Lecture Notes in Computer Science 3407

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
New York University, NY, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Zhiming Liu Keijiro Araki (Eds.)

Theoretical Aspects of Computing – ICTAC 2004

First International Colloquium
Guiyang, China, September 20-24, 2004
Revised Selected Papers

Springer

Volume Editors

Zhiming Liu
United Nations University
International Institute for Software and Technology
UNU-IIS, Macao SAR, China
E-mail: z.liu@iist.unu.edu

Keijiro Araki
Kyushu University
Department of Computer Science and Communication Engineering
Graduate School of Information Science and Electrical Engineering
6-10-1 Hakozaki, Higashi-ku, Fukuoka, 812-8581 Japan
E-mail: araki@csce.kyusyu-u.ac.jp

Library of Congress Control Number: 2005921895

CR Subject Classification (1998): F.1, F.3, F.4, D.3, D.2, C.2.4

ISSN 0302-9743
ISBN 3-540-25304-1 Springer 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. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11407058 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of ICTAC 2004, the 1st International Colloquium on Theoretical Aspects of Computing, which was held in Guiyang, China on 20–24 September 2004.

ICTAC was founded by the International Institute for Software Technology of the United Nations University (UNU-IIST). Its aim is to bring together practitioners and researchers from academia, industry, and government to present research results, and exchange experience, ideas, and solutions for their problems in theoretical aspects of computing. The geographic focus of the ICTAC events is on developing countries to help to strengthen them in their research, teaching, and development in computer science and engineering, to encourage research cooperation among developing countries, and to improve the links between developing countries and industrial countries.

The Program Committee of ICTAC 2004 received 111 submissions from over 30 countries and regions. Each paper was reviewed, mostly by at least three referees working in relevant fields, but by two in a few cases. Borderline papers were further discussed during an online meeting of the Program Committee. Thirty-four papers were accepted based on originality, technical soundness, presentation and relevance to software engineering and formal methods. We sincerely thank all the authors who submitted their work for consideration. We thank the Program Committee members and the other referees for their great effort and professional work in the reviewing and selecting process. Their names are listed on the following pages. In addition to the contributed papers, the proceedings also includes contributions from the invited speakers: José Luiz Fiadeiro, He Jifeng, Huimin Lin and Rustan Leino.

Six very good tutorials were selected as affiliated events of ICTAC 2004. We express our thanks to all those who submitted tutorial proposals. We have also included the abstracts of the tutorials in the proceedings.

We thank the Organizing Committee Chair, Danning Li, the Finance Chair, Dan Li, and the Publicity Chair, Bernhard Aichernig, for their great collaborative effort and hard work that made the event so successful and enjoyable. We are truly grateful to the Advisory Committee members for their advice and suggestions. We particularly thank Kitty Iok Sam Chan and Anna UnLai Chan of UNU-IIST for their hard work in maintaining the conference administration system. All the members of staff at UNU-IIST helped in many ways. In particular, the Acting Director, Chris George, actively supported the organization of ICTAC.

ICTAC 2004 was organized and sponsored by UNU-IIST and the Academy of Sciences of Guizhou Province, China.

December 2004 Keijiro Araki and Zhiming Liu

Organization

ICTAC 2004 was organized by the International Institute for Software Technology of the United Nations University (UNU-IIST) and Guizhou Academy of Sciences, China.

Conference Chairs

Program Co-chairs	Keijiro Araki (Kyushu University, Japan) Zhiming Liu (UNU-IIST, Macau SAR, China)
Organizing Committee Chair	Danning Li (Guizhou Academy of Sciences)
Finance Chair	Dan Li (Guizhou Academy of Sciences)
Publicity Co-chairs	Bernhard K. Aichernig (UNU-IIST, Macau SAR, China) Dan Li (Guizhou Academy of Sciences)

