), \forall E, \forall \sigma \in \alpha(U)^*$:

$$U \parallel_{\alpha(U)} (C \parallel E) \xRightarrow{\sigma} \mathbf{dead} \Rightarrow O \parallel_{\alpha(U)} (C \parallel (E \setminus_{wh} N)) \xRightarrow{\sigma'} \mathbf{dead}$$
for $\sigma' = \sigma \upharpoonright (\Sigma \setminus N)$.

Proof:

1. $\Rightarrow$ 2. Let $O \sqsubseteq_{wh} U$ and $U \parallel_{\alpha(U)} (C \parallel E) \xRightarrow{\sigma} \mathbf{dead}$. Then there must be some X, Y such that $(\sigma, X) \in \mathcal{F}(U)$, $(\sigma, Y) \in \mathcal{F}(C \parallel E)$ and $X \cup Y = \Sigma$. By definition of weak hiding we have $(\sigma \upharpoonright (\Sigma \setminus N), X \cup N) \in \mathcal{F}(U \setminus_{wh} N)$ and $(\sigma \upharpoonright (\Sigma \setminus N), Y \cup N) \in \mathcal{F}((C \parallel E) \setminus_{wh} N)$. With the assumption on alphabets ($\alpha(C) \subseteq \alpha(O)$) we have $(C \parallel E) \setminus_{wh} N = C \parallel (E \setminus_{wh} N)$. With proposition 6 we get the desired result.
2. $\Rightarrow$ 1. let $(\sigma', X) \in \mathcal{F}(U \setminus_{wh} N)$. Hence there are σ, Y such that $(\sigma, Y) \in \mathcal{F}(U)$ and $\sigma' = \sigma \upharpoonright (\Sigma \setminus N)$ and $X \subseteq Y \cup N$. We have to show that $(\sigma', X) \in \mathcal{F}(O)$. For this we construct some client C and environment E :

$$\begin{aligned} C &: c_0 \xRightarrow{\sigma'[1]} c_1 \dots \xRightarrow{\sigma'[n]} c_n \\ E &: e_0 \xRightarrow{\sigma''[1]} e_1 \dots \xRightarrow{\sigma''[m]} e_m \end{aligned}$$

such that $\sigma'' = \sigma \upharpoonright N, |\sigma'| = n, |\sigma''| = m$. Moreover $next(c_n) = Y \setminus N$ and $next(e_m) = Y \setminus (\Sigma \setminus N)$. Thus we get $U \parallel_{\alpha(U)} (C \parallel E) \xRightarrow{\sigma} \mathbf{dead}$. By 2 it follows that $O \parallel_{\alpha(U)} C \xRightarrow{\sigma'} \mathbf{dead}$. By construction we have $C \parallel (E \setminus_{wh} N) = C$. Hence $(\sigma', Y \setminus N \cup N) \in \mathcal{F}(O)$ and by subset closure of failures $(\sigma', X) \in \mathcal{F}(O)$. $\square$

Optimal subtyping guarantees that whenever something goes wrong on the subclass it also might go wrong on the superclass. It in particular goes wrong in the “same” way, with the “same” disturbance by an environment. The subclass thus strongly resembles the behaviour of the superclass.

The environment in the characterisation of optimistic subtyping plays a different role: rather than allowing interference by an environment, we require that (part of) the environment is able to run a fixed protocol together with U (on the new methods of U which might be called by U to achieve some service requested by the client). This is modelled by using a different synchronisation set in the parallel composition of U and C : we use $\alpha(O)$ here and thus U may execute all new methods alone. This corresponds to an environment running exactly the protocol with U that U requests.

Theorem 20 (Testing characterisation for optimistic subtyping). *Let O, U be transition systems, $\alpha(O) \subseteq \alpha(U)$. The following are equivalent:*

1. $O \sqsubseteq_{\tau} U$.
2. $\forall C$, such that $\alpha(C) \subseteq \alpha(O)$, $\forall \sigma \in \alpha(U)^*$:

$$U \parallel_{\alpha(O)} C \xrightarrow{\sigma} \mathbf{dead} \Rightarrow O \parallel_{\alpha(O)} C \xrightarrow{\sigma'} \mathbf{dead} \text{ with } \sigma' = \sigma \upharpoonright (\Sigma \setminus N).$$

This proof and all proofs for the following theorem can be found in [FW99].

Next we give the testing characterisation for safe subtyping. It deviates a little from the form of the former testing characterisations, as does also the definition of safe subtyping. For safe subtyping we again assume that an environment is free to request any service it wants. The idea behind the testing characterisation for safe subtyping is similar to optimal subtyping. If something goes wrong on the subclass, in presence of some environment, then it can also go wrong on the superclass. However, now the disturbance of the environment on the superclass might be different from the disturbance that caused a deadlock on the subclass. This difference is captured by substitution. Since this substituted environment may now also request additional old services, we have to fix the sequence of actions that the client is actually doing in the interaction. This sequence (denoted ξ below) has to remain the same in the usage of the subclass and the superclass (no difference observable).

Theorem 21 (Testing characterisation for safe subtyping). *Let O, U be transition systems, $\alpha(O) \subseteq \alpha(U)$. The following are equivalent:*

1. $O \sqsubseteq_s U$.
2. $\forall C$ such that $\alpha(C) \subseteq \alpha(O)$, $\forall E$, $\forall \sigma \in \alpha(U)^*$:

$$\begin{aligned} & U \parallel_{\alpha(U)} (C \parallel E) \xrightarrow{\sigma} \mathbf{dead} \text{ where } C \xrightarrow{\xi} C' \text{ is } C' \text{'s part of } \sigma \Rightarrow \\ & \exists \sigma', E' \text{ such that } \sigma' \triangleleft \sigma \text{ and} \\ & O \parallel_{\alpha(U)} (C \parallel E') \xrightarrow{\sigma'} \mathbf{dead} \text{ where again } C \xrightarrow{\xi} C' \text{ is } C' \text{'s part of } \sigma'. \end{aligned}$$

While optimal subtyping does not allow “new” failures in the subclass, safe subtyping does allow them, but they have to be explainable in terms of old services.

Weak subtyping is a relation which does not allow sharing of objects at all. As soon as an environment is requesting new services of the subclass anything may go wrong (the subtyping relation ignores any behaviour after the execution of a new method). Weak subtyping defines the requirements on subtyping in a setting with a single client accessing the subtype and supertype. Every deadlock on a subtype may also arise on the supertype.

Theorem 22 (Testing characterisation for weak subtyping). *Let O, U be transition systems, $\alpha(O) \subseteq \alpha(U)$. The following are equivalent:*

1. $O \sqsubseteq_r U$.
2. $\forall C, \forall \sigma \in \alpha(U)^*$:

$$U \parallel_{\alpha(U)} C \xrightarrow{\sigma} \mathbf{dead} \Rightarrow O \parallel_{\alpha(U)} C \xrightarrow{\sigma} \mathbf{dead}.$$

Summarising, for every subtyping relation we have found an equivalent testing characterisation which formalises the notion of type substitutability induced by the relation.

5 Conclusion

In this paper we have proposed four relations which are suitable as behavioural subtyping relations under different assumption on possible interferences by the environment. Weak subtyping is the right relation if we have exclusive access on an object. All other relations are used for shared objects.

Safe subtyping is useful if every new method can be substituted by a sequence of calls of old methods. Optimal subtyping does not allow new methods to change access to old methods at all. Both relations can be used with any other user which also has access to the object. Optimistic subtyping makes only sense if other users ‘use the new methods properly’; thus they don’t introduce unintended deadlocks using new methods. The pragmatics of the four relations was motivated with realistic examples. The four relations are taxonomically ordered in a strict hierarchy, and for every relation we have given an alternative testing characterisation which precisely formalises the degree of type substitutability captured by the relation.

Our results are a contribution for improving correctness issues in software designs for distributed object-oriented systems like the Jini architecture.

The variety of subtyping relations raises the question: Which one to choose in a concrete design? We suggest that one should record the appropriate subtyping relation desired in each (super)class. Thus a class itself should carry the information in which way it may be subtyped. An inherited class is then a correct subtype, if it lies within the intended subtyping relation. This gives the programmer of a superclass the chance to control the reuse and decide for which kind of sharing the class is intended.

Related Work. Extension [BSS87] is the process algebra correctness relation most often discussed as a subtyping relation for object oriented systems. To facilitate discussion, we give the definition here. Let the refusals of a transition system T after some trace σ be $Ref(T, \sigma) := \{X \mid \exists q \in S : q_0 \xrightarrow{\sigma} q \wedge X \subseteq refusals(q)\}$.

Definition 23.

$U \text{ ext } O$ iff $traces(U) \supseteq traces(O)$ and $\forall \sigma \in traces(O) : Ref(U, \sigma) \subseteq Ref(O, \sigma)$.

Extension allows extension of functionality, not only on new methods but also on old methods: the subtype may have new traces on old methods. This makes extension incompatible with all of our subtyping relations: the subtype may exhibit new behaviour even if no new methods are added to the supertype. Proposition 15 (all relations agree with failures refinement if no new methods are introduced) hence does not hold for extension. Our main assumption that the basis of subtyping is reduction of nondeterminism is thus not fulfilled. From the point of view of type substitutability this means that an observer cannot see a difference only as long as he requests chains of services (executes traces) that are part of the supertypes behaviour. If this is not guaranteed he might indeed see a difference, even if he is only requesting old services. As a second point, extension is not a suitable subtyping relation in an environment with shared objects (same problems as weak subtyping).

A definition of subtyping combining extension with restriction and hiding (with Petri nets as models) has been suggested in [BCP99]. In principle the same remarks as

above apply to these relations. Additionally [BCP99] propose *implementation inheritance*, a relation which captures the code-reuse aspect of inheritance.

Bowman, Derrick and others [BBSDS97] have already analysed the inadequateness of extension. In [BBSDS97] they have therefore proposed a different subtyping relation for the process algebra LOTOS. Interestingly, their proposed relation is also not able to distinguish the two processes, which are used as an example for the inadequateness of extension as a subtyping relation.

An approach to defining implementation relations somehow similar to our safe subtyping is the work of Gorrieri and Rensink [RG97], which combines process algebra correctness relations with action refinement. The context of use is however completely different: *vertical implementation relations* are used to describe the relation between systems specified on different levels of abstraction and thus do for instance allow a complete change of alphabet, whereas subtyping just adds new behaviour to an already existing class and thus stays on the same level of abstraction.

Similarities, however on a quite different level, can also be found in the area of extensions of abstract data types (ADTs) (see e.g. [EM85, LP]).

Future Work. We intend to further investigate subtyping relations in the context of formal methods combining state-based and behavioural formalisms, in particular for the formal method CSP-OZ [Fis97], a combination of CSP and Object-Z. The semantic basis for CSP-OZ is CSP's failure divergence model. Due to the integration of a state-based with a behavioural formalism, failures refinement in CSP-OZ is closely connected with *data refinement*, widely used as a refinement concept in purely data-oriented specifications. Similar to data refinement conditions, we expect to obtain subtyping conditions on the Z part of a CSP-OZ specification, which help in verifying subtyping relations among classes.

References

- [Ame91] P. America. Designing an object-oriented programming language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX Workshop: Foundations of Object-Oriented Languages*, number 489 in LNCS. Springer, 1991.
- [BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
- [BBSDS97] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On behavioural subtyping in LOTOS. In H. Bowman and J. Derrick, editors, *FMOODS 97*, pages 335 – 351. Chapman & Hall, 1997.
- [BCP99] C. Balzarotti, F. De Cindio, and L. Pomello. Observation equivalences for the semantics of inheritance. In P. Ciancarini and R. Gorrieri, editors, *FMOODS '99*, 1999.
- [BD99] H. Bowman and J. Derrick. A junction between state based and behavioural specification. In P. Ciancarini, F. Fantechi, and R. Gorrieri, editors, *FMOODS '99*, pages 213 – 239. Kluwer, 1999.
- [BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

- [BSS87] E. Brinksma, G. Scollo, and Ch. Steenbergen. LOTOS specifications, their implementations and their tests. In B. Sarikaya and G. v. Bochmann, editors, *PSTV*, pages 349 – 358. Elsevier, 1987.
- [Cus91] E. Cusack. Refinement, conformance and inheritance. *Formal Aspects of Computing*, 3:129 – 141, 1991.
- [DL96] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.
- [DNH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
- [EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6. Springer, 1985.
- [Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *FMOODS '97*, volume 2, pages 423–438. Chapman & Hall, 1997.
- [FW99] C. Fischer and H. Wehrheim. A hierarchy of behavioural subtyping relations (full version). Technical Report TRCF-99-1, University of Oldenburg, October 1999. <http://semantik.Informatik.Uni-Oldenburg.DE/~wehrheim>
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [Lis88] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5):17–34, May 1988.
- [LP] G. T. Leavens and D. Pigozzi. A complete algebraic characterization of behavioral subtyping. to appear in *Acta Informatica*, see also <ftp://ftp.cs.iastate.edu/pub/techreports/TR96-15/TR.ps.gz>.
- [LW94] B. Liskov and J. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811 – 1841, 1994.
- [LW95] G.T. Leavens and W.E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.
- [Mey97] B. Meyer. *Object-Oriented Software Construction*. ISE, 2. edition, 1997.
- [Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-oriented software composition*, pages 99 – 121. Prentice Hall, 1995.
- [RG97] A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97*, volume 1214 of *LNCS*, pages 772–786, 1997.
- [vdAB97] W.M.P. van der Aalst and T. Basten. Life-cycle inheritance - a Petri-net-based approach. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets*, number 1248 in *LNCS*, pages 62–81. Springer, 1997.
- [Wal99] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.
- [WZ88] P. Wegner and B. Zdonik. Inheritance as an incremental modification mechanism or What like is and isn't like. In *ECOOP'88*, volume 322 of *LNCS*. Springer, 1988.

Applying Category Theory to Derive Engineering Software from Encoded Knowledge

Michael Healy¹ and Keith Williamson²

¹ The University of Washington,
co 13544 23rd Place NE, Seattle, Washington 98125, USA
mjhealy@u.washington.edu

² The Boeing Company, PO Box 3707 MS 7L-66, Seattle, Washington 98124, USA
keith.e.williamson@boeing.com

Abstract. In an industrial research project, we have demonstrated the feasibility of applying category-theoretic methods to the specification, synthesis, and maintenance of industrial strength software systems. The demonstration used a first-of-its-kind tool for this purpose, Kestrel's **Specware**TM software development system. We describe our experiences and discuss broadening the application of such category-theoretic methods in industry.

Although the technology is promising, it needs additional development to make it generally usable. This is not surprising given its mathematical foundation. On the other hand, we believe our demonstration is a turning point in the use of mathematically rigorous approaches in industrial software development and maintenance. We have demonstrated here the capture via mathematical methods not only of software engineering design rationale, but also of the product design and manufacturing process rationale used by different engineering disciplines, and the production of usable software directly from the captured rationale. We feel that that further evolution of the tools for this technology will make formal systems engineering a reality.

1 Introduction

Industrial software developers are operating under an increasingly challenging workload. Their clients want development and maintenance of large and business-critical software systems made faster, cheaper, and better. Conspiring against this are high turnover rates in software analysts and engineers, the creators and implementors not only of software systems but of the institutional knowledge they represent. Is there a way to capture institutional knowledge efficiently and reliably in software development, and make it transparent and easily reusable so that system maintenance and use is less dependent upon the memories of individuals, system environment changes can be accommodated without the present major disruptions, and system knowledge can flow across organizational boundaries and through time free of the brittleness and inflexibility of legacy software?

We have been experimenting with a new approach to addressing these questions. Transparency and flexibility of the new approach accrues from the mathematical rigor upon which it is based. This statement would seem paradoxical to the industrial software developers who are the desired users of our approach. Most are highly skeptical of formal methods because of their perceived opacity and Inflexibility, together with the lack of success stories accompanying their use. Strong data typing and the requirement of deciphering program code to form provable assertions in a formal language are regarded with disdain: These activities are time-consuming and produce nothing. The developers, meanwhile, are under increasingly intense pressure to produce working software that helps build airplanes. The formal methods advocate must demonstrate an approach that helps accelerate the process by preventing errors upstream in the work flow, where most of the errors begin. The approach must do more, however. To avoid taxing the patience of developers under pressure to produce, it must compensate for the time spent using it by accelerating the later stages of software production. This must be demonstrated time and again, in projects of increasing complexity and magnitude, to convince the skeptics that the resulting methodology is indeed “industrial strength”.

What we hope to establish is that constructive methods are available, developed on the mathematical foundation of category theory. As a result of the mathematical foundation, software development and maintenance can become a true engineering occupation, the engineering of industrial knowledge. On the other hand, the tools involved are at an early stage of development and the mathematics behind them must undergo further translation. Category theory, even more than most formal methods, can occupy a dedicated student for years. There is a great deal of research ahead in making our approach fully practicable across industry.

In this paper, we describe our experiences in applying a category-theoretic approach to software specification, synthesis, and maintenance. In addition, we share an industrial perspective on what is needed to broaden the appeal of this technology to industry. We begin with some formal preliminaries and a brief discussion of **Specware**TM [11], the category-theory-based tool we have been using.

2 Formal Preliminaries

Recently, category-theoretic methods have been applied directly in software development. The methodology described here is based upon a categorical approach introduced in [6]. The basic idea is to refine formal specifications into software by operations performed upon diagrams in certain categories, chiefly the automatic derivation of colimits. Earlier work established the theoretical background for this approach [4,5,7]. Several categories are involved here, beginning with the category of formal specifications and their morphisms. We express the many different aspects of programming in specifications of theories written in formal logic. The morphisms express the part-to-whole relationships of simple theories

to more complex ones by mapping the terms, sorts, and axioms of the simpler to the more complex theory. A basic property of a morphism is that the axioms of the domain of a morphism are theorems of the codomain (here, we are speaking of theories and their specifications interchangeably, taking some expository license with the formalisms used). The composition of morphisms then allows us to transition from abstract theories to constructive, complex theories in a succession of diagram refinement steps employing colimits. The colimits “put everything together”, constructing a single theory from its component theories and their relationships.

A functorial, colimit-preserving mapping of diagrams into a category of programming language constructs allows the automatic construction of software as one end-item of this process, but there is more. The theories, morphisms and diagrams provide a valuable infrastructure for software development, maintenance, re-use and re-engineering. Should users’ requirements change or the operating environment of an application vary across platforms and engineering and manufacturing organizations, we can make the required changes to the software by replaying the diagram refinement with some theories and morphisms added or replaced.

2.1 Commutative Diagrams and Colimits

A diagram in a category is a means of highlighting some objects and morphisms between them for special consideration. For example, we can obtain a theory of points as pairs of integers by combining a theory of pairs and a theory of integers along the common theory **Triv**. The theory **Triv** contains nothing but a sort symbol, and the morphisms from it to the other two theories of the base diagram for integer-points represents the fact that the sort of integers is the same as the sort from which pairs are formed in the put-together theory (Figure 2.1).

The colimit of a diagram is a cone, with morphisms from each diagram object to an apical object. The diagram so formed (the defining diagram, where the original diagram is called the base diagram), with the cone added to the original diagram, is commutative. Because of this, any specification T_0 of the base diagram which is “shared” by two or more other specifications T_i of the base diagram, with morphisms $m_i: T_0 \rightarrow T_i$ ($i \neq 0$), becomes mapped into the apical object in only one way, since any two compositions along paths from it to the apical object are equal. Further, a colimit is an initial cone for its base diagram, so that there is a unique morphism from its apical object to the apical object of any other cone for the same base diagram with the two cones together forming a commutative diagram. The significance of this is that the theory represented by the colimit apex is minimal with respect to the objects and morphisms in the base diagram: No extraneous information is included. Thus, we can build theories with a clear semantics in terms of their component theories and the relationships among the components. We can re-use such theories by including them in different ways in diagrams and taking the colimits, as indicated in Figure 2.1 where the theory of points is used to make theories of lists of points and sets of points.

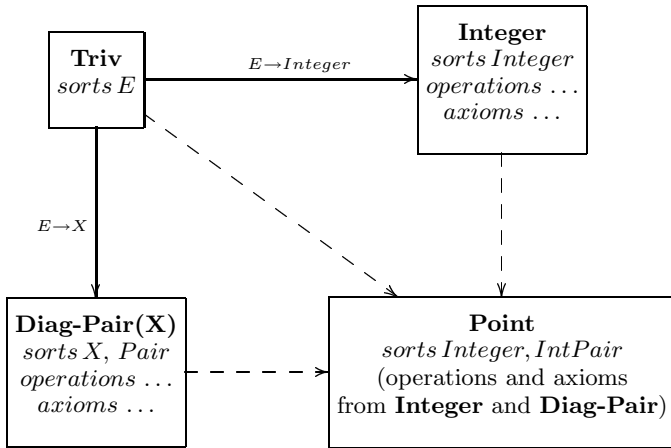


Fig. 1. Deriving a specification for points as integer-pairs. The colimit object contains the specifications for integers and pairs “welded together” so that the paired sort is the sort of integers.

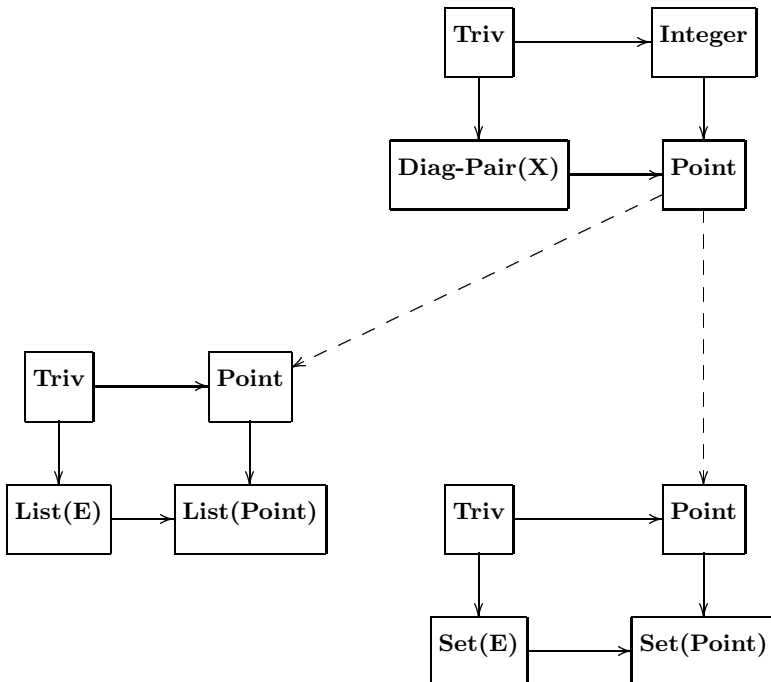


Fig. 2. Example of specification reuse: Colimit specification for points derived as pairs of integers, then re-used in further colimit derivations of lists of points and sets of points.

In our applications, we have colimit diagram derivations such as that in Figure 3. With such derivations, we combine domain theories, algorithm theories, and others to express the semantics of a software program as a construction from elementary “semantic modules” which are relatively easy to verify for correctness. The mathematical rigor of the colimit construction guarantees (in principle at least) that the combination is as correct as its components and their morphisms. As an example, we combine a theory of material properties, formed by adding definitions and axioms to a simple theory of physical properties (volume, mass, etc.) with a simple theory of geometry along a common theory of real numbers¹. In the colimit apex, a theory of airplane parts, there is only one copy of the real numbers, since they can be embedded in the **Parts** theory in only one way despite their having been embedded separately in the theories of physical quantities and geometry in the original diagram. The colimit is calculated by an algorithm that implements coproduct-coequalizer derivations (based upon the dual to Theorem 1.9.7 in Pierce[8]).

To date, we have produced two industrial applications. The first [13,14] was a computer-aided design (CAD) software code for a certain kind of airplane part, a panel optimized for weight subject to loads and other design constraints—and manufacturing constraints as well (see Figure 4). The software is meant to operate within an intelligent CAD system, accepting panel sizing information as input and generating a panel design with holes drilled to save weight subject to the constraints. Aside from automated code generation, this first application allowed us to demonstrate an immediate advantage of the use of formal methods in software design: The engineering and manufacturing rationale that determined the software design was clearly and unambiguously expressed in the theories and morphisms that formed the refinement constructs. We were able to further demonstrate another major advantage: The refinement process which produced the software could easily be “re-played” to re-generate the software after changing the assumptions used to characterize the application. This was done with a single change to a manufacturing process theory in a diagram that appears later than that of Figure 3 in the refinement process. The resulting software code optimized the airplane part using a different objective, based upon changed process assumptions. This was our proof-of-concept demonstration.

The follow-on challenge to our proof-of-concept demonstration was a scale-up demonstration. Many industrial applications are a great deal more involved than the simple CAD application, resulting in an order of magnitude more code. We chose to produce a program for solving a discrete optimization problem—placing electrical components within a commercial airliner. Problem constraints include electrical component bay locations, and shelf and supporting equipment locations within the bays. There are many others—for example, many components are actively air-cooled, so must be placed near air-cooling equipment. Redundancy is an important factor in airplane design; redundant components must not be located in proximity to one another. The problem objective is to minimize

¹ The “geometry” and “real numbers” theories were produced purely for expediency in this application and lack mathematical elegance.

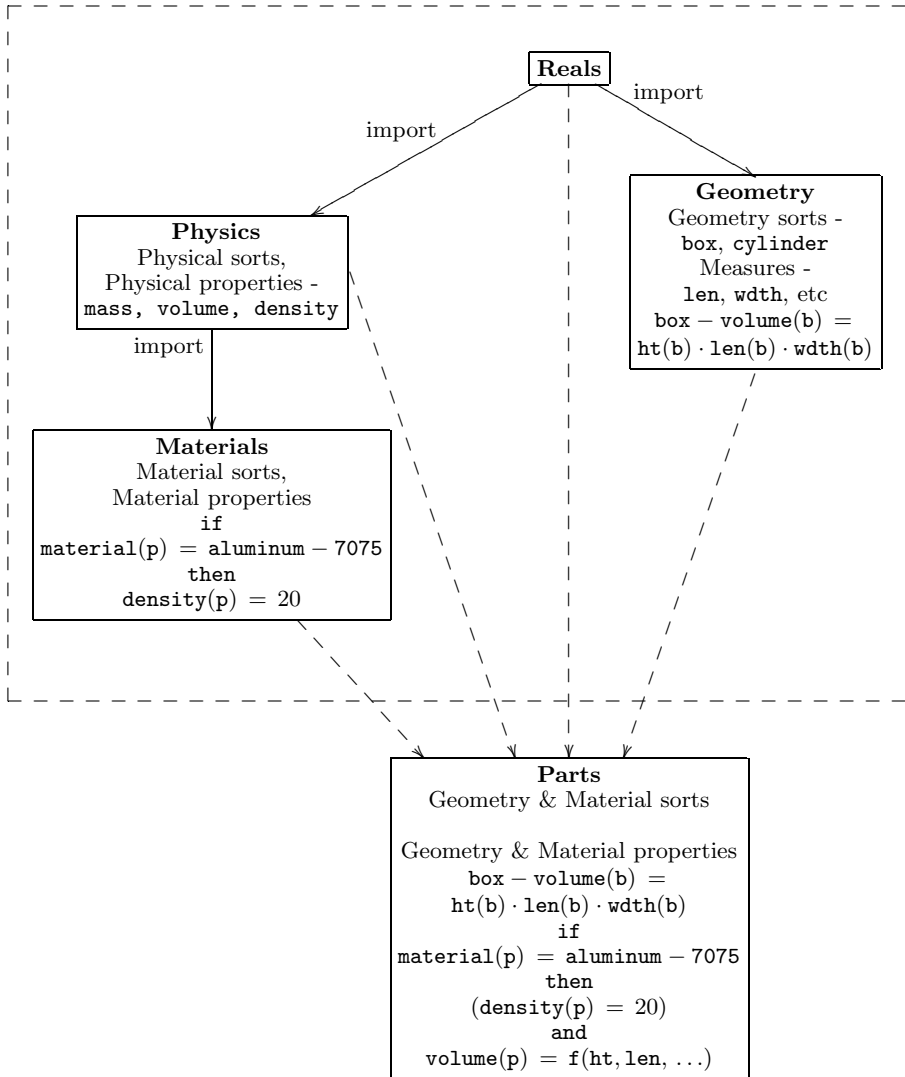


Fig.3. A colimit derivation for a theory of airplane parts, combining theories of materials and geometry along the common theory of reals. Here, the theory of reals is a convenient abstraction of the real datatype often appearing in programming languages. The theories of materials and geometry used here are similar. Mathematically correct theories (or “first principles”) would require more time to derive but would be better for re-useability.

the total length of wire needed to connect the components. This optimization problem is quite complex and good solutions are critical, making this a truly “industrial strength” application. Airplane layout and equipment placement design know-how were encoded as formalized knowledge in domain theories. Optimization problem-solving knowledge and the problem to be solved (i.e., the requirements) made up other theories. The following theories, with appropriate morphisms, made up the refinement process:

- * Geometry²,
- * Global and relative part positioning,
- * Major airplane part and zone definitions,
- * Operations and properties for pieces of equipment, shelves, and racks,
- * Separation and redundancy requirements for equipment,
- * An assignment of a piece of equipment to a position on a shelf,
- * A layout (a set of assignments),
- * Hard constraints on layouts,
- * Cost function on layouts,
- * Theories of classes of optimization problems,
- * Theories of search algorithms for classes of optimization problems,
- * The equipment locator problem statement,
- * A theory of optimization based upon branch and bound.

Roughly 7,000 lines of requirement and design specifications are needed to specify the statement and solution of this problem. The generated (Lisp) software exceeds 7,000 lines.

Again, we demonstrated not only refinement to a finished code based upon clearly-expressed design rationale, but also the ability to re-generate the code to meet changed requirements after making spot changes to component theories. This time, we were in a position to lay claim to a unique in-house capability to produce “industrial-strength” software with a methodology having the stated advantages.

This has turned out to be insufficient to guarantee implementation of this methodology. We shall discuss the reasons for this following a brief description of the tool we used.

3 The Software Development Tool Used

The tool we currently use for the categorical software development process is **Specware**TM, a software development and maintenance environment supporting the specification, design, and semi-automated synthesis of software. **Specware**TM represents a confluence of capabilities and lessons learned from Kestrel’s earlier prototype systems for software development, KIDS[9], REACTO[12], and DTRE[2]. It differs from the earlier systems in having a fully formal basis in category theory. It supports the automation of:

² Again, expedient.

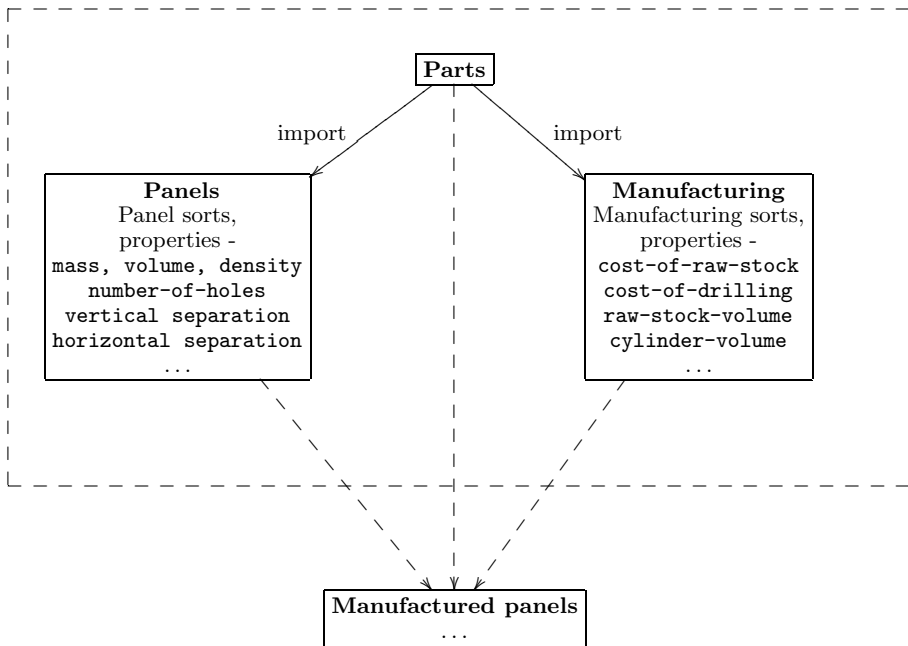


Fig. 4. A colimit derivation for a theory of manufactured panels, combining theories of panels and manufacturing along the common theory of parts.

- * Component-based specification of programs using a graphical interface
- * Incremental refinement of specifications into correct code in various target programming languages (currently C++ and LISP)
- * Recording and experimenting with design decisions
- * Domain-knowledge capture, verification and manipulation
- * Design and synthesis of software architectures
- * Design and synthesis of algorithm schemas
- * Design and synthesis of reactive systems
- * Data-type refinement
- * Program optimization

The **SpecwareTM** system has some of its roots in the formal methods community. There are numerous formal languages that have been used for specifying software systems, among them Z[10] and the Vienna Development Method (VDM) [1]. Of the many formal specification languages, VDM is one of the few that tries to formally tie software requirement specifications to their implementations in programming languages. This system is perhaps the closest to **SpecwareTM** in that it allows an engineer to specify a software system at multiple levels of refinement. VDM tools allow for the capture and discharge (sometimes manually)

of “proof obligations” that arise in the refinement of a specification from one level to another.

4 Getting the New Technology into Routine Use in Industry

We can state an important principle of technology transfer in industry based upon upon many person-years of experience (ours and other researchers’ as well): All the effort in obtaining funding for, doing the research and development on, and demonstrating a new technology seems small in comparison to the daunting task of getting it into everyday use.

Resistance comes from many quarters. Implementing a new technology generates a cost spike that attracts unfavorable attention. We need mention only the cost of training engineers and technicians to use a new technology based upon “esoteric mathematics” (in our case, algebraic methods in the form of category theory). The mathematical background of most industrial engineers encompasses, at most, second-year calculus, a semester of ordinary differential equations and a class in complex variables—no discrete mathematics, no abstract algebra and certainly no category theory. Resistance can come directly from technical people who have difficulty with the notion that they must learn a “new mathematics”. In fact, we find that most computer science graduates hired by industry (including PhDs) have less in their mathematical backgrounds than most engineers—and do not readily agree to improve! In fact, one might ask: How many computer science graduates actually spend much time with discrete mathematics in today’s curriculum, and how well does the average discrete mathematics course address the kinds of mathematics involved in the various algebraic methodologies? We think the answers are very discouraging for the adoption of mathematically-based software development methods. And, finally, training alone will not solve the difficulties in using a software tool that does not, in fact, have a relatively clear and straightforward usage. The ultimate resistance to introduction of a new technology occurs “in the trenches”—with the people who must use it in their everyday work but find it too difficult and time-consuming to be of practical use.

In the following, we shall cover areas in which our technology can be improved and made more amenable to technology transfer. We shall express much of this in terms of the particular tool we used in our effort. First, we present a brief discussion of our work flow.

5 Work Flow in Our Trial Applications

After having spent some time learning some of the basics of category theory as used in this methodology, we went on to learn how to use **Specware**TM while working on the panel layout problem. We also experimented with development of theories closer to pure mathematics, with a view to eventually developing

theories based upon “first principles” for physics, geometry and other areas that might lead into applications. We feel that such theories are essential for true re-use of the theory base at the earlier stages of refinement, where re-use can occur most effectively. Following our initial work, we went on to our second application, the “industrial strength” equipment locator problem.

We first proceeded to learn about the domain. We had three informal requirement documents to work from. These comprised about 20 pages of writing, drawings, etc. In addition, we had several pieces of supporting material (tables, drawings, etc). Only two discussions with a domain expert were needed, since the requirement documents were fairly clear and complete. Once we understood the requirements, we formalized them. We captured roughly 98% of the requirements found in the informal material. The remaining 2% dealt with the interfacing of the software system we were to generate with other software systems. At that time, we had insufficient information for dealing with such issues.

Following our initial formalization we performed a manual validation in which we compared the formal requirements with the informal ones. We wrote a brief document noting places where any of the following held:

- * Requirements were not formalized (the 2% just mentioned).
- * Additional detail was needed to formalize the requirements (due to some degree of ambiguity in the informal documents).
- * Some choice was made between alternate interpretations of the written material (since the three informal documents were written at different times, there were minor inconsistencies).

Having documented these issues, we made and then encoded our design decisions. This involved further decomposition and abstraction. For each design decision, we needed to choose data structures for sorts and algorithms for operators. **Specware**TM has many built-in specifications that can be used for this and other purposes. This was of enormous help in beginning the creation of an infrastructure of theories and morphisms for re-use in many applications. Included are specifications for sets, lists, and sets interpreted as lists, all helpful in developing abstract data structures for software design. The design decisions needed verification to ensure that requirement properties were upheld. Fortunately (again), **Specware**TM has a built-in theorem prover. This was used to prove roughly 25% of the proof obligations. The other 75% were proven by hand. Eventually, we refined every sort and operation from abstract theoretical construct down to some implementable data structure or operation provided by the the target programming language (Lisp was the one we used—the **Specware**TM version we had access to also had C++ implemented, but this did not always work with our compiler and was less straightforward to the one of us (Williamson) doing the bulk of the work).

Once an initial pass at generating the software had been performed, we maintained the software through further use of **Specware**TM. As we learned more about the problem domain, we made several changes to the requirement specifications. It proved to be easy to re-generate the software from the changed specifications. This was partly due to the fact that none of the changes required

significant redesign. (Eventually, we encountered a change that did.) The initial optimization algorithm used an exhaustive search (for expediency: A discrete optimization expert was not available and our time was limited). The resulting inefficiency led us to encode an additional theory for branch and bound optimization. This theory, unlike many of our earlier, problem-specific ones, is widely reusable. For us, then, maintenance amounts to experimentation with better theories and diagrams. This is not as far from standard software maintenance as it might appear. Industrial developers are always experimenting with improvements and add-ons to their software, the difference being that the experimentation occurs at the software code level instead of in the theory abstractions as in our case.

6 Suggested Improvements

6.1 A Methodology for Understanding Refinement System Interfaces

For maximum ease of use, the user interface of any system must reflect and reinforce the user's mental model of the artifacts and processes involved. The developer of a user interface must keep in mind the education, experience, and general characteristics of the user community. To make the tool usable by a broad range of engineers and/or computer scientists, items requiring an explicit, in-depth knowledge of category theory must be avoided, at least for the current generation of users. It would be best if human factors engineers were called upon to perform a preliminary task analysis for using this technology, including an analysis of domain modeling and requirements elicitation techniques that would be helpful. The emerging understanding of the usability issues with this technology could be used in specifying a user interface and an accompanying methodology for its use.

6.2 Linking Nonformal Requirements to Formal Requirements

To put it bluntly, the people most familiar with application domains may not be interested in authoring requirements in a language based upon higher-order sorted logic and category theory. Natural languages are more widely known and accepted, and provide a sort of freedom in ambiguity of expression with which industrial analysts feel comfortable. The restrictions of rules that seem artificial to their work, however correct these rules might be in the underlying theory, will drive these analysts to seek alternatives to any formal methods that impose such rules. This has been repeatedly observed in our context. In addition, individual communities often have their own notations and/or visual symbology, a phenomenon quite typical of traditional engineering workgroups, not to mention entire disciplines. To make formal or mathematically-based methods of any kind attractive, there must evolve a linkage mechanism to unite the less structured representations of requirements with their formal counterparts. For this, we suggest a contemporary sort of solution: A Web-based interface in which engineers

can “click on” a portion of a natural language document and thereby gain access to formal requirement constructs, usable in a modular fashion. The underlying theory would automate the composition of modules, tracking the user who would be “browsing” the informal requirement document.

6.3 Viewing the Linkage between Requirements, Design, and Software

In the same vein, it would be nice to be able to “click on” formal requirement specifications through the Web-based interface and view the design specifications that reflect those requirements. Specification elements could be viewed individually in this manner—sorts, operations, axioms, and theorems. Parts of the design specifications could be connected to other, related parts in similar fashion, or the design specifications could actually consist of a class of design alternatives, providing the user with a menu of possibilities. The underlying compositionality of the mathematical model would provide the mechanism for putting together elementary specifications for either requirements or designs, again connected through the Web interface. Finally, design specifications could be connected to software fragments or to whole software modules (again, exploiting the compositionality)—“just a click away”.

The theme in all this is visibility and traceability of requirements and designs through the software derivation history. If another analyst were to take over our work on the equipment locator problem, what could be provided to them that would make it easier for them to understand how requirements and designs are achieved throughout the software derivation? We feel that this is the most important question remaining to be answered.

6.4 Better Derivation Replay and Visibility

Replaying the software derivation for the equipment locator problem requires the execution of 150 operations via the diagrammatic graphical user interface. While this is not difficult, the tool needs to better automate the task by recording the sequence of steps so that they can be replayed simply by “pushing a button”. There is structure to these sequences of steps[6]. This structure appears easy to capture, display, and manipulate through the user interface. Note that in making this suggestion, we differ from our previous discussion in the granularity of the refinement constructs being tracked. Here, we suggest tracking objects at the level of specifications and diagrams as opposed to the specification fragments discussed in the previous subsection.

6.5 Improved Specification Libraries

Larger specification libraries available as an initial infrastructure within tools like **Specware**TM would greatly increase productivity. Obvious examples are theories about the constructs encountered in data structure and algorithm courses—trees,

graphs, search algorithms, disjoint set-union-find, and so forth. Among the many others one can name are theories for dimensional analysis and unit conversion of physical quantities. We have experimented with a system of theories about physical quantities known as the Engineering Mathematics Ontology, developed at Stanford[3], and also with a system of theories of pure geometry.

6.6 Better Proof Support

Use of the automated theorem prover currently supplied with **Specware**TM has forced some limitations upon our use of mechanical theorem-proving. A more widely usable theorem-prover would be a great boon to this technology, as would equation solvers and model checkers, individually or in combination. A more pragmatic approach might be to use a combination of proof checking and automated theorem proving. Why not allow software designers who are used to concocting proofs manually to record their proof steps? Perhaps a proof checker can validate some proof steps fairly easily. Some proof steps are cumbersome, but it might be possible to discharge them interactively. In the worse case, the tool could support the recording of manually-generated but unverified proofs, and these could then be verified by the proof checker manually when convenient.

6.7 Generating State Based Programs

The current version of **Specware**TM generates only functioning software. No use of state based variables is possible. This limits the efficiency of generated software as well as the use of **Specware**TM in embedded systems, where improvements to the quality of generated software are highly desirable.

6.8 Software Optimization Transformations

The Kestrel Interactive Software Development system (KIDS)[9] has some very useful program-level optimization transformations. These range from low-level transformations that are similar to compiler optimizations, to high-level transformations that perform finite differencing and a simple type of colimit construction (pushouts). The program-level and finite-differencing transformations add to the pushouts the capability to optimize the derived software for efficiency.