Advisory Committee

Dines Bjørner	Technical University of Denmark, Denmark
Manfred Broy	Technische Universität München, Germany
José Luiz Fiadeiro	University of Leicester, UK
Jifeng He	UNU-IIST, Macau SAR, China
Shaoying Liu	Hosei University, Japan
Zhiming Liu	UNU-IIST, Macau SAR, China
Jim Woodcock	York University, UK

Program Committee

Luis S. Barbosa	University of Minho, Portugal
Gabriel Baum	National University of La Plata, Argentina
Hubert Baumeister	LMU, Munich, Germany
Jonathan Bowen	London South Bank University, UK
Cristian S. Calude	University of Auckland, New Zealand
Ana Cavalcanti	University of York, UK
Yifeng Chen	University of Leicester, UK
Wei Ngan Chin	NUS, Singapore
Jim Davies	Oxford University, UK
Henning Dierks	University of Oldenburg, Germany
Jin Song Dong	NUS, Singapore
Wan Fokkink	CWI, the Netherlands
Susanne Graf	VERIMAG, France

Michael R. Hansen	DTU, Denmark
James Harland	RMIT University, Australia
Jozef Hooman	Embedded Systems Institute, Eindhoven, the Netherlands
Guoxing Huang	ECNU, Shanghai, China
Purush Iyer	North Carolina State University, USA
Ryszard Janicki	McMaster University, Ontario, Canada
Michael Johnson	Macquarie University, Sydney, Australia
Fabrice Kordon	University of Paris VI, France
Maciej Koutny	University of Newcastle upon Tyne, UK
Kung-Kiu Lau	Manchester University, UK
Antonia Lopes	University of Lisbon, Portugal
Jian Lu	Nanjing University, China
Huaikou Miao	Shanghai University, China
Paritosh Pandya	TIFR, Mumbai, India
Zongyan Qiu	Peking University, Beijing, China
Anders P. Ravn	Aalborg University, Denmark
Gianna Reggio	University of Genoa, Italy
Riadh Robbana	LIP2/EPT, Tunisia
Augusto Sampaio	Federal Univ. of Pernambuco, Recife, Brazil
Bernhard Schätz	TU München, Germany
Andrea Maggiolo-Schettini	University of Pisa, Italy
Irek Ulidowski	University of Leicester, UK
Miroslav Velev	Carnegie Mellon University, USA
Ji Wang	National Laboratory for Parallel and Distributed Processing, China
Wang Yi	Uppsala University, Sweden
Jian Zhang	Institute of Software, CAS, China
Mingyi Zhang	Guizhou Academy of Sciences, China
Weiqing Zhang	SWNU, Chongqing, China
Hongjun Zheng	Semantics Designs Inc., USA

External Referees

The Program Committee members and the external referees listed below did an excellent job in reviewing an unexpectedly big number of submissions under the pressure of a very tight deadline.

Referees

Farhad Arbab	Egidio Astesiano	Richard Banach
Marco Bellia	Adel Benzina	Sergey Berezin
Eike Best	Roderick Bloem	Chiara Bodei
Roberto Bruni	Marzia Buscemi	Elena Calude
Jacques Carette	Jessica Chen	Yihai Chen

Michael J. Dinneen	Perla Velasco Elizondo	Ramanathan Guha
Shui-Ming Ho	Alan Jeffrey	Qingguang Ji
R.K. Joshi	Wolfram Kahl	George Karakostas
Hanna Klaudel	Alexander Knapp	John Knudsen
Piotr Kosiuczenko	Moez Krichen	Mojmír Křetínský
	Mark Lawford	Ryan Leduc
Guangyuan Li	Donggang Liu	Jing Liu
Xinxin Liu	Quan Long	Jinwen Ma
Victor Marek	Marius Minea	Nebut Mirabelle
Isabel Nunes	G. Paun	Alessandra Di Pierro
Geguang Pu	Shengchao Qin	Thomas Quillinan
Hasna Riahi	James Riely	David Rydeheard
Andrei Sabelfeld	Jonathan Shapiro	Jeremy Sproston
Andrei Stefan	Jason Steggles	Jing Sun
S.P. Suresh	Tayssir Touili	Vasco T. Vasconcelos
Tiejun Wang	Heike Wehrheim	Rui Xue
Hans Zantema	Jianjun Zhao	Belhassen Zouari