6.9 Expressing Rationale behind Design Choices

During design, there are often multiple design alternatives. It is desirable to have a capability to clearly record these alternatives as well as the rationale behind the selection among them that one makes. The rationale could be stated formally, as a trade-off between space and time complexity, for example. When changes in requirements occur over time, this information would be very valuable. All that **Specware**TM currently stores in a derivation history is a single design alternative. Yet, an initial exploration of a design space yields a great deal of information about the trade-offs available. We propose that that information be captured for future use.

7 Conclusion

We have described our experiences in applying a category-theory-based approach to industrial strength, automated software specification, synthesis, and maintenance. It supports the capture and structuring of formal requirement specifications, design specifications, implementation software, and the refinement processes that lead from requirements to software. In this approach, the refinement process can guarantee correctness of the generated software. By recording, modifying, and then replaying the refinement history, we are able to more easily maintain the software. By capturing, abstracting, and structuring knowledge in a modular fashion, we are able to more easily reuse this knowledge for other applications. Our overall impression is that the technology does work for industrial strength applications, but that it needs additional work to make it more usable. It is interesting to note that when this technology is applied to software systems whose outputs are designs for airplane parts, the design rationale that is captured is not only software engineering design rationale, but also design rationale from other, more traditional, engineering disciplines, such as mechanical, material, and manufacturing engineering. This suggests the technology provides an approach to general systems engineering that enables one to structure and reuse engineering knowledge in the broadest sense.

References

1. D. Bjorner and C. Jones. *Formal Specification and Software Development*. Prentice-Hall International, 1982.
2. L. Blaine and A. Goldberg. Dtre - a semi-automatic transformation system. In B. Moller, editor, *Constructing Programs from Specifications*. North-Holland, 1991.
3. T. Gruber et al. An ontology for engineering mathematics. In *Proceedings of the Fourth International Conference on Principles of Representation and Reasoning*. Morgan Kaufman, 1994.
4. J. A. Goguen and R. M. Burstall. Some fundamental algebraic tools for the semantics of computation—part 1: Comma categories, colimits, signatures and theories. *Theoretical Computer Science*, 31(1,2):175–209, 1984.
5. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
6. R. Jullig and Y. V. Srinivas. Diagrams for software synthesis. In *Proceedings of KBSE '93: The Eighth Knowledge-Based Software Engineering Conference*, pages 10–19. IEEE Computer Society Press, 1993.
7. J. Meseguer. General logics. In *Logic Colloquium '87*, pages 275–329. Science Publishers B. V. (North-Holland), 1987.
8. B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
9. D. Smith. Kids: A knowledge based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*. MIT Press, 1991.
10. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
11. Y. V. Srinivas and R. Jullig. **specwaretm**: Formal support for composing software. In *Proceedings of the Conference of Mathematics of Program Construction*, 1995.

12. T. C. Wang and A. Goldberg. A mechanical verifier for supporting the design of reliable reactive systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1991.
13. K. Williamson and M. Healy. Formally specifying engineering design rationale. In *Proceedings of the Automated Software Engineering Conference-1997*, 1997.
14. K. Williamson and M. Healy. Deriving engineering software from requirements. *Journal of Intelligent Manufacturing*, 11(1):3–28, 2000.

A New Logic for Electronic Commerce Protocols^{*}

Kamel Adi, Mourad Debbabi, and Mohamed Mejri

LSFM Research Group, Computer Science Department, Laval University, Quebec,
Canada.

{adi,debbabi,mejri}@ift.ulaval.ca

Abstract. The primary objective of this paper is to present the definition of a new dynamic, linear and modal logic for security protocols. The logic is compact, expressive and formal. It allows the specification of classical security properties (authentication, secrecy and integrity) and also electronic commerce properties (non-repudiation, anonymity, good atomicity, money atomicity, certified delivery, etc.). The logic constructs are interpreted over a trace-based model. Traces reflect valid protocol executions in the presence of a malicious smart intruder. The logic is endowed with a tableau-based proof system that leads to a modular denotational semantics.

1 Motivations and Background

Security is becoming an increasingly important issue in computing, due to the incredible expansion of concurrent and distributed systems such as databases, world wide web, electronic mail and commerce, etc. In such contexts, information must be protected against mystification, destruction and disclosure. Accordingly, a great deal of interest has been expressed in the development and use of cryptographic protocols.

Financial institutions that use electronic commerce need to enforce special and additional security properties to warranty the validity of transactions. For instance, electronic commerce protocols have to ensure that there is no money creation or destruction (money atomicity) that could be engendered by transactions. Both the customer and the merchant must be able to exhibit a proof of the content of the delivered goods (certified delivery). Some people must be allowed to keep secret their identity (anonymity). The protocol must provide irrefutable evidence that makes impossible to a principal to deny its participation in a given transaction (non-repudiation).

Plenty of security protocols have been advanced to enforce the previously mentioned security properties. However, it is judicious to notice that several

^{*} This research is supported by a research grant from the National and Science Engineering Council, NSERC, and the Fonds pour la formation de Chercheurs et l'Aide à la Recherche, FCAR.

protocols have been shown flawed in computer security literature many years after their publication and use. This exemplifies the need for powerful and practical formal methods for the design and verification of these protocols.

The main intent of this work is to present the definition of a new logic for security protocols. The logic is compact, expressive and formal. It allows the specification of classical security properties (authentication, secrecy and integrity) and also electronic commerce properties (non-repudiation, anonymity, good atomicity, money atomicity, certified delivery, etc.). The logic constructs are interpreted over a trace-based model. Traces reflect valid protocol executions in the presence of a malicious smart intruder. Also, we present a tableau-based proof system for our logic with the aim to perform local model-checking. The tableau proof system is proven to be finite, sound and complete.

2 Related Work

The state of the art in terms of methods for the design and analysis of cryptographic protocols could be classified as follows: logical methods, general purpose formal methods and process algebra methods.

Typically, logical methods demonstrated that they are well suited to cope with the analysis of security protocols. Basically, these methods rest on the use of modal logics. The logic is used to specify the protocol (idealization) as well as the security properties. In 1989, Burrows, Abadi and Needham devised BAN, a modal logic of belief for the specification and verification of cryptographic protocols [4]. Simple and interestingly enough, BAN becomes the most known and famous logic dedicated to cryptographic protocols. Since then, plenty of derived logics have been advanced. In 1990, Bieber [2] developed CKT5, a modal logic of knowledge that has been revised and extended by Carlsen in [5] and Snekenes in [12]. Concurrently, many other logics attempted to combine several aspects of modal logics such as belief, knowledge and trust. These methods have been successfully used to detect many flaws in cryptographic protocols and they are very expressive while specifying security properties. Nevertheless, they are not very suitable to specify the protocols themselves. In fact, the protocols are often translated into a set of logical formulas. The translation process, often referred to as idealization, is error-prone since it is not systematic.

Another trend in formal cryptographic development consists in the accommodation of some well-known general-purpose formal methods. Representative specification languages that have been used in such accommodations are LOTOS [14], B, VDM [13], HOL [12], Ina Jo [8], Z [11] and Coq [3]. Although these formal methods are now firmly established and known to be of great use in specification and verification, it remains that they are not dedicated to cryptographic protocols.

Lately, the use of process algebra for cryptographic protocol specification and verification has been explored. In 1995, Gavin Lowe [9] was the first to use CSP [6] and model-checking techniques for cryptographic protocol analysis. Encouraged by the success in using CSP together with the model-checker FDR to

verify cryptoprotocols, John C. Mitchell, Mark Mitchell, Vitaly Shmatikov and Ulrich Stern [7] have used Murphy to analyze cryptographic protocols. These approaches rest on the generation of a set of valid execution traces. Afterwards, these traces are checked against a security property. The latter are expressed either as invariants on some event traces or as equivalences over processes expressed in a calculus accommodated to cope with cryptographic notions. For instance, Abadi and Gordon [1] have recently advanced Spi, a calculus for cryptographic protocols. The process algebra-based methods have been successfully used in the detection of several flaws in well-known cryptographic protocols. However, it is well known that the underlying verification techniques, mainly those based on model-checking, are problematic in the presence of processes that exhibit infinite behaviors.

3 Protocol Specification

Our concern in this section is to give a simple and intuitive representation of protocols that takes into account all the information needed to perform their analysis. First, we need to define the message syntax. It is captured by the following BNF grammar:

$m ::= A$	Principal Identifier
cte	Constant Message
$\{m\}_{m'}$	Encrypted Message
m, m'	Message Catenation
$f(m)$	Function Application

A constant message could be a simple text or a number. For instance, in electronic commerce protocols, e-good descriptions are simple texts. All non-constant messages will be represented as terms through functional application. For example, a fresh nonce N , created by a principal A in a protocol run α , is represented by the term $N(A, \alpha)$ and a client credit card number could be represented by the term $cc(A, B)$, where A is the client and B is the bank. For the sake of convenience, messages may be annotated. We use superscript annotations to indicate run identifiers and subscript annotations to indicate the principal association. Accordingly, the term $N(A, \alpha)$ could be merely represented by N_a^α . We classify protocol actions in two categories: external and internal actions, as follows:

- External actions or communication actions are made of send and receive actions. We will write $\alpha.i \ A \triangleright B : m$ to denote a send operation in which the principal A transmits the message m over the network, intentionally to the principal B , at the step i of the session α . We will write $\alpha.i \ A \triangleleft B : m$ to denote a receive operation in which the principal A gets the message m from the network, intentionally from the principal B , at the step i of the session α . The set of all external actions (send and receive actions) will be denoted by $\mathcal{E}Act$.
- Internal actions denote computation steps. We will write $\alpha.i \ act(m_1, \dots, m_n)$ to denote a computation step performed at the step i of the session α . It consists

of the invocation of the function *act* over the parameters $m_1, \dots, m_n$. The set of all internal actions will be denoted by $\mathcal{I}Act$.

Actions are abstracted to their names and their parameters. The set of all external and internal actions will be designated by $\mathcal{A}ct$. Actually, we have:

$$\mathcal{A}ct = \mathcal{E}Act \cup \mathcal{I}Act$$

To cope with electronic commerce protocols, we need to make the internal actions part of the protocol specification. Suppose, for example, that we have a protocol that allows a principal C (customer) to purchase goods from a vendor M (merchant) that will be paid through a trusted bank B . Somewhere in the protocol, the bank debits the customer account and credits the vendor account throw the internal action $debit(B, C, x)$ and $credit(B, M, x)$. Here is the formal definition of a protocol specification.

Definition 1 (Protocol). *A protocol is a finite sequence of statements of the form:*

$$\begin{array}{l} \left(\begin{array}{l} \alpha.i \text{ Act}_1(m_1^1, \dots, m_p^1) \\ \dots \\ \alpha.i \text{ Act}_q(m_1^q, \dots, m_r^q) \end{array} \right) \\ \alpha.i \ A \triangleright B : m \\ \alpha.i \ B \triangleleft A : m \\ \left(\begin{array}{l} \alpha.i \text{ Act}'_1(n_1^1, \dots, n_{p'}^1) \\ \dots \\ \alpha.i \text{ Act}'_l(n_1^q, \dots, n_{r'}^q) \end{array} \right) \end{array}$$

which state that, at a protocol step i , the actions

$$Act_1(m_1^1, \dots, m_p^1), \dots, Act_q(m_1^q, \dots, m_r^q)$$

are performed after which the message m is transmitted from agent A to agent B . Once the message m is received, the internal actions

$$Act'_1(n_1^1, \dots, n_{p'}^1), \dots, Act'_l(n_1^q, \dots, n_{r'}^q)$$

will be executed. The terms m_i^j and n_i^j are messages.

For the sake of convenience and in order to be compliant with the standard notation used in cryptoprotocol literature, we will write $\alpha.i \ A \rightarrow B : m$ as an abbreviation for $\alpha.i \ A \triangleright B : m$ followed by $\alpha.i \ B \triangleleft A : m$.

4 Trace-Based Model

We will adopt a trace-based model for our logic. The motivation underlying such a choice is three-fold: First, traces constitute the most simple and natural execution model considered so far in the security literature. Second, a security flaw in a cryptoprotocol is usually represented as a trace that violates a security

property. Third, there are many automatic trace generators that have been advanced in the literature. Hence, it is natural to check a security property against a trace that reflects a valid run of the protocol. A trace is said valid if all the messages sent by the intruder could be derived from its cumulated knowledge (initial knowledge and received messages) and all the involved principals respect the protocol. We assume that all the messages, sent or received by honest principals, transit by the intruder. This is to capture the fact that the intruder is aware of any message circulating over the network. Traces reflect a finite number of parallel and sequential protocol runs where concurrent events are interleaved to form a simple sequence of actions. We have no restrictions on traces in the sense that we support multi-session (an agent could participate in many sessions) and multi-role (an agent can be an initiator in some sessions and a responder in others). Let a be an action in $\mathcal{Act}$, we define the domain of traces denoted by $\mathcal{T}$, by the following grammar:

$$t ::= \epsilon \mid a.t$$

where ϵ stands for the empty trace and $a.t$ for a trace whose first action is a . The initial knowledge of each agent occurring in the trace is made explicit and represented by special communication steps called “magic” steps. We arrange to deliver in a magic way to each agent its initial knowledge thanks to these steps. We denote such steps by terms of the form $(\alpha.0, A \triangleleft \_ : m)$ where the sender is abstracted to the “ $\_$ ” symbol and “0” as a step identifier. This encoding does not affect the expressiveness of the specification (at the protocol or property levels). The main advantage of this representation is to make compact and uniform the logic constructs. The next section introduces the syntactic definition of our logic.

5 Logic Syntax

Most of the logics that are used to specify and analyze security protocols (BAN, CKT5, GNY, etc.) fail to capture in a natural and easy way e-commerce properties such as good atomicity, money atomicity, certificate delivery, etc. Actually, the specification of these properties goes beyond belief and knowledge manipulation. Furthermore, these logics could not analyze traces that explicitly make reference to the intruder.

In order to be able to express security properties, the used logic must be endowed with several characteristics. The authentication property, for example, requires, that a message received by a regular principal A has to be previously sent by a regular principal B in the same protocol run and if not, then there is an authentication flaw. This property suggests that the logic should handle the temporal aspects. Another example is the property of money atomicity in an e-commerce protocol. This property stipulates that there is no money creation or destruction induced by *Debit/Credit* transactions. This property requires the logic to be able to express assertions of the form: if an action *Debit* exists in the trace, then an action *Credit* must also exist within the same trace. Moreover, if we remove these actions from the trace, the property still has to remain true on

the resulting sub-trace. This suggests that the logic must be based on a dynamic model. The semantic interpretation of the formulas may have then to modify the model. The logic also has to be endowed with a recursion mechanism in order to state assertions such as “somewhere something may/must happen”. This is achieved by making fixpoint operators part of the logic syntax. Indeed, our logic is propositional, modal, dynamic and enriched with fixpoints. The modalities (possibility and necessity) are parameterized by trace patterns. This new logic capture classical and e-commerce security properties in a uniform and easy way.

In the sequel, we define the notion of a pattern. A trace pattern is a sequence of actions and action variables. They are used to specify trace properties. For example, the pattern $p = x.(\_i \text{ credit}(B, M, x_0)).y$ could be used to verify if somewhere in the analyzed trace t , the action $\_i \text{ credit}(B, M, x_0)$ (at the i^{th} step of a the protocol the bank B has credited the account of the merchant M by an amount x_0) has been performed. Patterns are a natural way to specify temporal properties. The first action of the pattern has to match the first action of the analyzed trace and so on. The pattern $p = a.x$ specifies that the first action of the trace must be a . The past and the present are captured by the action ordering induced by the pattern. For example, the pattern $p = x.a.y.b.z$ specifies that the action a has to be followed, somewhere in the trace, by the action b . This notion of time is consolidated with the possibility and necessity modalities.

The modal μ -calculus is an adequate logic to specify temporal properties such as (always, never, eventually, etc.). Nevertheless, though its high expressiveness, it fails to capture many important security properties. Actually, many electronic commerce protocol properties such good atomicity, money atomicity, involves irregular expressions over traces. As an example, consider the following property: the occurrence number of the action a (Debit) in the trace t is equal to the occurrence number of the action b (Credit) in the same trace. Such a property is clearly not expressible in the modal μ -calculus. By allowing trace variables to occur in patterns, we give a dynamic aspect to the logic. Actually, the semantic interpretation of formulas is made dependent on the values assigned to action variables. To illustrate this feature, let us take a simple example. Suppose that we want to specify that in a given trace, each action a is followed by an action b and the number of b actions in this trace is greater or equal to the number of a actions. This property could be simply specified by the following formula:

$$\nu X.(\langle x.a.y \rangle \# \rightarrow \langle x.a.z.b.t \rangle X)$$

This formula states that if the action a is present in the trace t , then it must be followed by the action b and if we remove these two actions (considered as resources with a limited number of use) from the trace, the reduced trace still satisfies the formula. Furthermore, to be able to express some security properties, we decided to add a “linearity” feature to the logic. For this, we added the connector $!$ that gives another dimension to the logic. In fact, actions could be considered as resources with a limited or unlimited number of use as in linear logic. For example in the formula $[p]\Phi$, all the constant actions specified in the

pattern p are considered as resources with a limited use if they are not preceded by the connector $!$. So, those actions could not be used another time when we verify the sub-formulas of Φ . However, all constant actions preceded by the connector $!$ are considered as resources with an unlimited number of use.

Let a be an action from $\mathcal{Act}$, and x be a pattern variable, the syntax of patterns is as follows:

$$p := a.p \mid !a.p \mid x.p \mid \epsilon$$

where ϵ stands for the empty pattern. The set of all patterns will be denoted by $\mathcal{P}$. We define two operators on patterns: the simplification pattern operator $\_^- : \mathcal{P} \rightarrow \mathcal{P}$ and the increase pattern operator $\_+ : \mathcal{P} \rightarrow \mathcal{P}$ as follows:

$$\begin{array}{l|l} \epsilon^- = \epsilon & \epsilon^+ = x \\ (a.p)^- = p^- & (a.p)^+ = x.a.p^+ \\ (!a.p)^- = !a.p^- & (!a.p)^+ = x.!a.p^+ \\ (x.p)^- = x.p^- & (x.p)^+ = x.p^+ \end{array}$$

The $\_^-$ operator is used to remove constant actions from patterns whereas the $\_+$ operator is used to introduce new pattern variables in pattern expressions. For example, $(x.a.y.b.z)^- = (x.y.z)$ and $(a.b)^+ = (x.a.y.b.z)$ where a and b are constant actions and x, y and z are pattern variables. Notice that the $\_^-$ operator does not remove constant actions preceded by the linear operator $!$ since these actions are considered as infinite resources. The syntax of formulas is presented in Table 1.

Table 1. The Logic Syntax.

$$\phi ::= X \mid \neg\phi \mid [p]\phi \mid \phi_1 \wedge \phi_2 \mid \nu X.\phi$$

6 Denotational Semantics

The semantic interpretation of formulas is achieved by the function $\llbracket \_ \rrbracket_e^-$ that takes as parameters a formula Φ , a trace t , a substitution σ and an environment e . It returns a value $\llbracket \Phi \rrbracket_e^{t,\sigma}$ that is a subset of $t_\downarrow$, where $t_\downarrow$ is defined as follows:

- (i) $t \in t_\downarrow$
- (ii) $t_1.a.t_2 \in t_\downarrow \Rightarrow t_1.t_2 \in t_\downarrow$

We will write $e[U \mapsto X]$, where U is a subset of $t_\downarrow$ and X is a variable formula, to denote the environment that is like e except that X is mapped to U . The interpretation of a formula Φ , written $\llbracket \Phi \rrbracket_e^{t,\sigma}$ is inductively defined on the structure of Φ and it is given in Table 2. Intuitively, a trace t satisfies the formula $[p]\Phi$, if for each substitution σ such that $p\sigma = t$, the trace $p^-\sigma$ satisfies the formula $\Phi\sigma$. For

example, let $\Phi_0 = [x.a.y][x.b.z]t = [p]\Phi$, where, $p = x.a.y$ and $\Phi = [x.b.z]t$. This formula is satisfied by a trace t , if for each substitution σ such that $t = (x.a.y)\sigma$, then the trace $(x.y)\sigma$ satisfies $([x.b.z]t)\sigma$.

Table 2. The Denotational Semantics of the Logic.

$\llbracket X \rrbracket_e^{t,\sigma} = e(X)$
$\llbracket \neg \Phi \rrbracket_e^{t,\sigma} = t_\downarrow - \llbracket \Phi \rrbracket_e^{t,\sigma}$
$\llbracket \Phi_1 \wedge \Phi_2 \rrbracket_e^{t,\sigma} = \llbracket \Phi_1 \rrbracket_e^{t,\sigma} \cap \llbracket \Phi_2 \rrbracket_e^{t,\sigma}$
$\llbracket [p]\Phi \rrbracket_e^{t,\sigma} = \{t \in t_\downarrow \mid \forall \sigma' \in \{\sigma'' \mid p\sigma\sigma'' = t\} . p^- \sigma\sigma' \in \llbracket \Phi \rrbracket_e^{t,\sigma' \circ \sigma}\}$
$\llbracket \nu X.\Phi \rrbracket_e^{t,\sigma} = \nu\varphi$ where $\varphi : U \mapsto \llbracket \Phi \rrbracket_{e[U \mapsto X]}^{t,\sigma}$

7 The Tableau-Based Proof System

The denotational semantics proposed in the previous section is based on a global model-checking paradigm. In such a paradigm, if we want to check whether a trace t satisfies or not a given formula Φ , we need to find all the traces in $t_\downarrow$ satisfying Φ and then verify if t belongs to this set or not. However, the local model-checking paradigm allows to verify if a given trace satisfies or not a formula without dealing with all the traces of $t_\downarrow$.

In this section, we present a local model-checking approach based on a tableau proof system to verify a formula specified in our logic. First, we present some notation used to define the tableau-based proof system. Let b and b' in $\{\epsilon, \neg\}$, and let Φ be a formula, we introduce bb' , $b \times b'$ and $b\Phi$ as follows:

$\epsilon\epsilon = \epsilon$	$\epsilon \times \epsilon = \epsilon$	$b = \epsilon \Rightarrow b\Phi = \Phi$
$\epsilon\neg = \neg$	$\epsilon \times \neg = \neg$	$b = \neg \Rightarrow b\Phi = \neg\Phi$
$\neg\epsilon = \neg$	$\neg \times \epsilon = \neg$	$b = b' \Rightarrow b\Phi = b'\Phi$
$\neg\neg = \epsilon$	$\neg \times \neg = \neg$	

Table 3 shows the tableau-based proof system. The idea behind the tableau rules is to capture in a deductive way when a trace t satisfies a formula Φ in a model $M = \langle t, \sigma \rangle$ under some environment e i.e. $t \in \llbracket \Phi \rrbracket_e^M$. In that case, the sequent $H, b, e, \sigma \vdash t \in \Phi'$ has a successful tableau i.e. all the leaves of the tableau (or the derivation tree generated by the tableau-based proof system) are successful, where $b\Phi' = \Phi$ and H is a set of hypotheses of the form $[X \mapsto S]$ associating a variable X to a set of traces S . A sequent θ has a successful tableau if there exists a finite tableau having θ as a root and all its leaves are successful. A leaf θ is successful when it meets one of the following conditions:

Table 3. The Tableau-Based Proof System.

$R_{\neg}$	$\frac{H, b, e, \sigma \vdash t \in \neg\Phi}{H, \neg b, e, \sigma \vdash t \in \Phi}$
$R_{\wedge}$	$\frac{H, b, e, \sigma \vdash t \in (\Phi_1 \wedge \Phi_2)}{H, b_1, e, \sigma \vdash t \in \Phi_1 \quad H, b_2, e, \sigma \vdash t \in \Phi_2} C_1$
R_{ν}	$\frac{H, b, e, \sigma \vdash t \in \nu X.\Phi}{H_X \dagger [X \mapsto H(X) \cup \{t\}], b, e, \sigma \vdash t \in \Phi[\nu X.\Phi/X]} t \notin H(X)$
$R_{\Box}$	$\frac{H, b, e, \sigma \vdash t \in [p]\Phi}{H, b_1, e, \sigma_1 \circ \sigma \vdash p^- \sigma \sigma_1 \in \Phi \quad \dots \quad H, b_n, e, \sigma_n \circ \sigma \vdash p^- \sigma \sigma_n \in \Phi} C_2$
$C_1 = \{b_1, b_2\} \in \{\{b'_1, b'_2\} \mid b'_1 \times b'_2 = b\}$ $C_2 = \left(\begin{array}{c} \{\sigma_1, \dots, \sigma_n\} = \{\sigma' \mid p\sigma\sigma' = t\sigma\sigma'\} \neq \emptyset \\ \text{and} \\ \{b_1, \dots, b_n\} \in \{\{b'_1, \dots, b'_n\} \mid b'_1 \times \dots \times b'_n = b\}, n > 0 \end{array} \right)$	

- $\theta = H, e, \sigma \vdash t \in X$ and $t \in e(X)$.
- $\theta = H, \neg, e, \sigma \vdash t \in X$ and $t \notin e(X)$.
- $\theta = H, e, \sigma \vdash t \in \nu X.\Phi$ and $t \in H(X)$.
- $\theta = H, e, \sigma \vdash t \in [p]\Phi$ and $\{\sigma' \mid p\sigma\sigma' = t\sigma\sigma'\} = \emptyset$.

It follows that an unsuccessful leaf θ , will denotes sequents that are not successful leaves and no rule of the tableau system could be applied on them.

Unlike common inference systems, the inference rules in tableau-based proof systems are used in a top-down fashion. The intuitive meaning of the rules of our tableau system is:

- The rule $R_{\neg}$: To verify if a trace t satisfies, or not, a formula $\neg\Phi$, it is sufficient to verify if t satisfies, or not, the formula Φ and, then, decide for the original formula. For that reason, we use the flag b in which we store the negations (to remember if we are dealing with the formula Φ or the formula $\neg\Phi$). Also, from this rule, it is immediate that verifying if a trace t satisfies, or not a formula $\neg\neg\Phi$ is equivalent to verify if t satisfies, or not, the formula Φ , since $\neg\neg b = b$ and:

$$(R_{\neg}) \quad \frac{H, b, e, \sigma \vdash t \in \neg\neg\Phi}{(R_{\neg}) \quad \frac{H, \neg\neg b, e, \sigma \vdash t \in \neg\Phi}{H, \neg\neg b, e, \sigma \vdash t \in \Phi}}$$

By using the flag b of the sequent, to treat the $\neg$ connector, we have simplified considerably the number of rules in our tableau-based proof system (comparatively to others defined in the literature).

- The rule $R_{\wedge}$: According to this rule, the sequent $H, b, e, \sigma \vdash t \in \Phi_1 \wedge \Phi_2$ has a successful tableau, if and only if the sequent $H, b_1, e, \sigma \vdash t \in \Phi_1$ and the sequent $H, b_2, e, \sigma \vdash t \in \Phi_2$ have successful tableaux for some b_1 and b_2 such that $b = b_1 \times b_2$.

- The rule R_ν : According to this rule, the sequent $H, b, e, \sigma \vdash t \in \nu X.\Phi$ has successful tableau, if the sequent $H_X \dagger [X \mapsto H(X) \cup \{t\}]$, $b, \sigma \vdash t \in \Phi[\nu X.\Phi/X]$ has successful tableau where H_X is $H \setminus H(X)$ and the trace t belongs to the set of hypotheses. Moreover, the side condition $t \notin H(X)$ must hold.

- The rule $R_\square$: This rule means that if there exists $\{\sigma_1, \dots, \sigma_n\} = \{\sigma' \mid p\sigma\sigma' = t\sigma\sigma'\} \neq \emptyset$ and for $i = 1 \dots n$, $p^- \sigma\sigma_n$ satisfies Φ , then t satisfies Φ .

We have established the following three major results whose proofs are reported in the extended version of this paper.

Theorem 1 (Finiteness). *For any sequent $\theta_1 = H_1, b_1, \sigma_1 \vdash t_1 \in \Phi_1$, such that t_1 is a finite trace, then there is a maximum height tableau with root θ_1 .*

Theorem 2 (Soundness). *If $\emptyset, b, e, \sigma \vdash t \in \Phi$ has a successful tableau then $t \in \llbracket b\Phi \rrbracket_e^\sigma$.*

Theorem 3 (Completeness). *If $t \in \llbracket b\Phi \rrbracket_e^\sigma$ then $\emptyset, b, \sigma \vdash t \in \Phi$ has a successful tableau.*

8 Security Properties

To exemplify the expressiveness of our logic, we present hereafter the formalization of some important security properties.

8.1 Authentication

In [10], Gavin Lowe discusses many authentication definitions proposed in the literature. He pinpoints their weaknesses and he concludes by proposing a hierarchy of definitions. The strongest one is: Whenever an agent A completes a run of the protocol, apparently with B , then B has recently been running the protocol, apparently with A , and the two agents agree upon who initiated the run, and agree upon all data values used in the run; moreover, there is a one-one relationship between the runs of A and the runs of B .

We adopt here an amended version of this definition by taking into account those dataflow constraints on messages induced by communications i.e. a message received by a regular principal A has to be previously sent by a regular principal B in the same protocol run. To formalize this definition, we need the following auxiliary notation:

Let S be a set of pairs (principal identity, session variable), P a protocol and L a set of natural numbers. We define P_L^S as follows:

$$\begin{aligned} \epsilon_L^S &= \epsilon \\ (a.P)_L^S &= \begin{cases} (x.i \ A \triangleright B : m).(P)_L^S & \text{if } i \in L, (A, x) \in S \text{ and } a = (\alpha.i \ A \triangleright B : m) \\ (x.i \ A \triangleleft B : m).(P)_L^S & \text{if } i \in L, (A, x) \in S \text{ and } a = (\alpha.i \ A \triangleleft B : m) \\ P_L^S & \text{Otherwise} \end{cases} \end{aligned}$$

The operation P_L^S aims to extract from the protocol P those communication steps whose step numbers are in L and for which there exists an element (A, x) in S such that A participates in these steps. The session identifier of the protocol steps in which A participates should be renamed by x . This operator achieves a projection of the protocol over some steps in L and some principals in S up to renaming of the session identifiers. For example, let P be the following protocol:

$$\left\{ \begin{array}{l} \alpha.1 A \rightarrow B : m_1 \\ \alpha.2 B \rightarrow A : m_2 \\ \alpha.3 A \rightarrow B : m_3 \end{array} \right. \equiv \left\{ \begin{array}{l} \alpha.1 A \triangleright B : m_1 \\ \alpha.1 B \triangleleft A : m_1 \\ \alpha.2 B \triangleright A : m_2 \\ \alpha.2 A \triangleleft B : m_2 \\ \alpha.3 A \triangleright B : m_3 \\ \alpha.3 B \triangleleft A : m_3 \end{array} \right.$$

Now, let L be the set $\{1, 2\}$ and S be the set $\{(A, x), (B, y)\}$. Therefore, P_L^S is:

$$\begin{array}{l} x.1 A \triangleright B : m_1 \\ y.1 B \triangleleft A : m_1 \\ y.2 B \triangleright A : m_2 \\ x.2 A \triangleleft B : m_1 \end{array}$$

We say that A proves its identity to B in the steps reported in L (the set L could be used for a partial authentication.), and we write $\mathcal{A}_B^A(L)$, if: whenever B runs the L -steps in a session y , there exists a session x in which A runs the L -steps and the L -steps of the session x are a “forward” of the L -steps of the session y i.e. what has been sent (resp. received) in the session x by A has been received (resp. sent) in the session y by the principal B . Moreover, the property remains true if we remove from the model the steps $P_L^{\{(A,x),(B,y)\}}$. The specification of the authentication property in our logic is reported in Table 4. Let us take a concrete example. Let P be the simplified version of the Needham-Shroeder protocol:

$$\begin{array}{l} \alpha.1 A \rightarrow B : \{N_a\}_{K_{ab}} \\ \alpha.2 B \rightarrow A : \{N_a + 1\}_{K_{ab}} \end{array}$$

The objective of this protocol is to achieve mutual authentication between two principals A and B . In other words, the intention is to convince the principal playing role A that B is operational. This protocol could be attacked as follows:

$$t = \left\{ \begin{array}{l} \beta.1 A \rightarrow I(B) : \{N_a\}_{K_{ab}} \\ \gamma.1 I(B) \rightarrow A : \{N_a\}_{K_{ab}} \\ \gamma.2 A \rightarrow I(B) : \{N_a + 1\}_{K_{ab}} \\ \beta.2 I(B) \rightarrow A : \{N_a + 1\}_{K_{ab}} \end{array} \right. \equiv \left\{ \begin{array}{l} \beta.1 A \triangleright B : \{N_a\}_{K_{ab}}. \\ \gamma.1 A \triangleleft B : \{N_a\}_{K_{ab}}. \\ \gamma.2 A \triangleright B : \{N_a + 1\}_{K_{ab}}. \\ \beta.2 A \triangleleft B : \{N_a + 1\}_{K_{ab}}. \end{array} \right.$$

Now let us see how this flaw could be discovered using our logic. The authentication property is in our case:

$$\mathcal{A}_A^B(\{1, 2\}) \equiv \nu X. (\langle P^{\{(A,y)\}}(\{1, 2\})^+ \rangle \# \rightarrow \langle P^{\{(B,x),(A,y)\}}(\{1, 2\})^+ \rangle. X)$$

Table 4. Formalizing Security Properties.

Authentication :	
$\mathcal{A}_B^A(L) \equiv \nu X.(\langle P^{\{(B,y)\}}(L)^+ \rangle \# \rightarrow \langle P^{\{(A,x),(B,y)\}}(L)^+ \rangle X)$	
Secrecy :	
$\mathcal{S}(m) \equiv \neg \mathcal{K}(m)$	
$\mathcal{K}(m) \equiv \langle x.(\_ \_ \_ I \triangleleft \_ : m).x' \rangle \# \vee \langle x.(\_ \_ \_ \triangleright \_ : m).x' \rangle \# \vee$	
$\mathcal{K}_{comp}(m) \vee \mathcal{K}_{dcomp}(m)$	
$\mathcal{K}_{comp}(m) \equiv \begin{cases} \mathcal{K}(m_1) \wedge \mathcal{K}(m_2) & \text{if } m = m_1, m_2 \text{ or } m = \{m_1\}_{m_2} \\ ff & \text{otherwise} \end{cases}$	
$\mathcal{K}_{dcomp}(m) \equiv \mathcal{K}(m.m') \vee \mathcal{K}(m'.m) \vee (\mathcal{K}(\{m\}_{m'}) \wedge \mathcal{K}(m'))$	
Money Atomicity :	
$\mathcal{MA}(\{a_1, \dots, a_n\}) \equiv \nu X.(\bigvee_{a \in \{a_1, \dots, a_n\}} \langle x.a.x' \rangle \# \rightarrow \bigvee_{p \in \pi(\{a_1, \dots, a_n\})} \langle p^+ \rangle X)$	
Good Atomicity :	
$\mathcal{GA}(\{a_1, \dots, a_n\}) \equiv \nu X.(\bigvee_{a \in \{a_1, \dots, a_n\}} \langle x.a.x' \rangle \# \rightarrow \bigvee_{p \in \pi(\{a_1, \dots, a_n\})} \langle p^+ \rangle X)$	

According to the definition of $P^{\{(A,y)\}}(\{1,2\})^+$ and $P^{\{(B,x),(A,y)\}}(\{1,2\})^+$, we have:

$$P^{\{(A,y)\}}(\{1,2\})^+ = \begin{cases} x_1. \\ (y.1 \ A \triangleright B : \{N_a\}_{K_{ab}}). \\ x_2. \\ (y.2 \ A \triangleleft B : \{N_a - 1\}_{K_{ab}}). \\ x_3 \end{cases}$$

$$P^{\{(A,x)\}}(\{1,2\})^+ = \begin{cases} y_1. \\ (x.1 \ A \triangleright B : \{N_a\}_{K_{ab}}). \\ y_2. \\ (x.1 \ B \triangleleft A : \{N_a\}_{K_{ab}}). \\ y_3. \\ (x.2 \ B \triangleright A : \{N_a - 1\}_{K_{ab}}). \\ y_4. \\ (x.2 \ A \triangleleft B : \{N_a - 1\}_{K_{ab}}). \\ y_5 \end{cases}$$

Notice that the authentication property is not satisfied by the trace t , since the pattern $P^{A,x}(\{1,2\})^+$ could be made equal to the trace t however the pattern $P^{A,x}(\{1,2\})^+$ could not.

8.2 Secrecy

Secrecy is the fact of keeping secret certain pieces of information. The formalization of this property in our logic is given in Table 4. Intuitively, a message m

is secret if it is not known by the intruder, i.e. $\neg\mathcal{K}(m)$. A message m is known by the intruder if:

- The message is in the intruder initial knowledge, i.e: $\langle x.(\_ \_ I \triangleleft \_ : m).x' \rangle \#$
- The intruder has received the message m , i.e: $\langle x.(\_ \_ \triangleright \_ : m).x' \rangle \#$
- The intruder can deduce the message using its initial knowledge, the received messages and its usual abilities i.e. $\mathcal{K}_{comp}(m) \vee \mathcal{K}_{dcomp}(m)$.

The intruder initial knowledge is expressed as virtual communication steps. Suppose for example that the intruder knows a key K_{is} , this knowledge will be expressed by a communication steps as follows: $\langle \_ \_ \rightarrow I : K_{is} \rangle$. This virtual communication step capturing the intruder initial knowledge must be added at the beginning of the analyzed trace.

8.3 Money Atomicity

Money atomicity in an electronic commerce protocol allows to guarantee the transfer of money without the possibility of destruction (sent but not received) or of creation (received but not sent). For example, consider the following protocol:

$$\begin{aligned}
 \alpha.1 \ C &\rightarrow M : \{CC\}_{K_b} \\
 \alpha.2 \ C &\rightarrow B : \{CC, \$V\}_{K_{cb}}, \\
 &\quad \alpha.2 \ Debit(B, CC, \$V) \\
 \alpha.3 \ M &\rightarrow B : \{CM, \{CC, \$V\}_{K_b}\}_{K_{mb}}, \\
 &\quad \alpha.3 \ Credit(B, CM, \$V)
 \end{aligned}$$

In this protocol, C is the client, M is the merchant, B is the bank, CC is the client account number, CM is the merchant account number. This protocol clearly is not money atomic because if the message 3 is not received by the bank then the client account will be debited without crediting the merchant account. A multiset of internal actions (involving money) written $\{a_1, \dots, a_n\}$ satisfies the money atomicity property and we write $\mathcal{MA}(\{a_1, \dots, a_n\})$, if the presence of one action a_i in the model entails the presence of all the other remaining actions. The term $\pi(\{a_1, \dots, a_n\})$ denotes all the permutations of the sequence $a_1, \dots, a_n$. The specification of money atomicity property in our logic is represented in Table 4. To clarify this property, let us consider a concrete example. Let t be the following trace:

$$t = \left\{ \begin{array}{l}
 \beta.1 \ C \triangleright M : \{CC\}_{K_b}. \\
 \gamma.1 \ M \triangleleft C : \{CC\}_{K_b}. \\
 \alpha.2 \ C \triangleright B : \{CC, \$V\}_{K_{cb}}. \\
 \gamma.2 \ B \triangleleft C : \{CC, \$V\}_{K_{cb}}. \\
 \quad \gamma.2 \ Debit(B, CC, \$V). \\
 \alpha.3 \ M \triangleright B : \{CM, \{CC, \$V\}_{K_b}\}_{K_{mb}}. \\
 \gamma.3 \ B \triangleleft M : \{CM, \{CC, \$V\}_{K_b}\}_{K_{mb}}. \\
 \quad \gamma.3 \ Credit(B, CM, \$V). \\
 \theta.2 \ B \triangleleft C : \{CC, \$V\}_{K_{cb}}. \\
 \quad \theta.2 \ Debit(B, CC, \$V)
 \end{array} \right.$$

The money atomicity property is:

$$\begin{aligned}
& \mathcal{MA}(\{(x_0.2 \text{ Debit}(B, x_1, \$V)), (y.3 \text{ Credit}(B, y_1, \$V))\}_t) \\
& \equiv \nu X. (\langle x.(x_0.2 \text{ Debit}(B, x_1, \$V)).x' \rangle \# \vee \langle x.(y.3 \text{ Credit}(B, y_1, \$V)).x' \rangle \# \rightarrow \\
& \quad \langle z_1.(x_0.2 \text{ Debit}(B, x_1, \$V)).z_2.(y.3 \text{ Credit}(B, y_1, \$V)) \rangle X \vee \\
& \quad \langle z_1.(y.3 \text{ Credit}(B, y_1, \$V)).z_2.(x_0.2 \text{ Debit}(B, x_1, \$V)).z_3 \rangle X)
\end{aligned}$$

Informally, this property states that the number of *Debit* actions must be equal to the number of the *Credit* actions in the trace t . Unlike the modal μ -calculus, our logic allows the verification of such properties that involve counting. Actually, this is achieved by the dynamic aspect of our model and the pattern algebra. Notice that the trace t does not satisfy the money atomicity property. In fact, if we remove one action *Debit* and one action *Credit* from this trace, then the rest of the trace will not satisfy the money atomicity formula.

8.4 Good Atomicity

Good atomicity in an electronic commerce protocol stands for an exact transfer of goods with respect of money i.e. the client receives the good if and only if the money is transferred. For example, consider the following protocol:

$$\begin{aligned}
& \alpha.1 \ C \rightarrow M : \{CC\}_{K_b} \\
& \alpha.2 \ M \rightarrow B : \{CM, \{CC, \$V\}_{K_b}\}_{K_{mb}}, \\
& \quad \alpha.2 \ \text{Debit}(CC, \$V) \\
& \quad \alpha.2 \ \text{Credit}(CM, \$V) \\
& \alpha.3 \ M \rightarrow C : \text{Data}
\end{aligned}$$

In this protocol *Data* corresponds to an electronic good that the consumer has paid. This protocol is not good-atomic because if the message 3 is not received by the consumer then he will not receive the good for which he was charged. The specification of the good atomicity property is reported in Table 4. Technically speaking, it is quite similar to the money atomicity property.

9 Conclusion

We have reported in this paper the definition of a new dynamic, linear and modal logic for security protocols. The logic is compact, expressive and formal. It allows the specification of classical security properties (authentication, secrecy and integrity) and also electronic commerce properties. We present the syntactic definition of the logic together with the underlying formal semantics. The logic constructs are interpreted over a trace-based model. Traces reflect valid protocol executions in the presence of a malicious smart intruder. The logic is endowed with a tableau-based proof system that naturally leads to a modular (local) relativized denotational semantics of the logic. The tableau proof system is proven to be finite, sound and complete. As a future work, we intend to combine this logic with a dedicated calculus for cryptographic protocols.