Table of Contents

Invited Speakers

Software Services: Scientific Challenge or Industrial Hype?
José Luiz Fiadeiro 1

Integrating Variants of DC
He Jifeng, Jin Naiyong 14

Challenges in Increasing Tool Support for Programming
K. Rustan M. Leino 35

A Predicate Spatial Logic and Model Checking for Mobile Processes
Huimin Lin 36

Concurrent and Distributed Systems

Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes
Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, Martin Steffen 37

Specifying Software Connectors
Marco Antonio Barbosa, Luís Soares Barbosa 52

Replicative - Distribution Rules in P Systems with Active Membranes
Tseren-Onolt Ishdorj, Mihai Ionescu 68

A Generalisation of a Relational Structures Model of Concurrency
Ryszard Janicki 84

A Logical Characterization of Efficiency Preorders
Neelesh Korade, S. Arun-Kumar 99

Inherent Causal Orderings of Partial Order Scenarios
Bill Mitchell 113

Atomic Components
Steve Reeves, David Streader 128

Towards an Optimization-Based Method for Consolidating Domain Variabilities in Domain-Specific Web Services Composition
Jun-Feng Zhao, Lu Zhang, Ya-Sha Wang, Ying Jiang, Bing Xie 140

Model Integration and Theory Unification

A Formal Framework for Ontology Integration Based on a Default Extension to DDL
Yinglong Ma, Jun Wei, Beihong Jin, Shaohua Liu 154

A Predicative Semantic Model for Integrating UML Models
Jing Yang, Quan Long, Zhiming Liu, Xiaoshan Li 170

An Automatic Mapping from Statecharts to Verilog
Viet-Anh Vu Tran, Shengchao Qin, Wei Ngan Chin 187

Reverse Observation Equivalence Between Labelled State Transition Systems
Yanjun Wen, Ji Wang, Zhichang Qi 204

Program Reasoning and Testing

Minimal Spanning Set for Coverage Testing of Interactive Systems
Fevzi Belli, Christof J. Budnik 220

An Approach to Integration Testing Based on Data Flow Specifications
Yuting Chen, Shaoying Liu, Fumiko Nagoya 235

Combining Algebraic and Model-Based Test Case Generation
Li Dan, Bernhard K. Aichernig 250

Verifying OWL and ORL Ontologies in PVS
Jin Song Dong, Yuzhang Feng, Yuan Fang Li 265

Verification

Symbolic and Parametric Model Checking of Discrete-Time Markov Chains
Conrado Daws 280

Verifying Linear Duration Constraints of Timed Automata
Pham Hong Thai, Dang Van Hung 295

Idempotent Relations in Isabelle/HOL
Florian Kammüller, J.W. Sanders 310

Program Verification Using Automatic Generation of Invariants
Enric Rodríguez-Carbonell, Deepak Kapur 325

Theories of Programming and Programming Languages

Random Generators for Dependent Types
Peter Dybjer, Haiyan Qiao, Makoto Takeyama 341

A Proof of Weak Termination Providing the Right Way to Terminate
Olivier Fissore, Isabelle Gnaedig, Hélène Kirchner 356

Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn
Silvio Ranise, Christophe Ringeissen, Duc-Khanh Tran 372

Real Time Reactive Programming in Lucid Enriched with Contexts
Kaiyu Wan, Vasu Alagar, Joey Paquet 387

Revision Programs with Explicit Negation
Yisong Wang, Mingyi Zhang 403

Real-Time and Co-design

An Algebraic Approach for Codesign
Marc Aiguier, Stefan Béroff, Pierre-Yves Schobbens 415

Duration Calculus: A Real-Time Semantic for B
Samuel Colin, Georges Mariano, Vincent Poirriez 431