References

1. Martin Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*. ACM Press, April 1997.
2. P. Bieber. A Logic of Communication in a Hostile Environment. In *Proceedings of the Computer Security Foundations Workshop III*, pages 14–22. IEEE Computer Society Press, 1990.
3. Dominique Bolignano. An Approach to the Formal Verification of Cryptographic Protocols. In *Proceedings of the Third ACM Conference on Computer and Communications Security, CCS'96, New Delhi, India*, pages 106–118. ACM Press, 1996.
4. M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *Proceedings of the Royal Society of London A Vol.426*, pages 233–271, 1989.
5. U. Carlsen. *Formal Specification and Analysis of Cryptographic Protocols*. PhD thesis, Thèse d'Informatique soutenue à l'Université PARIS XI, October 1994.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. V. Shmatikov J. C. Mitchell and U. Stern. Finite-state analysis of SSL 3.0. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 201–216, Berkeley, January 26–29, 1998. Usenix Association.
8. R. A. Kemmerer. Using Formal Verification Techniques to Analyze Encryption Protocols. In *Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy*, pages 134–139. IEEE Computer Society Press, 1987.
9. Gavin Lowe. An Attack on the Needham-Schroeder Public Key Authentication Protocol. *Information Processing Letters*, 56(3):131–136, November 1995.
10. Gavin Lowe. Some New Attacks upon Security Protocols. In *Proceedings of the Computer Security Foundations Workshop VIII*. IEEE Computer Society Press, 1996.
11. Einar Snekkenes. Authentication in Open Systems. In *10th IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification*, pages 313–324, June 1990.
12. Einar Snekkenes. *Formal Specification and Analysis of Cryptographic Protocols*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, Norwegian Defence Research Establishment, P.O. Box 25, N-2007, Kjeller, Norway, January 1995.
13. The commission of the European Communities CEC DG-XIII. Security Investigation Final Report. Technical Report S2011/7000/D010 7000 1000, CEC, September 1993.
14. Vijay Varadharajan. Verification of Network Security Protocols. *Computers and Security*, 8, December 1989.

Extended Institutions for Testing

Marielle Doche^{1*} and Virginie Wiels²

¹ Department of Electronics and Computer Science
University of Southampton, Highfield
Southampton SO17 1BJ, United-Kingdom
`mfd@ecs.soton.ac.uk`

² ONERA-CERT/DTIM
BP 4025, 2 avenue E. Belin,
F-31055 Toulouse Cedex 4, France
`wiels@cert.fr`

Abstract. In this paper, we present an extension of the notion of institution that takes into account test cases. Our approach is to incrementally generate functional tests from a structured formal specification, we generate tests from small specifications and compose them. The issue is then to prove the correctness of the resulting tests. We thus extend the classical notion of institution with a category of test cases and a notion of satisfaction of test cases.

Keywords: Functional test generation, formal specification, category theory, institutions.

1 Introduction

Recently, formal methods have been used to generate specification-based tests, to detect if a system differs from its specification [13,4]. But these approaches are quickly limited by the size of the specifications.

So we propose an approach based on structured specifications and compositionality properties: firstly we generate tests from small specifications, then we compose them to obtain test for a cluster of specifications. With such a strategy, it is also possible to have an incremental testing approach, classical in the software engineering area: first we test in details the basic components (unit level), then we test interactions between components (integration level), and finally we test the behavior of a cluster taken as a whole (cluster level).

Theoretical frameworks exist to handle the question of the correctness of generated test for simple specifications [13], but such frameworks do not deal with composition issues. The notion of institution defined by Goguen and Burstall [15] allows to formalize the notions of specification, model, satisfaction of properties and takes into account composition issues. We propose to extend the definition of institution to include the notion of test. In this new framework, we are able to express the correctness requirements for simple specifications and to reason about their preservation after composition.

* This work was done while I was working at ONERA, Toulouse.

Section 2 describes the formal specification languages we are considering and particularly the way specifications can be composed. In section 3, we present our testing framework, and especially the correctness properties. In the fourth section we define the notion of extended institution to deal with test generation and correctness. Finally, in section 5, we explain how this approach has been applied in practice.

2 Formal Framework

In this section, we present the specification framework on which we base our work on test. We do not wish to give too many details about the exact syntax and semantics of our specifications, because it would only complexify the presentation of the test part which is our main contribution. We consider a more general framework which describes the properties that our specification language must verify: specifications should form a finitely cocomplete category and our formal language should be an institution.

2.1 Specification Framework

Specification. A specification (also called *presentation*) $Spec = (Sig, Ax)$ is composed of two parts: the signature Sig with the set of sorts S and the vocabulary Voc and a set of axioms Ax . S contains names of sorts. Voc is composed of several sets of symbols. For example, for a specification in typed first order logic, Voc is a set of functions and a set of predicates (see for instance [12]).

Ax is the set of axioms that describe the behavior of the modelled system as well as constraints on the environment. Axioms are well-formed formulas on Voc .

Specification Morphism. A *specification morphism* $m : A \rightarrow B$ from a specification A to a specification B maps any element of the signature of A to an element of the signature of B that is compatible (i.e. same nature and corresponding types).

We then define the image of an A -formula by m in a classical way. m must also be such that the image of any axiom of A is a theorem of B . In view of testing activity, we only consider morphisms that are injective (two different elements of the signature of A cannot have the same image by m), so the existence of an injective morphism m between A and B means that B is a refinement of A up to renaming introduced by m . In behavior terms, the constraint on the axioms means that the behavior of A is preserved in B .

The Category of Specifications. In the sequel, we assume that specifications and specification morphisms form a category. For example of such categories with algebraic language or first order language, see [15]; for examples with temporal logics, see [11,1].

Moreover, we suppose that this category is finitely cocomplete [23]: so we can build push-out.

In practice, we have defined the Moka language: we have considered specifications whose axioms are formula in the metric linear temporal logic TRIO [14,17]. We proved in [21] that these specifications with an adequate definition of specification morphism constitute a finitely cocomplete category.

Push-Out of Specifications. In order to compose specifications, we need an operation called *push-out*.

Given two specifications B , C , and a description of what they share (by means of a specification A and two morphisms f and g), the push-out builds the “smallest” specification D that contains B and C .

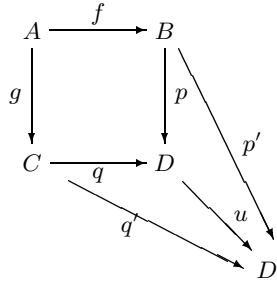


Fig. 1. Pushout

Formally, the pushout of two morphisms $f : A \rightarrow B$ and $g : A \rightarrow C$ is a specification D and two morphisms $p : B \rightarrow D$ and $q : C \rightarrow D$ such that $p \circ f = q \circ g$ and for all specification D' and morphisms p' and q' such that $p' \circ f = q' \circ g$, there exists a unique morphism $u : D \rightarrow D'$ such that $u \circ p = p'$ and $u \circ q = q'$. Notation: $D = B +_A C$

With this operation, we can describe the interaction of several small specifications of components of the system and then we incrementally build a specification which describes the whole system.

2.2 Institution

The notion of institution defined by Goguen and Burstall [15] formalizes the relationships between specifications, models and the notion of satisfaction of a property. The slogan for institutions is that “Truth is invariant under change of notation”.

In the following we give the definition of an institution. For more details and examples, see [15].

Definition. An *institution* $\mathcal{I}$ consists of:

1. a category **Sign**, whose object are called *signatures*;
2. a functor $Sen : \mathbf{Sign} \rightarrow \mathbf{Set}$, giving for each signature Σ a set whose elements are called sentences over the signature;
3. a functor $\mathbf{Mod} : \mathbf{Sign} \rightarrow \mathbf{Cat}^{\mathbf{op}}$, giving for each signature Σ a category whose objects are called Σ -models, and whose arrows are called Σ -models-morphisms (**Cat** is the category of categories, $\mathbf{Cat}^{\mathbf{op}}$ its opposite) and
4. a relation $\models_{\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times Sen(\Sigma)$ for each Σ in $|\mathbf{Sign}|$, called Σ -satisfaction,

such that for each morphism $\phi : \Sigma \rightarrow \Sigma'$ in **Sign** the *Satisfaction Condition*:

$$m' \models_{\Sigma'} Sen(\phi)(e) \Leftrightarrow \mathbf{Mod}(\phi)(m') \models_{\Sigma} e \quad (1)$$

holds for each $m' \in |\mathbf{Mod}(\Sigma')|$ and each $e \in Sen(\Sigma)$.

Examples. In [15], the authors give some examples of institutions, like many-sorted equational logic or first order logic. In [1], the authors prove that LTL and CTL are institutions.

We have proved, in [8], that our Moka specification language, with axioms expressed in the TRIO logic, is an institution.

3 Formal Testing

The aim of *functional* or *black-box* testing is to detect if a system does not satisfy an expected behavior. Lots of works [13,4] are based on formal specifications to generate functional tests. Indeed, each model of the specification should be an expected execution of the system, and so, it is a base to define a *test case*. Thus, lots of tools, based on several formal languages, have been defined to automatically generate test cases, for example by using model-checking algorithms [16] or semantic tableau method [6].

However, the generated tests must verify some properties to ensure a confidence in the correctness of the tested system with respect to the test selection process.

In this section we present some definitions and properties based on [13,2] to define a rigorous testing framework. Then we describe the limitations of these approaches.

3.1 Definitions and Properties to Ensure

In [3,13], the authors present a rigorous framework to formalize the quality of test sets with respect to selection criteria. This work is presented for algebraic specifications and proposed methodologies are strongly linked to this formal language.

In order to test object-oriented software, the authors of [2] adapt this framework to deal with specifications based on Petri nets and temporal logic. This adaptation accepts the use of different formalisms for specifications, test sets and programs insofar as a satisfaction relation can be defined between them.

In the sequel, we present another adaptation of this framework to our specification formalism.

Preliminary Definitions. We first give useful definitions and notations :

- In functional testing, the *test goal* is the description of an expected behavior of the system. Given a *formal specification* $Spec = (Sig, Ax)$, the test goal is the whole behavior described in this specification, indeed Ax .
- *Prog* is a *program* under test, which should implement $Spec$ and we suppose that *Prog* and $Spec$ share the same vocabulary. A program execution P is a complete sequence of instantiations of program data and can be considered as an interpretation structure on the language of *Prog*.

Test aim is to check whether all possible executions P of the program *Prog* satisfy the test goal Ax . If we suppose that P is defined on Sig , we say that " P satisfies Ax " and it is denoted $P \models_{Sig} Ax$.

In functional testing, this aim is achieved by executing a set of scenarios, or *test set*, on the program, which must detect as many errors as possible. An ideal test set is an exhaustive set of executions of the program. But this is rarely possible and we must choose criteria of selection to obtain a smaller set of scenarios: these criteria and the selected test set define the *test context*.

In the next paragraph, we give formal definitions of test sets and test contexts. Then we explicit the properties a test context must verify in order to adequately evaluate correctness of the program.

Test Sets and Test Contexts. A set of selected scenarios is called a *test set* T , each scenario is a *test case* τ . In a test case, we give for some items of $Spec$ vocabulary, a value for each parameter of the item and a value for the item: we call this an *instance* of an item. If the specification is based on a temporal logic, we must also define the world in which the item is evaluated. It is not necessary to give a value to each item of the vocabulary, but we keep in a test set only consistent test cases : there is no contradiction between two instances of a test case, which means that in a given test case, an item cannot have two different values for the same parameter values.

Thus, a test case τ represents a set of execution models.

For example, we define four states for an aircraft:

$\{cruise\_speed, landing, takeoff, parking\}$

Our test goal is that the aircraft is *cruise_speed* if and only if its altitude is above a threshold *alt_min*, which can be written in first order logic:

$$altitude \geq alt\_min \Leftrightarrow state = cruise\_speed$$

We can generate the following test cases to check this test goal :

$$\begin{aligned} \tau_1 &= \begin{pmatrix} alt\_min = 15 \\ altitude = 20 \\ state = val\_cs \\ cruise\_speed = val\_cs \end{pmatrix} & \tau_2 &= \begin{pmatrix} alt\_min = 15 \\ altitude = 10 \\ state = val\_t \\ takeoff = val\_t \end{pmatrix} \\ \tau_3(x) &= \begin{pmatrix} alt\_min = 15 \\ altitude = x \in 0..15 \\ state = val\_l \\ landing = val\_l \\ takeoff = val\_t \end{pmatrix} & \tau_4 &= \begin{pmatrix} alt\_min = 15 \\ altitude = 10 \\ state = val\_l \\ state = val\_t \\ landing = val\_l \\ takeoff = val\_t \end{pmatrix} \end{aligned}$$

$state = val\_l$ is an instance for the variable *state*. Obviously, the test case τ_4 is not consistent because the variable *state* takes two different values. $\{\tau_1, \tau_2, \tau_3\}$ form a possible test set.

H is the set of *test hypotheses*. An hypothesis describes assumptions about the system or the test process to reduce the size of the test set. An obvious hypothesis of the testing process is that a finite set of finite test cases is enough to test a system which has infinite running. Besides, test criteria, which define an abstract value of items in a test case, can be defined by the *uniformity hypotheses*: we define sub-domains of interpretation for items, such that the system has the same behavior for all the values of the sub-domain. Then we assume that testing for a value of a sub-domain is enough to test for all its values. For example, if we suppose that the value of *altitude* is an integer in the range 0..45, a uniformity hypothesis assumes that success of the test case $\tau_3(x)$ for a given value of x in 0..15 ensures success of the test case $\tau_3(x)$ for any other value of x in 0..15.

A test set is defined with respect to these test hypotheses: a *test context* is a couple (H, T) where T is a test set selected with respect to H .

In the sequel, we suppose that all test hypotheses can be formalized, examples of formalization are given in [3,2,8].

Oracle. To conclude on the success of a test activity, we must decide whether a program execution is correct with respect to the expected result. This is the *oracle problem*. Expected results are defined in each test case by some instances, thus the problem consists here in deciding whether a possible execution P of *Prog* satisfies the constraints of a test case τ .

If an execution P satisfies one test case τ of the test set T (all defined on the signature *Sig*), we say that " P satisfies the test case τ " and moreover we say that " P satisfies the test set T " and we notice: $P \prec_{Sig} \tau$ and $P \prec_{Sig} T$.

Correctness of a Test Context. Now we are interested in the efficiency of a test set to detect errors and hence to decide if the program is correct with respect to the test goal. So we present the properties a test context must verify, for a program $Prog$ and a test goal Ax , defined on the signature Sig .

(H, T) is unbiased: T does not reject correct programs: any execution of $Prog$ which satisfies the test hypotheses H and the test goal Ax , must satisfy the test set T .

$$\forall P \in Prog, (P \text{ satisfies } H \wedge P \models_{Sig} Ax) \Rightarrow P \prec_{Sig} T$$

(H, T) is valid: T accepts only correct programs: any execution of $Prog$ which satisfies the test hypotheses H and the test set T , must satisfy the test goal Ax .

$$\forall P \in Prog, (P \text{ satisfies } H \wedge P \prec_{Sig} T) \Rightarrow P \models_{Sig} Ax$$

(H, T) is correct: (H, T) is unbiased and valid: any execution of $Prog$ which satisfies the test hypotheses H , satisfies the test goal Ax if and only if it satisfies the test set T .

$$\forall P \in Prog, P \text{ satisfies } H \Rightarrow (P \models_{Sig} Ax \Leftrightarrow P \prec_{Sig} T)$$

3.2 Limitations

Formal methods offer a complete framework to assist functional testing activity: mechanisms and tools to generate tests from formal specifications and criteria to evaluate the correctness of these tools.

However, such strategies are limited to small specifications, and it is not possible to have an incremental approach of testing.

So, we propose an approach where test generation is based on formal structured specifications, as explained in the next section.

Remark. The approach assumes that the tested programs are structured and that their structuration reflects the structure of the specifications.

4 Test Generation from Structured Specifications

The specification framework presented in section 2 allows us to define structured specifications: starting from small ones and composing them. We would like to exploit this structuration to have incremental test generation, but we have to keep aware of the correctness criteria.

4.1 Intuition

Our objective is first to generate test cases from small specifications, then to rename them according to specification morphism, and finally to compose sets of test cases according to a push-out of specifications.

In the following, we give more details on these mechanisms: renaming and composition.

Renaming Given a specification morphism $f : A \rightarrow B$. We suppose we have generated a test context (H_A, T_A) from the specification A which is correct for the test goal Ax_A .

We want to obtain a test context (H_B, T_B) defined on the vocabulary of B (in order to test the interaction between two components expressed by f or to use the test context (H_B, T_B) as a basis for a complete test context for B).

Firstly, we rename each test case of T_A :

- each item which appears in a test case is defined in the vocabulary of A ;
- then we replace it by its image according to the specification morphism f ;

Thus we obtain a set of test cases T_B defined on the signature of B .

For example if the test case τ_1 has been generated from the specification A , and the morphism f adds a prefix $B\_$ to all the items of Sig_A , we can build the image τ_B :

$$\tau_B = \left(\begin{array}{l} B\_alt\_min = 15 \\ B\_altitude = 20 \\ B\_state = val\_cs \\ B\_cruise\_speed = val\_cs \end{array} \right)$$

Secondly, we build the image of the hypotheses H_A : they are formula and their images are classically defined according to the morphism f .

So we have built a test context (H_B, T_B) defined on the vocabulary of B . The new problem is to conclude on the correctness of this context with respect to the test goal $Sen(f)(Ax_A)$, indeed the image by f of the axioms of A .

Composition Given a push-out of specifications (cf figure 1), we suppose we have a test context (H_B, T_B) correct for Ax_B and another context (H_C, T_C) correct for Ax_C .

We wish to obtain a test context (H_D, T_D) defined on the vocabulary of D in order to test the whole system described by D .

We can rename the two contexts (H_B, T_B) and (H_C, T_C) according to the morphisms p and q respectively, to obtain the contexts (H'_B, T'_B) and (H'_C, T'_C) defined on the vocabulary of D .

The hypotheses H_D is the conjunction of both hypotheses H'_B and H'_C .

Then, we want to define T_D . We propose to construct a test case τ_D from each pair (τ_B, τ_C) composed of a test case of T'_B and a test case of T'_C : τ_D is the union of instances of τ_B and τ_C . If τ_D is consistent (there are no contradiction between its instances, that is to say in this case that there are no contradictions between instances of τ_B and τ_C), it is a test case of T_D . It is worth noting that

we may not be able to define consistent test case τ_D , this means that the specification D is not consistent under the hypotheses H_D .

For example, we suppose that the specification B contains the previous test goal, and T'_B contains the test cases τ_1 and τ_2 . Moreover, the specification C expresses that the flaps of the aircraft are deployed only during landing or takeoff. We suppose that τ_C is a test case of T'_C :

$$\tau_C = \left(\begin{array}{l} state = val\_cs \\ cruise\_speed = val\_cs \\ flaps = undeployed \end{array} \right)$$

Now, we suppose that the specifications B and C share the variable $state$ and the four constants $\{cruise\_speed, landing, takeoff, parking\}$. We can then build the following test cases by composing τ_C respectively with τ_1 and τ_2 :

$$\theta_1 = \left(\begin{array}{l} alt\_min = 15 \\ altitude = 20 \\ state = val\_cs \\ cruise\_speed = val\_cs \\ flaps = undeployed \end{array} \right) \quad \theta_2 = \left(\begin{array}{l} alt\_min = 15 \\ altitude = 10 \\ state = val\_t \\ takeoff = val\_t \\ state = val\_cs \\ cruise\_speed = val\_cs \\ flaps = undeployed \end{array} \right)$$

θ_1 is a test case of T_D , while θ_2 is not consistent.

Here again, the problem is to verify that the test context (H_D, T_D) is correct for the test goal Ax_D .

4.2 Category of Test Cases

As mentioned just above, we need a theoretical framework to deal with correction of test contexts and its preservation by renaming or composition. To answer this need, we define a notion of extended institution that includes test cases.

We first define a category of test cases on a signature Σ , with a functor **TC** between the category of signatures and the category of test cases. Then we add a second notion of satisfaction, the satisfaction of a test set with the $\prec$ -relation. Finally we add a condition on this relation which expresses the invariance under change of notation. Extended institutions deal with satisfaction of test cases ($\prec$ -relation) in a similar way to the satisfaction of formulas ($\models$ -relation).

Firstly, we define the category of test cases on a signature.

Given a signature Σ , we define the category of test cases on Σ as follows:

- its objects are the test cases, as defined in section 3, which items are defined on the vocabulary of Σ ;

- its morphisms are defined such that: for a test case-morphism $t : \tau_a \rightarrow \tau_b$, at each instance of τ_a we associate an instance of τ_b with the same item. Intuitively, such morphism defines an inclusion between the set of instances of the both test cases (for example we can define a morphism from τ_2 to τ_4 , where τ_2 and τ_4 are defined in section 3).

Test cases and test cases morphisms form a category, called *test cases category on Σ* . Moreover this category is finitely cocomplete: initial objet is an empty test case, push-out is defined by putting together the instances of both test cases without repetition of shared instances.

In such a framework, a test set from a specification (Sig, Ax) is a subset of consistent test cases of the category of test cases on Sig .

4.3 Extended Institution

We can then define a covariant functor **TC** between the category of signatures **Sign** and the category **Cat** of the category of test cases on a signature. This functor associates:

- to each signature, the category of test cases on this signature;
- to each signature morphism, a functor between two categories of test cases.

Moreover, we suppose that for each signature Σ the $\prec_\Sigma$ -relation is defined between models on Σ and test cases on Σ . A test case τ of **TC**(Σ) is consistent if there exists at least one model M of **Mod**(Σ) such that M satisfies τ .

Now, we can give the definition of an *extended institution*: it is an institution plus a functor **TC** and a new satisfaction condition on the $\prec$ -relations.

Definition. An *extended institution* $\mathcal{IE}$ consists of:

1. a category **Sign**, whose object are called *signatures*;
2. a functor $Sen : \mathbf{Sign} \rightarrow \mathbf{Set}$, giving for each signature Σ a set whose elements are called sentences over the signature;
3. a functor $\mathbf{Mod} : \mathbf{Sign} \rightarrow \mathbf{Cat}^{\mathbf{op}}$, giving for each signature Σ a category whose objects are called Σ -models, and whose arrows are called Σ -models-morphisms,
4. a relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times Sen(\Sigma)$ for each Σ in $|\mathbf{Sign}|$, called Σ -satisfaction,
5. a functor $\mathbf{TC} : \mathbf{Sign} \rightarrow \mathbf{Cat}$, giving for each signature Σ a category whose objects are called Σ -test cases, and whose arrows are called Σ -test cases-morphisms and
6. a relation $\prec_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times |\mathbf{TC}(\Sigma)|$ for each Σ in $|\mathbf{Sign}|$,

such that for each morphism $\phi : \Sigma \rightarrow \Sigma'$ in **Sign** the *Satisfaction Conditions*:

$$m' \models_{\Sigma'} Sen(\phi)(e) \Leftrightarrow \mathbf{Mod}(\phi)(m') \models_\Sigma e \quad (2)$$

$$m' \prec_{\Sigma'} \mathbf{TC}(\phi)(\tau) \Leftrightarrow \mathbf{Mod}(\phi)(m') \prec_\Sigma \tau \quad (3)$$

hold for each $m' \in |\mathbf{Mod}(\Sigma')|$, each $e \in Sen(\Sigma)$ and each $\tau \in |\mathbf{TC}(\Sigma)|$.

Remark. In our definition we have chosen a covariant functor **TC**. Intuitively, we generate test cases from the smallest specifications and then define test cases on the bigger specifications. So it seems natural to define a covariant functor.

However, we can also imagine a functor $\mathbf{CT} : \mathbf{Sign} \rightarrow \mathbf{Cat}^{\mathbf{op}}$ contravariant like the **Mod** functor. But in this case it is more difficult to define a satisfaction condition on the $\prec$ -relation. Given a signature-morphism $\phi : \Sigma \rightarrow \Sigma'$, a model M' in $|\mathbf{Mod}(\Sigma')|$, and a test case τ' in $|\mathbf{CT}(\Sigma')|$:

- if we suppose that $M' \prec_{\Sigma'} \tau'$, we can easily verify by forgetting on each side that $\mathbf{Mod}(\phi)(M') \prec_{\Sigma'} \mathbf{CT}(\phi)(\tau')$.
- reversely, if $\mathbf{Mod}(\phi)(M') \prec_{\Sigma'} \mathbf{CT}(\phi)(\tau')$, we can not conclude that M' satisfies τ' , because we can say nothing on the items on Σ' which are not image of an item of Σ .

Thus we chose to define a covariant functor.

Expression of Renaming and Composition in an Extended Institution.

In this definition of extended institution, we can notice that our renaming mechanism according to a specification morphism is described by the image by the functor **TC** of the (signature) morphism.

Composition is more complex: first of all, we rename a test context defined on B and a test context defined on C to obtain two contexts (H'_B, T'_B) and (H'_C, T'_C) defined on D . Then we compose test cases of each context: in fact we compute the push-out of the test cases. Finally the expected test context on D is obtained by removing non consistent test cases.

4.4 Correctness Preservation

The last problem is to conclude on the correctness of the test contexts defined by renaming or composition.

Renaming Given a specification morphism $f : A \rightarrow B$, we suppose we have generated a test context (H_A, T_A) from A , correct for the test goal Ax_A . Then we define by renaming a context (H_B, T_B) on Sig_B .

We prove that this context is correct for the test goal $Sen(f)(Ax_A)$.

We consider a model M' defined on Sig_B such that M' satisfies H_B . Because of the forgetful functor on models and the definition of H_B by renaming, we know that $\mathbf{Mod}(f)(M')$ satisfies H_A . Thus the definition of correctness gives:

$$\mathbf{Mod}(f)(M') \models_{Sig_A} Ax_A \Leftrightarrow \mathbf{Mod}(f)(M') \prec_{Sig_A} T_A$$

The Satisfaction Condition 2 gives that:

$$\mathbf{Mod}(f)(M') \models_{Sig_A} Ax_A \Leftrightarrow M' \models_{Sig_B} Sen(f)(Ax_A)$$

The Satisfaction Condition 3 gives that:

$$\mathbf{Mod}(f)(M') \prec_{Sig_A} T_A \Leftrightarrow M' \prec_{Sig_B} \mathbf{TC}(f)(T_A)$$

Because $\mathbf{TC}(f)(T_A) = T_B$ we obtain that:

$$M' \models_{Sig_B} Sen(f)(Ax_A) \Leftrightarrow M' \prec_{Sig_B} T_B$$

Thus the context (H_B, T_B) is correct for the test goal $Sen(f)(Ax_A)$.

Composition Given a push-out of specifications (cf figure 1), we suppose we have a test context (H_B, T_B) correct for Ax_B and another context (H_C, T_C) correct for Ax_C .

We are going to prove that (H_D, T_D) obtained by composition is correct for the test goal $Ax_D = Sen(p)(Ax_B) \wedge Sen(q)(Ax_C)$.

By renaming according to p we have a test context (H'_B, T'_B) defined on Sig_D and correct for $Sen(p)(Ax_B)$, and by renaming according to q we have a test context (H'_C, T'_C) defined on Sig_D and correct for $Sen(q)(Ax_C)$.

We consider a model M' defined on Sig_D such that M' satisfies $H_D = H'_B \wedge H'_C$. M' satisfies H'_B and H'_C and correctness of (H'_B, T'_B) and (H'_C, T'_C) gives:

$$M' \models_{Sig_D} Sen(p)(Ax_B) \Leftrightarrow M' \prec_{Sig_D} T'_B$$

$$M' \models_{Sig_D} Sen(q)(Ax_C) \Leftrightarrow M' \prec_{Sig_D} T'_C$$

If M' satisfies T'_B and T'_C , we can build a consistent test case on Sig_D by composition, so M' satisfies T_D . Reversely, if M' satisfies T_D , it exists a test case of T_D satisfied by M' and build by composition of a test case of T'_B and a test case of T'_C . So M' satisfies T'_B and T'_C .

So we obtain that:

$$M' \models_{Sig_D} Sen(p)(Ax_B) \wedge Sen(q)(Ax_C) \Leftrightarrow M' \prec_{Sig_D} T_D$$

Because $Ax_D = Sen(p)(Ax_B) \wedge Sen(q)(Ax_C)$, we obtain that (H_D, T_D) is correct for Ax_D .

5 Practical Framework

In the previous sections, we have described the theoretical aspects of our approach. Now we are giving some hints on its practical aspects: the specification language used, the tools developed to support it and results on an industrial application.

5.1 Modular Framework of Specification

In practice, the institution we consider is the Moka-TRIO language: category theory gives the structure of the specifications, the temporal logic TRIO allows to express dynamic behaviors and temporal constraints for each component [20,21].

Moreover, we have added to our language a notion of module [19], adapted from Ehrig and Mahr [10]. We thus dispose of interfaces, encapsulation and genericity which are useful properties for specifying systems. Generation of test sets should reuse the mechanisms of renaming and composition presented in this paper, but a new difficulty appears due to the presence of an interface for each module.

5.2 Moka Toolkit

We have developed at ONERA-CERT a platform to support modular specification and validation. The Moka platform contains four parts:

- The Moka kernel [21] implements the modular specification framework. It encodes the notions of specification, morphism, module, automatizes the operations between modules and implements some checks: parsing of specifications, syntactical verification of morphisms with generation of proof obligations.
- The TRIO Model Generator [6] provides means to validate specifications by animating them. Moreover, it implements generation of specification-based test sets according to the approach described in section 3 [5,9].
- The third part implements the algorithms of renaming and composition presented in this paper [8]. Thus it allows us to generate test sets from structured or modular specifications.
- A user interface [7] makes links between the three other parts and offers a friendly environment to define modular specifications, validate them and generate test sets.

5.3 Application to an Electrical Flight Control System

This approach has been applied to an electrical flight control system which manages the flaps of an airplane during take-off and landing according to values given by sensors (altitude, speed, etc) and pilot commands [9,8].

Our approach allows to generate test sets from a complex specification, when classical approaches failed (for example the TRIO Model Generator directly applied to a result of push-out of specifications).

The main remark is that to limit the growth of the test sets during composition and to avoid generating unusable tests, we must choose carefully the order in which we consider the composition operations: it is better to start from components which have strong interactions, (which share lots of items). However, in each test set, the size of the test cases grows like the size of the specifications built by push-out.

We have also specified this application using modules. The main advantage is that the size of test cases does not grow because we only keep the items of the interface of the modules [9,8].

6 Conclusion

We have defined an extension of the notion of institution in order to take into account test cases. With this extended framework, we can formally relate test cases to specifications and models, express the correctness criteria and prove their preservation under renaming and composition. We thus dispose of a unified theoretical framework for our approach of incremental test generation.

This approach is useful in practice to handle test generation for large systems. Classical approaches are limited by the size of the specifications while our incremental method allows to exploit the structure of the system. Moreover the approach is supported by a toolkit; it automatizes the construction of the system specification and the generation of test sets, thus allowing to apply our approach to real industrial systems like the electrical flight control system.

Areas of future work include exploring links between proof and test, taking advantage of Meseguer work on general logics [18]. The practical aspect would be to use the proof of a property on a specification to generate test cases (property-based test generation).

It would also be interesting to work on the notion of trusted component where the description of a component contains not only its specification but also its properties [22] and the generated test cases that will be runned on the implementation of the component. In this way, reuse can be applied to the component but also to its properties and test cases; this kind of approach should help in the management of change for large structured systems.

References

1. M. Arrais and J. L. Fiadeiro. Unifying theories in different institutions. In *Recent Trends in Data Type Specification*, volume 1130 of *LNCS*, pages 81–101. Springer Verlag, 1996.
2. S. Barbey, D. Buchs, and C. Péraire. Test selection for object-oriented software based on formal specifications. In *Proceedings of PROCOMET'98*, number Technical Report EPFL-DI No 97/252, December 1997.
3. G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6, November 1991.
4. E. Brinksma. Formal methods for conformance testing: Theory can be practical. In *CAV'99*, number 1633 in *LNCS*, pages 44–46. Springer Verlag, July 1999.
5. J. Cazin, C. Seguin, W. Osnowycz, E. Ciapessoni, and E. Ratto. An experience in the specification and test of an electrical flight control system using a temporal logic framework. In *DASIA 96 Data Systems in Aerospace*. ESA publications, May 1996.
6. E. Ciapessoni, E. Corsetti, M. Migliorati, and E. Ratto. Specifying industrial real-time systems in a logical framework. In *ICLP 94 - Post Conference Workshop on Logic Programming in Software Engineering*, 1994.

7. A. Daurat, M. Doche, and D. Le Berre. *Manipulations des principaux objets du formalisme Moka-TRIO par l'interface Java-Moka*. ONERA-CERT / DTIM, 1999.
8. M. Doche. *Techniques formelles pour l'évaluation de systèmes complexes. Test et modularité*. PhD thesis, ENSAE, ONERA-CERT/DTIM, December 1999.
9. M. Doche, C. Seguin, and V. Wiels. A modular approach to specify and test an electrical flight control system. In *FMICS-4, Fourth International Workshop on Formal Methods for Industrial Critical Systems*, July 1999.
10. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2 : Modules specifications and constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.
11. J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent system specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.
12. J.H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper and Row, 1986.
13. M-C. Gaudel. Testing can be formal, too. In *TAPSOFT'95*, pages 82–96. Springer Verlag, 1995.
14. C. Ghezzi, D. Mandrioli, and A. Morzenti. A model parametric real-time logic. *ACM Transactions on programming languages and systems*, 14(4):521–573, October 1992.
15. J. A. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, January 1992.
16. T. Jérón and P. Morel. Test generation derived from model-checking. In *Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, pages 108–121. Springer-Verlag, July 1999.
17. D. Mandrioli, S. Morasca, and A. Morzenti. Functionnal test case generation for real-time systems. In *Dependable computing for critical applications 3.*, volume 8 of *Dependable computing and fault-tolerant systems*, pages 29–61. Springer Verlag, 1992.
18. J. Meseguer. General logics. In H-D. Ebbinghaus and all, editors, *Proceedings of Logic Colloquium*. North-Holland, 1987.
19. P. Michel and V. Wiels. A framework for modular formal specification and verification. In *Proceedings of FME'97*, number 1313 in *LNCS*. Springer-Verlag, 1997.
20. C. Seguin and W. Wiels. Using a logical and categorical approach for the validation of fault-tolerant systems. In *FME96, Formal Methods Europe*, March 1996.
21. V. Wiels. *Modularité pour la conception et la validation formelles de systèmes*. PhD thesis, ENSAE - ONERA/CERT/DERI, October 1997.
22. V. Wiels and S. Easterbrook. Management of evolving specifications using category theory. In *Proceedings of Automated Software Engineering'98*, 1998.
23. M. Wirsing. Algebraic specification. In *Handbook of theoretical computer science, Formal Models and Semantics*, volume B, pages 675–788. Elsevier and MIT Press, 1990.

Testing from Structured Algebraic Specifications

Patrícia D.L. Machado*

LFCS, Division of Informatics, University of Edinburgh, JMCB, King's Buildings,
Edinburgh EH9 3JZ, UK
pd1m@dcs.ed.ac.uk

Abstract. This paper deals with testing from structured algebraic specifications expressed in first-order logic. The issue investigated is the so-called oracle problem, that is, whether a finite and executable procedure can be defined for interpreting the results of tests. For flat specifications, the oracle problem often reduces to the problem of comparing values of a non-observable sort and how to deal with quantifiers. However, specification-building operations introduce an additional barrier to this problem which can restrict the way specifications and test suites are defined. In this paper, we present a framework for testing from structured specifications and a thorough discussion of the problems which can arise together with proposed solutions.

1 Introduction

Specification-based testing is concerned with deriving testing information from formal specifications of programs to permit testing to be integrated in formal development frameworks. The intention is to apply testing together with proofs whenever full formal verification is not cost-effective. In this case, testing usually corresponds to check whether specification axioms are satisfied by a program with respect to a given test set. Several formal approaches to testing have been proposed [1,5,8,12,7,11]. Some interesting problems to look into in this area are how to give an accurate interpretation of testing results and also how to define finite test sets to replace exhaustive ones.

Oracles are decision procedures to interpret the results of tests. Some solutions to the so-called oracle problem, that is, whether such procedures can be defined, have already been proposed for flat specifications [8,7,11]. An initial attempt towards a modular testing approach for hierarchical conditional specifications is given in [7]. For flat algebraic specifications expressed in first-order logic, the oracle problem often reduces to the problem of comparing two values of a non-observable sort (equality problem) and also how to deal with quantifiers which may demand infinite test sets. In this context, *approximate equalities*, binary relations on the carriers of an algebra, are enough to sort out the equality problem under certain conditions [11].

* Supported by CNPq – Brazilian Research Council and on leave from DSC/CCT, Federal University of Paraíba, Campina Grande-PB, Brazil patricia@dsc.ufpb.br

When considering structured specifications, the oracle problem can be harder, since the structure has to be taken into account in order to make sense of axioms and define the necessary tests. For instance, as a given sort can be introduced in different signatures in the structure of a specification, a family of equalities, one for each signature may be required. Specification-building operations may impose restrictions on what families can be used. Moreover, it may be necessary to check axioms composed of hidden definitions which are not implemented in the program under test. Furthermore, for practicability and efficiency, it is reasonable to have different test sets for the same sort in different signatures.

In the sequel, we refer to *structured testing* as the style of testing which takes the structure of the specification into account in order to test whether an algebra satisfies it. In this case, different tests may be necessary for different parts of the specification. On the other hand, *flat testing* checks an algebra against a flat version of the specification, that is, an unstructured set of axioms computed from the original specification. In this case, a single test needs to be performed. Many advantages arise from performing structured testing rather than flat testing. For instance, more accurate results can be achieved, insofar as there is more flexibility to define approximate equalities and appropriately tackle hidden definitions. In this paper, we show how oracles and test sets can be defined from structured specifications composed of specification-building operations like union, translate and hide [13,9,2]. We deal with the oracle problem in this context, comparing testing results with behavioural satisfaction [3] and considering the instantiation to the specific case of observational satisfaction. Whether structured testing coincides with flat testing is also discussed.

The paper is structured as follows. Section 2 presents a definition of test sets, a testing satisfaction relation and families of equalities. Section 3 introduces specifications with testing interface and reviews previous results obtained for flat specifications in [11]. Section 4 presents operations to construct structured specifications with testing interface. An example is given to illustrate the main ideas and results. Section 5 presents some theorems which demonstrate advantages/drawbacks of structured testing versus flat testing. Section 6 instantiates the results obtained in section 5 to the observational case. Finally, concluding remarks along with pointers for future work are given in section 7. The reader is assumed to be familiar with general concepts of algebraic specifications [13]. Let $\Sigma = (S, F)$ be a signature and $T_\Sigma(X)$ be the Σ -term algebra, where X is an S -indexed set of countably infinite sets of variables. For any two Σ -terms t and t' , $t = t'$ is a Σ -equation and first-order Σ -formulas are built from Σ -equations, the usual logical connectives and quantifiers. A Σ -sentence is a Σ -formula without free variables. For any Σ -algebra A and $\alpha : X \rightarrow |A|$, there exists a unique Σ -homomorphism $\alpha^\# : T_\Sigma(X) \rightarrow A$ which extends α . The value of $t \in |T_\Sigma(X)|_s$ in A under α is $\alpha^\#(t) \in |A|_s$, where $s \in S$. If $t \in T_\Sigma$, i.e., t is a ground Σ -term, the value of t in A is $^\#(t)$, where $^\# : T_\Sigma \rightarrow A$ is the unique homomorphism.

2 Testing Satisfaction

In the field of specification-based testing, test sets are usually defined from specifications rather than from programs. The reason is that we are ultimately interested in verifying properties stated in the specification. Test sets are defined here as sets of ground terms: $T = \{T_s\}_{s \in S}$ is a Σ -test set if $T \subseteq |T_\Sigma|$.

Equality on values of a Σ -algebra A can be interpreted by an indistinguishability relation – *behavioural equality* – a partial Σ -congruence $\approx_A$, denoted by a family $\approx_A = (\approx_{A,s})_{s \in S}$ of partial equivalence relations which are symmetric and transitive relations and compatible with Σ , i.e., $\forall f \in F : s_1 \dots s_n \longrightarrow s$, $\forall a_i, b_i \in A_{s_i}$, if $a_i \approx_{A,s_i} b_i$, then $f^A(a_1, \dots, a_n) \approx_{A,s} f^A(b_1, \dots, b_n)$. The observational equality $\approx_{Obs,A} = (\approx_{Obs,A,s})_{s \in S}$ is a special case of the behavioural equality where related elements are those which cannot be distinguished by observable computations.¹ A behavioural equality $\approx_A$ is *total* if all relations are also reflexive. The definition domain of $\approx_A$ is given as $Dom(\approx_A) = \{a \mid a \approx_A a\}$. Let $\approx = (\approx_A)_{A \in Alg(\Sigma)}$ be a Σ -behavioural equality, a family of behavioural equalities, one for each Σ -algebra A [3]. As the relations in $\approx$ are compatible with Σ , they are reflexive on values of ground terms, i.e., $\forall v \in \#(T_\Sigma) \Rightarrow v \in Dom(\approx_A)$.

As it is not always possible or it may be too complicated to give an implementation of the behavioural equality for testing purposes, we might need to define something simpler but more feasible. Let a Σ -approximate equality $\sim = (\sim_A)_{A \in Alg(\Sigma)}$ be a family of approximate equalities, one for each Σ -algebra A , as defined in [11]. An *approximate equality* $\sim_A$ is a binary relation on A which is *sound* if and only if $\forall a, a' \in Dom(\approx_A) \cdot a \sim_A a' \Rightarrow a \approx_A a'$, and is *complete* if and only if $\forall a, a' \cdot a \approx_A a' \Rightarrow a \sim_A a'$. When two approximate equalities are applied together, one sound and one complete, they are enough to sort out the equality problem under certain conditions as we show in section 3.

The satisfaction relation given below is a generalisation of the standard notion of behavioural satisfaction [3] with equality interpreted by the approximate equalities and quantifiers ranging over the test set rather than all values in the definition domain. This relation establishes whether Σ -algebras satisfies Σ -formulas w.r.t a given valuation, pair of approximate equalities and test set.

Definition 1 (Testing Satisfaction Relation). Let Σ be a signature, T be a Σ -test set and $\sim, \simeq$ be two Σ -approximate equalities. Let A be a Σ -algebra and $\alpha : X \rightarrow Dom(\approx_A)$ be a valuation. The testing satisfaction relation denoted by $\models_{\sim, \simeq}^T$ is defined as follows.

1. $A, \alpha \models_{\sim, \simeq}^T t = t' \text{ iff } \alpha^\#(t) \sim_A \alpha^\#(t')$;
2. $A, \alpha \models_{\sim, \simeq}^T \neg \psi \text{ iff } A, \alpha \models_{\simeq, \sim}^T \psi \text{ does not hold}$;
3. $A, \alpha \models_{\sim, \simeq}^T \psi_1 \wedge \psi_2 \text{ iff both } A, \alpha \models_{\sim, \simeq}^T \psi_1 \text{ and } A, \alpha \models_{\sim, \simeq}^T \psi_2 \text{ hold}$;
4. $A, \alpha \models_{\sim, \simeq}^T \forall x : s \cdot \psi \text{ iff } A, \alpha[x \mapsto v] \models_{\sim, \simeq}^T \psi \text{ holds for all } v \in \#(T)_s$.

¹ Values a and b of a non-observable sort s are observationally equal $a \approx_{Obs,A} b$ iff they cannot be distinguished by observable contexts, i.e., $\forall C \in C_{Obs} \forall \alpha : X \rightarrow |A| \cdot \alpha_a^\#(C) = \alpha_b^\#(C)$, where C_{Obs} is the set of all observable contexts and $\alpha_a, \alpha_b : X \cup \{z_s\} \rightarrow |A|$ are the unique extensions of α defined by $\alpha_a(z_s) = a$ and $\alpha_b(z_s) = b$.

where $\alpha[x \mapsto v]$ denotes the valuation α superseded at x by v . In this relation, $\sim$ is always applied in positive positions and $\simeq$ is always applied in negative positions. Positive/negative positions are as usual in logic. Note that equalities are reversed when negative positions are reached.

If $A, \alpha \models_{\sim, \simeq}^T \phi$ for all $\phi \in \Phi$, then $A, \alpha \models_{\sim, \simeq}^T \Phi$, where Φ is a set of Σ -formulas. When ϕ is a Σ -sentence, we write $A \models_{\sim, \simeq}^T \phi$ without α . Note that valuations ranges over $Dom(\approx_A)$ rather than $|A|$, as we consider $\approx$ to be partial.

As we aim to deal with structured specifications, where the signature of different parts of the specification may be different, in the sequel let $\approx = (\approx_\Sigma)_{\Sigma \in \text{Sign}}$ be a family of Σ -behavioural equalities, one for each signature Σ , where Sign is the category of signatures. A Σ -behavioural equality is defined as before $\approx_\Sigma = (\approx_{\Sigma, A})_{A \in \text{Alg}(\Sigma)}$, one congruence relation for each algebra A . Whenever Σ is obvious, $\approx$ is used without subscript to denote $\approx_\Sigma$. When A is also obvious, $\approx$ and $\approx_\Sigma$ are used to denote $\approx_{\Sigma, A}$. Likewise, let $\sim = (\sim_\Sigma)_{\Sigma \in \text{Sign}}$ and $\simeq = (\simeq_\Sigma)_{\Sigma \in \text{Sign}}$ denote families of Σ -approximate equalities. The reduct of a Σ -equality $\sim_\Sigma$ by the morphism $\sigma : \Sigma' \rightarrow \Sigma$ considers only the relations of sorts in Σ' , i.e., $(\sim_\Sigma)|_\sigma = ((\sim_{\Sigma, A})|_\sigma)_{A \in \text{Alg}(\Sigma)}$ is the reduct of $\sim_\Sigma$ by $\sigma : \Sigma' \rightarrow \Sigma$, where $(\sim_{\Sigma, A})|_\sigma = ((\sim_{\Sigma, A})_{\sigma(s)})_{s \in \text{Sorts}(\Sigma')}$. Regarding signature morphisms, families can be compatible, reduction-compatible and translation-compatible as follows.

Definition 2 (Compatible). *The family $\sim$ is compatible with signature morphisms in Sign if for all $\sigma : \Sigma' \rightarrow \Sigma$ and all Σ -algebras A , $\sim_{\Sigma', A|_\sigma} = (\sim_{\Sigma, A})|_\sigma$.*

Proposition 1. *The literal equality $=$ on values of an algebra is compatible.*

Definition 3 (Reduction-Compatible). *The family $\sim$ is reduction-compatible with signature morphisms in Sign if for all $\sigma : \Sigma' \rightarrow \Sigma$ and all Σ -algebras A , $\sim_{\Sigma', A|_\sigma} \supseteq (\sim_{\Sigma, A})|_\sigma$.*

Proposition 2. *The family $\approx_{\text{Obs}} = (\approx_{\text{Obs}, \Sigma})_{\Sigma \in \text{Sign}}$ of Σ -observational equalities is reduction-compatible.*

Note that the family $\approx_{\text{Obs}}$ is not compatible. Suppose $\sigma : \Sigma' \hookrightarrow \Sigma$ is an inclusion. In this case, Σ may have more observers than Σ' . Thus, $(\approx_{\text{Obs}, \Sigma})|_\sigma$ may be finer than $\approx_{\text{Obs}, \Sigma'}$, i.e., $(\approx_{\text{Obs}, \Sigma})|_\sigma \subseteq \approx_{\text{Obs}, \Sigma'}$, but $(\approx_{\text{Obs}, \Sigma})|_\sigma \not\supseteq \approx_{\text{Obs}, \Sigma'}$.

Definition 4 (Translation-Compatible). *The family $\sim$ is translation-compatible with signature morphisms in Sign if for all $\sigma : \Sigma' \rightarrow \Sigma$ and all Σ -algebras A , $\sim_{\Sigma', A|_\sigma} \subseteq (\sim_{\Sigma, A})|_\sigma$.*

Let $\sigma : \Sigma' \rightarrow \Sigma$ be a signature morphism. This gives rise to translate Σ' -terms to Σ -terms and Σ' -formulas to Σ -formulas. Let T' be a Σ' -test set, Φ' be a set of Σ' -formulas and A be Σ -algebra. Let $\alpha : X \rightarrow \text{Dom}(\approx_A)$ and $\alpha' : X' \rightarrow \text{Dom}(\approx_{A|_\sigma})$ be valuations such that $\forall s \in \text{Sorts}(\Sigma') \cdot \alpha'(x_s) = \alpha(\sigma(x_s))$. Translations of Σ' -formulas and Σ' -test sets do only preserve the testing satisfaction relation under certain conditions on the families of equalities being considered.

Theorem 1. *If $\sim$ is reduction-compatible and $\simeq$ is translation-compatible then $A, \alpha \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{\sigma(T')} \sigma(\Phi')$ implies $A|_{\sigma}, \alpha' \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{T'} \Phi'$.*

Proof. We need to show that the above conclusion holds for all $\phi' \in \Phi'$. By induction on ϕ' , and Definitions 3 and 4.

Theorem 2. *If $\sim$ is translation-compatible and $\simeq$ is reduction-compatible then $A|_{\sigma}, \alpha' \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{T'} \Phi'$ implies $A, \alpha \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{\sigma(T')} \sigma(\Phi')$.*

Obviously, if $\sim$ and $\simeq$ are compatible, the implication holds in both directions, that is, translations preserve the testing satisfaction relation.

3 Basic Specifications with Testing Interface

Providing test suites in the specification interface means that they can be uniformly planned at specification level. The specification together with testing suites is called a *specification with testing interface*. Motivations for this come also from the notion of *deliverables* [4], which consists of a program together with a proof of correctness. Focusing on testing, the idea is that for each program, an oracle, defined from the specification axioms, verifies the program based on the test set. Basic specifications with testing interface are defined as follows.

Definition 5 (Basic Specification). *Let Σ be a signature, T be a Σ -test set and Φ be a set of Σ -sentences. $SP = \langle \Sigma, \Phi, T \rangle$ is a basic specification, with:*

$$Sig(SP) \stackrel{def}{=} \Sigma$$

$$Test(SP) \stackrel{def}{=} T$$

$$Mod_{\approx}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A \models_{\approx_{\Sigma}} \Phi\}$$

$$ChMod_{\sim, \simeq}(SP) \stackrel{def}{=} \{A \in Alg(\Sigma) \mid A \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^T \Phi\}$$

where $Mod_{\approx}(SP)$ denotes the class of “real” models of SP w.r.t. a Σ -behavioural equality $\approx_{\Sigma} = (\approx_{\Sigma, A})_{A \in Alg(\Sigma)}$ and $ChMod_{\sim, \simeq}(SP)$ denotes the class of “checkable” models of SP by testing w.r.t. the Σ -approximate equalities $\sim_{\Sigma} = (\sim_{\Sigma, A})_{A \in Alg(\Sigma)}$, $\simeq_{\Sigma} = (\simeq_{\Sigma, A})_{A \in Alg(\Sigma)}$ and the Σ -test set T . Note that we consider behavioural satisfaction instead of standard satisfaction for defining real models.

The following theorems show some situations in which a real model A is also a checkable model and vice-versa. These are the main results obtained in [11].

Theorem 3. *Let SP be a basic specification. If $\sim_{\Sigma}$ is a complete equality,² $\simeq_{\Sigma}$ is a sound equality, and for all $\phi \in \Phi$, ϕ has only positive occurrences of $\forall$ and negative occurrences of $\exists$ then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim, \simeq}(SP)$.*

Proof. Show that $A \models_{\approx} \phi$ implies $A \models_{\sim, \simeq}^T \phi$ for all $\phi \in \Phi$. By induction on ϕ .

² $\sim_{\Sigma}$ is complete (sound) w.r.t $\approx_{\Sigma}$ iff for all A , $\sim_{\Sigma, A}$ is complete (sound) w.r.t $\approx_{\Sigma, A}$.

Theorem 3 covers a common use of $\forall$ and $\exists$ whereby, as one might expect, incorrect programs can be accepted by testing satisfaction. This theorem generalises previous results obtained by Gaudel in the context of positive conditional specifications [8] where the only negative occurrences of equations are in the conditions which are restricted to observable equations so that the computational equality, which is sound and complete on observable sorts, can be used. The black-box equality, which is complete but need not be sound, is used in the conclusion, which is a positive occurrence. Furthermore, the only quantifiers in conditional equations are universal quantifiers outermost, that is, in positive position. Theorem 4 below is the dual of 3 and covers a case where correct programs can be rejected by testing, but success in testing guarantees correctness.

Theorem 4. *Let SP be a basic specification. If $\sim_\Sigma$ is a sound equality, $\simeq_\Sigma$ is a complete equality and for all $\phi \in \Phi$, ϕ has only negative occurrences of $\forall$ and positive occurrences of $\exists$ then $A \in ChMod_{\sim, \simeq}(SP)$ implies $A \in Mod_{\approx}(SP)$.*

The conditions on Theorem 4 mean that it is rarely applicable. Furthermore, assumptions on quantifiers can always be dropped for Theorem 3 if T is unbiased and for Theorem 4 if T is valid [11]. Note that the assumptions on quantifiers in Theorem 3 is equivalent to restrict its application to universal formulas.

4 Structured Specifications with Testing Interface

Definition 6 below, presents some specification-building operations [13,9] for creating structured specifications with testing interface.

Definition 6 (Structured Specification).

$$SP = \langle \Sigma, \Phi, T \rangle \mid SP_1 \cup SP_2 \mid \text{translate } SP' \text{ with } \sigma \mid \text{hide sorts } S' \text{ opns } F' \text{ in } SP'$$

where $\langle \Sigma, \Phi, T \rangle$ is a basic specification, SP' , SP_1 , SP_2 are structured specifications, σ is a morphism, S' is a set of sorts and F' is a set of function declarations.

The testing satisfaction relation cannot always check all axioms of a structured specification because they may refer to hidden sorts and functions not implemented in the algebra being considered. Therefore, it may be necessary to look at visible (testable) axioms only.

Definition 7 (Testable Axioms). *The set of testable (visible) axioms of a structured specification SP can be defined as follows.*

- $TAx(\langle \Sigma, \Phi, T \rangle) \stackrel{\text{def}}{=} \Phi$
- $TAx(SP_1 \cup SP_2) \stackrel{\text{def}}{=} TAx(SP_1) \cup TAx(SP_2)$
- $TAx(\text{translate } SP' \text{ with } \sigma) \stackrel{\text{def}}{=} \sigma(TAx(SP'))$
- $TAx(\text{hide sorts } S' \text{ opns } F' \text{ in } SP') \stackrel{\text{def}}{=} TAx(SP') \cap Sen(\Sigma)$, where $\Sigma = Sig(\text{hide sorts } S' \text{ opns } F' \text{ in } SP')$

Note that name clashes do not occur in $TAx(SP)$ because axioms with hidden symbols are excluded.

In the next subsections, the semantics of the specification-building operations is given as usual. Also, some lemmas are given concerning whether flat testing, which is represented by testing satisfaction of visible axioms ($A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$), implies structured testing, which is represented by membership in the class of checkable models ($A \in ChMod_{\sim, \simeq}(SP)$), and vice-versa. These lemmas are used to prove the main theorems presented in section 5. Example 1 below is used to illustrate the points discussed in the sequel.

Example 1. Let S_1 and S_2 be signatures defined as follows.

$$S_1 = \left(\begin{array}{l} \text{sorts } \{list\} \\ \text{opns } \{nil : list \\ \quad cons : int * list \rightarrow list \\ \quad head : list \rightarrow int \\ \quad tail : list \rightarrow list\} \end{array} \right) \quad S_2 = \left(\begin{array}{l} \text{sorts } \{list\} \\ \text{opns } \{... S_1 \text{ operations } ... \\ \quad is\_sorted : list \rightarrow bool \\ \quad sort : list \rightarrow list\} \end{array} \right)$$

Let $\sigma : S_1 \hookrightarrow S_2$. The following are well-formed structured specifications.

$$List = \left[\begin{array}{ll} \text{sig} & S_1 \\ \text{testset} & T_1 \\ \text{axioms} & \forall l : list; x : int \cdot cons(x, l) \neq l \\ & \forall l : list; x : int \cdot head(cons(x, l)) = x \\ & \forall l : list; x : int \cdot tail(cons(x, l)) = l \end{array} \right]$$

$$SList1 = \text{translate } List \text{ by } \sigma \quad SList2 = \left[\begin{array}{ll} \text{sig} & S_2 \\ \text{testset} & T_2 \\ \text{axioms} & is\_sorted(nil) = true \\ & \dots \\ & \forall l \cdot is\_sorted(sort(l)) = true \end{array} \right]$$

$$SList = SList1 \cup SList2 \quad IntList = \text{hide sorts } \emptyset \text{ opns } \{is\_sorted\} \text{ in } SList$$

4.1 Union

The union of two specifications with the same signature gives a new specification whose models are the ones in the intersection of the sets of models of the former.

Definition 8 (Union). Let SP_1 and SP_2 be structured specifications, such that $Sig(SP_1) = Sig(SP_2)$. Then $SP = SP_1 \cup SP_2$ is defined as follows.

- $Sig(SP) \stackrel{def}{=} Sig(SP_1) = Sig(SP_2)$
- $Test(SP) \stackrel{def}{=} Test(SP_1) \cup Test(SP_2)$
- $Mod_{\approx}(SP) \stackrel{def}{=} Mod_{\approx}(SP_1) \cap Mod_{\approx}(SP_2)$
- $ChMod_{\sim, \simeq}(SP) \stackrel{def}{=} ChMod_{\sim, \simeq}(SP_1) \cap ChMod_{\sim, \simeq}(SP_2)$

Let $SP = SP_1 \cup SP_2$. Then, $A \in ChMod_{\sim, \simeq}(SP)$ may not imply $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$, even if both SP_1 and SP_2 are basic specifications, because axioms of SP_1 and SP_2 are tested with a unique, possibly bigger, test set $Test(SP)$ in the latter. Similarly, the converse may not hold. The reason is that, due to the existential quantifier, the result obtained by checking an axiom with $Test(SP)$ may not imply the result obtained by checking the same axiom with $Test(SP_1) \subseteq Test(SP)$. In Example 1, T_1 and T_2 may be different, so the axioms of $List$ and $SList2$ may be checked with different test sets when testing $SList$ in the former and latter way. Thus, it is necessary to specify some restrictions on test sets. One possible restriction is that they must coincide at least on sorts referred by the axioms of both SP_1 and SP_2 . In Example 1, this would also imply that, for these sorts, T_2 cannot include ground terms with symbols not in S_1 .

Lemma 1. *Let $SP = SP_1 \cup SP_2$ be a structured specification. Let A be a $Sig(SP)$ -algebra. Let S be the intersection of the sets of sorts of bound variables in $TAx(SP_1)$ and in $TAx(SP_2)$. If*

1. (a) $Test(SP_1) \upharpoonright_S = Test(SP_2) \upharpoonright_S$ and
 (b) $A \in ChMod_{\sim, \simeq}(SP_1)$ implies $A \models_{\sim, \simeq}^{Test(SP_1)} TAx(SP_1)$ and
 (c) $A \in ChMod_{\sim, \simeq}(SP_2)$ implies $A \models_{\sim, \simeq}^{Test(SP_2)} TAx(SP_2)$
 then $A \in ChMod_{\sim, \simeq}(SP)$ implies $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$
2. (a) $Test(SP_1) \upharpoonright_S = Test(SP_2) \upharpoonright_S$ and
 (b) $A \models_{\sim, \simeq}^{Test(SP_1)} TAx(SP_1)$ implies $A \in ChMod_{\sim, \simeq}(SP_1)$ and
 (c) $A \models_{\sim, \simeq}^{Test(SP_2)} TAx(SP_2)$ implies $A \in ChMod_{\sim, \simeq}(SP_2)$
 then $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$ implies $A \in ChMod_{\sim, \simeq}(SP)$

Proof. (1.) Suppose $A \in ChMod_{\sim, \simeq}(SP)$. Then $A \in ChMod_{\sim, \simeq}(SP_1)$ and $A \in ChMod_{\sim, \simeq}(SP_2)$. By assumption 1(b), $A \models_{\sim, \simeq}^{Test(SP_1)} TAx(SP_1)$ and by assumption 1(c), $A \models_{\sim, \simeq}^{Test(SP_2)} TAx(SP_2)$. Then, by 1(a), $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP_1)$ and $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP_2)$. So, $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$. (2.) Similar to item 1. $\square$

Note that, by Definition 5, assumptions 1(b)-(c) and 2(b)-(c) in Lemma 1 will hold if SP_1 and SP_2 are basic specifications.

4.2 Translate

Given a specification and a signature morphism, translate produces a new specification from the former according to the morphism.

Definition 9 (Translate). *Let SP' be a structured specification. $SP = \text{translate } SP'$ with σ , where $\sigma : \Sigma' \rightarrow \Sigma$ and $Sig(SP') = \Sigma'$, is defined as follows.*

- $Sig(SP) \stackrel{\text{def}}{=} \Sigma$
- $Test(SP) \stackrel{\text{def}}{=} \sigma(Test(SP'))$
- $Mod_{\approx}(SP) \stackrel{\text{def}}{=} \{A \in Alg(\Sigma) \mid A|_{\sigma} \in Mod_{\approx}(SP')\}$
- $ChMod_{\sim, \simeq}(SP) \stackrel{\text{def}}{=} \{A \in Alg(\Sigma) \mid A|_{\sigma} \in ChMod_{\sim, \simeq}(SP')\}$

Let $SP = \text{translate } SP' \text{ with } \sigma : \Sigma' \rightarrow \Sigma$. Then, $A \in ChMod_{\sim, \simeq}(SP)$ implies $A \models_{\sim, \simeq}^{Test(SP)} TA_x(SP)$ and vice-versa, provided that this holds in both directions for SP' and appropriate assumptions on equalities are made, since translation of terms and formulas does not always respect the testing satisfaction relation. Consider Example 1 and let $\sim$ denote a family of approximate equalities defined by a finite set of observable contexts. Clearly, $\sim_{S_2}$ can be finer than $\sim_{S_1}$, since S_2 has an additional *list* observer (*is_sorted*). Moreover, it is easy to check that flat testing of *SList1* considers $\sim_{S_2}$, whereas structured testing considers $\sim_{S_1}$. Therefore, compatibility conditions are necessary to compare them.

Lemma 2. *Let $SP = \text{translate } SP' \text{ with } \sigma$. Let A be a Σ -algebra. If*

1. (a) $\sim$ is translation-compatible and $\simeq$ is reduction-compatible and
 (b) $A|_\sigma \in ChMod_{\sim, \simeq}(SP')$ implies $A|_\sigma \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{Test(SP')} TA_x(SP')$
 then $A \in ChMod_{\sim, \simeq}(SP)$ implies $A \models_{\sim_\Sigma, \simeq_\Sigma}^{Test(SP)} TA_x(SP)$.
2. (a) $\sim$ is reduction-compatible and $\simeq$ is translation-compatible and
 (b) $A|_\sigma \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{Test(SP')} TA_x(SP')$ implies $A|_\sigma \in ChMod_{\sim, \simeq}(SP')$
 then $A \models_{\sim_\Sigma, \simeq_\Sigma}^{Test(SP)} TA_x(SP)$ implies $A \in ChMod_{\sim, \simeq}(SP)$.

Proof. (1.) Suppose $A \in ChMod_{\sim, \simeq}(SP)$. Then $A|_\sigma \in ChMod_{\sim, \simeq}(SP')$ by Definition 9. By assumption, $A|_\sigma \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{Test(SP')} TA_x(SP')$. Thus, by Theorem 2, $A \models_{\sim_\Sigma, \simeq_\Sigma}^{Test(SP)} TA_x(SP)$. (2.) Similar to item 1. $\square$

Assumptions 1(b) and 2(b) in Lemma 2 will hold if SP' is a basic specification.

4.3 Hiding

The hiding operation constructs a new specification from SP whose models are the models of SP with some sorts and functions removed.

Definition 10 (Hiding). *Let SP' be a specification. $SP = \text{hide sorts } S' \text{ opns } F' \text{ in } SP'$ is defined as follows.*

- $Sig(SP) \stackrel{def}{=} Sig(SP') - \langle S', F' \rangle$
- $Test(SP) \stackrel{def}{=} Test(SP') \cap T_{Sig(SP)}$
- $Mod_{\sim}(SP) \stackrel{def}{=} \{A'|_{\Sigma'} \mid A' \in Mod_{\sim}(SP')\}$
- $ChMod_{\sim, \simeq}(SP) \stackrel{def}{=} \{A'|_{\Sigma'} \mid A' \in ChMod_{\sim, \simeq}(SP')\}$

where S' is a set of sorts, F' is a set of function declarations and $\Sigma' = Sig(SP)$ is required to be a well-formed signature.

Let $SP = \text{hide sorts } S' \text{ opns } F' \text{ in } SP'$. In order to check whether $A \in ChMod_{\sim, \simeq}(SP)$, it is necessary to check whether $A' \in ChMod_{\sim, \simeq}(SP')$ for some $Sig(SP')$ -algebra A' such that $A = A'|_{Sig(SP)}$. In Example 1, one must provide a *SList*-algebra A' (with an implementation of *is_sorted*) in order to check A

against *IntList* by structured testing. Clearly, $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$ may not imply $A \in ChMod_{\sim, \simeq}(SP)$ as SP' may have axioms not in $TAx(SP)$ and the test set of SP may be smaller than the test set of SP' . Likewise, the converse may not hold. Note that $TAx(IntList)$ does not include any axiom referring to *is_sorted* and $Test(IntList)$ does not include any ground term referring to *is_sorted*. Furthermore, when *hide* is present, assumptions on equalities are also necessary in order to compare flat and structured testing. In Example 1, the observer *is_sorted* is hidden.

Lemma 3. *Let $SP = \text{hide sorts } S' \text{ opns } F'$ in SP' with $\Sigma = Sig(SP')$ and $\Sigma' = Sig(SP)$. Let A' be a Σ -algebra. If*

1. (a) $\sim$ is reduction-compatible and $\simeq$ is translation-compatible and
 (b) $TAx(SP)$ has only positive $\forall$ and negative $\exists$ and
 (c) $A' \in ChMod_{\sim, \simeq}(SP')$ implies $A' \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{Test(SP')} TAx(SP')$
 then $A'|_{\Sigma'} \in ChMod_{\sim, \simeq}(SP)$ implies $A'|_{\Sigma'} \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{Test(SP)} TAx(SP)$
2. (a) $TAx(SP) = TAx(SP')$ and
 (b) $\sim$ is translation-compatible and $\simeq$ is reduction-compatible and
 (c) $TAx(SP)$ has only negative $\forall$ and positive $\exists$ and
 (d) $A' \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{Test(SP')} TAx(SP')$ implies $A' \in ChMod_{\sim, \simeq}(SP')$
 then $A'|_{\Sigma'} \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{Test(SP)} TAx(SP)$ implies $A'|_{\Sigma'} \in ChMod_{\sim, \simeq}(SP)$

Proof. (1.) Suppose $A'|_{\Sigma'} \in ChMod_{\sim, \simeq}(SP)$. Then, $A' \in ChMod_{\sim, \simeq}(SP')$. By 1(c), $A' \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{Test(SP')} TAx(SP')$. By definition of $TAx(SP)$, $A' \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{Test(SP')} TAx(SP)$. As $Test(SP) \subseteq Test(SP')$, by 1(b), $A' \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{Test(SP)} TAx(SP)$. Let $\sigma : \Sigma' \hookrightarrow \Sigma$ be an inclusion ($\Sigma' \subseteq \Sigma$). Then, $A' \models_{\sim_{\Sigma}, \simeq_{\Sigma}}^{\sigma(Test(SP))} \sigma(TAx(SP))$, as $\sigma(Test(SP)) = Test(SP)$ and $\sigma(TAx(SP)) = TAx(SP)$. By 1(a) and Theorem 1, $A'|_{\Sigma'} \models_{\sim_{\Sigma'}, \simeq_{\Sigma'}}^{Test(SP)} TAx(SP)$. (2.) Similar to item 1. $\square$

Assumption 2(a), in Lemma 3, whereby SP' cannot have axioms with symbols in S' and F' , is a strong restriction. However, one could see this assumption as requiring that specifications should be transformed such that hidden axioms are replaced by their visible consequences.

5 Structured Testing *versus* Flat Testing

Structured testing of a Σ -algebra A against a structured specification SP corresponds to membership in the class of checkable models of SP , i.e., $A \in ChMod_{\sim, \simeq}(SP)$, whereas *flat testing* corresponds to the testing satisfaction of visible axioms of SP , i.e., $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$. The former is based on the structure of SP and may consist of more than one test with specific test sets applied for specific groups of axioms (see e.g. Definition 8) and may demand additional sorts and operations not in A , due to hiding. On the other hand, flat testing is a monolithic experiment based on an unstructured view of the specification without considering hidden elements and using a single test set $Test(SP)$

which is deduced from SP and applied for all axioms in $TAx(SP)$. In Example 1, flat testing from $IntList$ corresponds to the test $A \models_{\sim, \simeq}^{Test(IntList)} TAx(IntList)$, whereas structured testing ($A \in ChMod_{\sim, \simeq}(IntList)$) corresponds to the following tests $A'|_{List} \models_{\sim, \simeq}^{T_1} \Phi_1$ and $A' \models_{\sim, \simeq}^{T_2} \Phi_2$, where A be an $IntList$ -algebra, Φ_1 and Φ_2 are the axioms of $List$ and $SList2$ respectively and A' is a $SList$ -algebra with $A = A'|_{IntList}$. Advantages of structured versus flat testing are presented throughout this section. For instance, interpretation of flat testing cannot always be given for structured specifications. Therefore, specifications may be required to be consistent w.r.t. test sets and/or transparent as follows.

Definition 11 (Transparent Specification). *Let SP be a structured specification. SP is transparent if it has no occurrences of $hide$ in its structure.*

Definition 12 (Consistent Specification). *Let SP be a structured specification. If for every specification $SP_1 \cup SP_2$ which is part of SP , $Test(SP_1) \upharpoonright_S = Test(SP_2) \upharpoonright_S$, then SP is consistent w.r.t. test sets, where S is the intersection of the sets of sorts of bound variables in $TAx(SP_1)$ and $TAx(SP_2)$.*

In the sequel, we present some theorems which relate structured/flat testing to a notion of correctness w.r.t. the class of real models. Let SP be a structured specification with $\Sigma = Sig(SP)$ and A be a Σ -algebra.

5.1 Structured Testing and Correctness

Theorem 5 is a generalisation of Theorem 3 for structured specifications. Under certain conditions incorrect programs can be accept by structured testing, that is, not every checkable model is a real model, but any real model is a checkable model. The assumptions that families of equalities need to be complete (sound)³ seem to be strong, but we can consider only signatures arising in the structure of SP when defining these families.

Theorem 5. *If $\sim$ is complete, $\simeq$ is sound, and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$ then $A \in Mod_{\approx}(SP)$ implies $A \in ChMod_{\sim, \simeq}(SP)$.*

Proof. By induction on the structure of SP .

Theorem 6 below is a generalisation of 4 for structured specifications and the dual of 5. This theorem shows that under the conditions stated, real models can be rejected by structured testing, but any checkable model is a real model. Again, the assumptions on quantifiers of Theorems 5 and 6 can be dropped if SP has only unbiased test sets and valid test sets respectively.

Theorem 6. *If $\sim$ is sound, $\simeq$ is complete and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$ then $A \in ChMod_{\sim, \simeq}(SP)$ implies $A \in Mod_{\approx}(SP)$.*

³ $\sim$ is complete (sound) w.r.t $\approx$ iff $\forall \Sigma \in Sign, \sim_{\Sigma}$ is complete (sound) w.r.t. $\approx_{\Sigma}$.

Example 1 matches Theorem 5 since axioms of *IntList* have only positive $\forall$ and negative $\exists$. Given $\sim$ complete and $\simeq$ sound, any *IntList*-algebra A can be checked against *IntList* by structured testing in the sense that an interpretation can be given for test results based on Theorem 5, no matter how T_1 and T_2 are defined. In this case, we need to consider only the signatures S_1 and S_2 in the families $\sim$ and $\simeq$. In practice, an oracle for *IntList* can be defined as a procedure which drives the tests required. Note that hidden axioms are taken into account and an implementation of *is_sorted* is needed.

Interesting properties of specifications may be expressed by hidden axioms built from visible and hidden symbols (see *sort* in Example 1) and these axioms cannot always be replaced by their visible consequences. According to Definition 10, in order to check hidden axioms by structured testing, an implementation of hidden symbols may be required for each occurrence of *hide*, which is a reasonable constructive requirement. In a similar way to structured testing, compositional proof systems for implementations of structured specifications may require a persistent extension SP'_i of SP_i to be checked against SP' in order to prove that SP_i is an implementation of *hide sorts* S' *opns* F' in SP' [9]. Also, [6] shows that under those circumstances the visible part of the specification can be proved by importing the hidden part into the implementation. One drawback of testing hidden axioms is that the implementation of hidden symbols may itself have errors which might lead to wrong conclusions regarding correctness. However, the main goal of testing is to uncover errors so that a debugger can be used to pinpoint them. Efforts to provide an implementation of hidden symbols which conforms to their specification are compensated by the fact that this can be used to check different implementations of the visible part. Moreover, it is likely that these hidden symbols represent standard components and functions already implemented and verified. Another drawback is that the specification of hidden symbols might not have an implementation (see Example in [10]). In this case, unless the specification can be transformed, only flat testing can be applied.

5.2 Structured Testing and Flat Testing

If SP is a structured specification, structured testing does not always coincide with flat testing. However, the following theorems show special cases where either one or the other direction of the implication may hold.

Theorem 7. *Let SP be consistent w.r.t. test sets. If $\sim$ and $\simeq$ are compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in \text{ChMod}_{\sim, \simeq}(SP)$ implies $A \models_{\sim, \simeq}^{Test(SP)} TA_x(SP)$.*

Proof. By induction on SP . From Definition 5, Lemmas 1(1.), 2(1.) and 3(1.).

Theorem 8 below, the dual of 7, require SP to be transparent.

Theorem 8. *Let SP be transparent and consistent w.r.t. test sets. If $\sim$ and $\simeq$ are compatible and the axioms of SP have only negative occurrences of $\forall$ and positive of $\exists$, then $A \models_{\sim, \simeq}^{Test(SP)} TA_x(SP)$ implies $A \in \text{ChMod}_{\sim, \simeq}(SP)$.*

Equivalence of structured and flat testing require specifications to be consistent w.r.t. test sets and transparent, families of equalities to be compatible and also contradictory assumptions on quantifiers. According to Theorem 7, in Example 1, structured testing implies flat testing if T_1 and T_2 coincide and the families $\sim$ and $\simeq$ are compatible at least for S_1 , S_2 and $Sig(IntList)$. Obviously, Theorem 8 cannot be used, since $IntList$ has positive $\forall$ and is not transparent.

5.3 Flat Testing and Correctness

The definition of real models relies on the compositional semantics of specifications. Therefore, in contrast to structured testing, further assumptions on specifications and equalities are required to identify whether a real model is accepted by flat testing and vice-versa.

Theorem 9. *Let SP be consistent w.r.t. test sets. If $\sim$ is complete and compatible and $\simeq$ is sound and compatible and the axioms of SP have only positive occurrences of $\forall$ and negative occurrences of $\exists$, then $A \in Mod_{\sim}(SP)$ implies $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$.*

Proof. Follows by Theorems 5 and 7.

Again to obtain a result from flat testing in Example 1, according to Theorem 9, no great flexibility is given to define T_1 and T_2 , since $IntList$ must be consistent and also equalities on $Sig(IntList)$ must belong to compatible families. For the consistency requirement, $(T_2)_{List}$ cannot have any ground term like $sort(l)$, where l is a list. This would allow tests like $head(cons(x, sort(l))) = x$ to be performed, where the $sort$ operation is somehow exercised. Thus, also because hidden axioms cannot be considered and $sort$ is defined in terms of is_sorted , the operation $sort$ cannot be checked by flat testing. Structure testing is the only way of effectively verifying this operation.

Theorem 10. *Let SP be consistent w.r.t. test sets and transparent. If $\sim$ is sound and compatible and $\simeq$ is complete and compatible and the axioms of SP have only negative occurrences of $\forall$ and positive occurrences of $\exists$, then $A \models_{\sim, \simeq}^{Test(SP)} TAx(SP)$ implies $A \in Mod_{\sim}(SP)$.*

Proof. Follows by Theorems 8 and 6.

Due to the hide operation, in order to check all axioms of a structured specification SP by flat testing, this specification may need to be normalised by a process similar to normalisation of ordinary specifications [2] to give a specification $nf(SP) = \text{hide sorts } S' \text{ opns } F' \text{ in } \langle \Sigma', \Phi', T' \rangle$, which is equivalent to SP w.r.t. signature equivalence and model class equality. If such a normal form $nf(SP)$ can be computed, in order to test A against $nf(SP)$ (and consequently, SP), it would be necessary to test $A + \Delta(A)$ against $\langle \Sigma', \Phi', T' \rangle$, where $\Delta(A)$ implements the hidden sorts and functions. It is easy to check that whenever the assumptions of Theorem 10 hold, $A + \Delta(A) \models_{\sim, \simeq}^{T'} \Phi'$ implies $A + \Delta(A) \in Mod_{\sim}(\langle \Sigma', \Phi', T' \rangle)$

and then, by Definition 10, $A \in Mod_{\approx}(nf(SP))$. Normalisation of specifications with testing interface still needs to be defined. However, the complexity of constructing such a normal form for large specifications and the loss of structure in the test experiment itself might put one off [9,2]. Also, in order to guarantee equality of classes of real and checkable models, it may only be possible to apply normalisation to specifications with testing interface which are consistent w.r.t test sets and where the families of equalities considered are compatible.

6 The Observational Case

If we restrict behavioural equalities to be observational equalities (as defined in section 2), then we have a family $\approx_{Obs} = (\approx_{Obs, \Sigma})_{\Sigma \in Sign}$, where $\approx_{Obs, \Sigma} = (\approx_{Obs, \Sigma, A})_{A \in Alg(\Sigma)}$ is a Σ -observational equality with one observational equality for each Σ -algebra A . In this section, we focus on observational equalities and show how families of Σ -approximate equalities can be defined so that the theorems of section 5 hold. By Proposition 2, $\approx_{Obs}$ is reduction-compatible.

For structured testing, we need to define a complete and a sound family of approximate equalities w.r.t. $\approx_{Obs}$. Let $\approx = (\approx_{\Sigma})_{\Sigma \in Sign}$ be a family of contextual equalities built from the set of crucial contexts⁴ as in [3]. It is easy to check that the family $\approx$ is complete w.r.t. $\approx_{Obs}$. In Example 1, given that *list* is a non-observable sort and *int* and *bool* are observable sorts, S_1 has *head*(z) as crucial context, whereas S_2 has *head*(z) and *is_sorted*(z). Furthermore, let $\simeq = (\simeq_{\Sigma})_{\Sigma \in Sign_{Obs}}$ be the literal equality on the values of the Σ -algebras. This family is clearly sound w.r.t. $\approx_{Obs}$. The following is a corollary of Theorem 5.

Corollary 1. *If the axioms of SP have only positive $\forall$ and negative $\exists$, then $A \in Mod_{\approx_{Obs}}(SP)$ implies $A \in ChMod_{\approx, \simeq}(SP)$.*

For flat testing, from Theorem 9, we need to define *compatible* families of equalities. Note that $\approx$ is not translation-compatible. In Example 1, this is due to the observer *is_sorted* which is added in S_2 . Let $\approx$ protect encapsulation, i.e., for all $\sigma : \Sigma' \rightarrow \Sigma$ arising in SP , Σ does not introduce new observations for sorts in Σ' . In Example 1, this means that *is_sorted* must be defined in S_1 instead of S_2 . Clearly, this restricted $\approx$ is compatible. By Proposition 1, $\simeq$ is compatible.

7 Concluding Remarks

Not all testing experiments are really interesting unless a decision procedure to interpret their results can be given – this is the so-called oracle problem. In order to overcome the oracle problem, restrictions on both specifications and test suites may be needed. For instance, the use of sound/complete approximate equalities and assumptions on the presence/absence of quantifiers may be required for testing from flat specifications expressed in first-order logic. In this paper, we present

⁴ Contexts of the form $f(x_1, \dots, x_{k-1}, z_{s_k}, x_{k+1}, \dots, x_n)$, with $f \in F$ of arity $s_1 \dots s_{k-1} s_k s_{k+1} \dots s_n \rightarrow s$, $s \in Obs$ and $s_k \in S \setminus Obs$.

a framework for testing from structured algebraic specifications and show that structured specifications introduce further complications to the oracle problem. Families of equalities might be needed for tackling different parts of the specification and they might be required to be compatible with signature morphisms. Also, different test sets for different parts of a structured specification can be defined. Contrary to flat testing, provided that families of approximate equalities which are sound/complete can be defined, we generalise previous results on the use of approximate equalities for structured testing without any additional restriction since structured testing is based on the compositional semantics of specifications. However, structured testing demands an implementation of hidden symbols which is a reasonable requirement, but cannot always be met. Flat testing is more limited than structured testing. For instance, specifications are required to be consistent w.r.t. test sets which may impede certain tests to be performed. Moreover, these specifications may need to be transparent, that is, have no occurrences of *hide*, unless either hidden axioms can be replaced by their visible consequences or the specification can be normalised, which may be complicated for large specifications, not to mention the loss of structure. Finally, flat testing can only be interpreted if the equalities considered belong to families which are compatible w.r.t. signature morphisms. In the observational case, signatures cannot add observers for sorts previously introduced in other signatures. As further work, we aim at taking parameterised programs into account. This may naturally lead to a method for performing integration testing. Moreover, we aim to use testing together with proofs in a stepwise formal development. Substantial case studies and assistant tools are also to be constructed.

8 Acknowledgments

The author wishes to thank specially Don Sannella for important discussions and also the referees for comments which helped to improve this paper. This research has been partially supported by EPSRC grant GR/K63795.

References

1. G. Bernot. Testing Against Formal Specifications: a Theoretical View. In S. Abramsky and T.S.E. Maibaum, eds., *TAPSOFT'91*, LNCS 494. Springer, 1991.
2. M. Bidoit, M. V. Cengarle, and R. Hennicker. *Proof Systems for Structured Specifications and Their Refinements*, chapter 11. IFIP Reports. Springer, 1999.
3. M. Bidoit and R. Hennicker. Behavioural Theories and the Proof of Behavioural Properties. *Theoretical Computer Science*, 165(1):3–55, 1996.
4. R. Burstall and J. Mckinna. Deliverables: an Approach to Program Development in the Calculus of Constructions. ECS-LFCS-91-133, Edinburgh University, 1992.
5. J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-based Specifications. In J. Wookcook and P. Larsen, eds., *FME'93*, LNCS 670. Springer, 1993.
6. J. Farrés-Casals. Proving correctness w.r.t. specifications with hidden parts. In H. Kirchner and W. Wechler, eds., *Alg. and Logic Prog.*, LNCS 463. Springer, 1990.

7. P. Le Gall and A. Arnould. Formal Specification and Test: Correctness and Oracle. In M. Haverdaen, O. Owe, and O. Dahl, eds., *Recent Trends in Data Type Specification*, LNCS 1130. Springer, 1996.
8. M. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, eds., *TAPSOFT'95*, LNCS 915. Springer, 1995.
9. R. Hennicker. *Structured Specifications with Behavioural Operators: Semantics, Proof Methods and Applications*. Habilitation thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, München, Germany, june 1997.
10. Stefan Kahrs and Donald Sannella. Reflections on the design of a specification language. In E. Astesiano, ed., *FASE'98*, volume 1382 of LNCS. Springer, 1998.
11. P. D. L. Machado. On Oracles for Interpreting Test Results against Algebraic Specifications. In A. Haeberer, ed., *AMAST'98*, LNCS 1548. Springer, 1999.
12. D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comp. Syst.*, 13(4):365–398, 1995.
13. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675–788. Elsevier, 1990.

Author Index

- Adi, K., 499
Ancona, D., 454
Andova, S., 323
Astesiano, E., 273
- Börger, E., 293
Braga, de O., C., 407
Breitling, M., 11
Broy, M., 89
Busi, N., 198
- Cavarra, A., 293
Cerioli, M., 273
Cerone, A., 57
Choppy, C., 165
Coudert, S., 438
- Debbabi, M., 499
De Nicola, R., 339
Desharnais, J., 355
Doche, M., 514
Durán, F., 422
- Erwig, M., 135
- Fidge, C.J., 213
Fischer, C., 469
Focardi, R., 258
Froebel, P., 73
- Gibbons, J., 371
Gorrieri, R., 198, 258
Guerra, S., 26
Gurevich, Y., 197
- Haeusler, E.H., 407
Healy, M., 484
Heinemann, B., 41
Hill, P.M., 181
- Jacobs, B., 150
- Lawford, M., 73
Le Gall, P., 438
Loreti, M., 339
- Machado P.D.L, 529
Maggiolo-Schettini, A., 228
Martinelli, F., 258
McDougall, J., 73
Mejri, M., 499
Meseguer, J., 407
Möller, B., 355
Moor, de, O., 371
Mosses, P.D., 407
Moum, G., 73
- Parnas, D.L., 1, 9
Philipps, J., 11
Poizat, P., 165
Poll, E., 150
- Reggio, G., 243, 273
Repetto, L., 243
Riccobene, E., 293
Royer, J.-C., 165
- Spoto, F., 181
- Talcott, C.L., 391
Tchier, F., 355
Tini, S., 228
- Van Wyk, E., 119
- Wehrheim, H., 469
Wiels, V., 514
Williamson, K., 484
Wing, J.M., 2
Wirsing, M., 89, 309
- Ying, M., 309
- Zavattaro, G., 198