An Algebra of Petri Nets with Arc-Based Time Restrictions
Apostolos Niaouris 447

A Calculus for Shapes in Time and Space
Andreas Schäfer 463

A Framework for Specification and Validation of Real-Time Systems Using *Circus* Actions
Adnan Sherif, He Jifeng, Ana Cavalcanti, Augusto Sampaio 478

Automata Theory and Logics

Switched Probabilistic I/O Automata
Ling Cheung, Nancy Lynch, Roberto Segala, Frits Vaandrager 494

Decomposing Controllers into Non-conflicting Distributed Controllers
Padmanabhan Krishnan 511

Reasoning About Co–Büchi Tree Automata
Salvatore La Torre, Aniello Murano 527

Foundations for the Run-Time Monitoring of Reactive Systems - *Fundamentals of the MaC Language*
Mahesh Viswanathan, Moonzoo Kim 543

Tutorials at ICTAC 2004

A Summary of the Tutorials at ICTAC 2004
Zhiming Liu 557

Author Index 561

Software Services: Scientific Challenge or Industrial Hype?

José Luiz Fiadeiro

Department of Computer Science, University of Leicester,
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

Abstract. Web-services keep making headlines, not only in technical journals but also in the wider media like The Economist. Is this just a sales plot of the fragile software industry targeted to the companies and organisations that want to operate in the new economy as enabled by the internet and wireless communication? Or is there a new paradigm as far as software development is concerned? Should we, scientists, regard this as a challenge? Or dismiss it as hype? In this paper, we analyse these questions in the context of the notions of complexity that arise in software development and the methods and techniques that can be offered to address them.

1 Introduction

component *(n): a constituent part*
complex *(a): composed of two or more parts*
architecture *(n):*
1 *: formation or construction as, or as if, the result of conscious act;*
2 *: a unifying or coherent form or structure*

Hardly anybody working in computer science or software engineering can claim to be immune to the hype that surrounds "web services". However, in spite (or because…) of all the frenzy, it is not clear whether there is any room for a real scientific discussion. After all, it is the big companies that have been driving most of the activity in this area. This is why many people in academia and research are asking if this isn't just a sales plot of the software industry targeted to the companies and organisations that want to operate in the internet… Is there really a new paradigm as far as software development is concerned? Should we, scientists, regard this as a challenge? Or dismiss it as mere industrial hype?

One of arguments made in favour of a new discipline, and a line that one could envisage pursuing in a scientific debate, opposes "components" to "services". However, the term "component" is being used more and more frequently in software engineering, at the expense of conveying less and less meaning. People are also drawing analogies with the use of these concepts in arts, science, and engineering without a clear sense of purpose. This is raising more confusion and less confidence in the usage of methods and tools being advertised for component-based development or software architecture design.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 1–13, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Nevertheless, bringing the term component into the arena is important because it points a finger to one of the crucial dimensions of (software) engineering: *complexity*. Decomposition of a problem into sub-problems is the best (only?) way that humans have found to tackle complexity. Every discipline of decomposition leads to, or is intrinsically based on, a notion of component and composition: *ça va de soi*! The way we decompose a problem, or the discipline that we follow in the decomposition, further leads to an architecture, or architectural style, that identifies the way the problem is structured in terms of sub-problems and the mechanisms through which they relate to one another.

This remark can hardly be classified as a deep insight but the fact is that the differences that we can witness in the use of the terms "software component" and "software architecture" can be attributed to the simple fact they address different notions of complexity that arise in the process of engineering software systems. Hence, can we frame the debate on services in the context of complexity? Are services the components of a new architectural approach? If so, for which notion of complexity?

In our opinion, any answer to this question requires an analysis of the forces that have made software development practice evolve over the past 50 years or so, as well as the recognition of some of the fundamental milestones of this evolution. The purpose of this paper is, precisely, to guide the reader through a journey in the history of software engineering, hoping that, at the end, a clear case for a new paradigm will have emerged. In this process, we will make use of the following figure borrowed from [24]:

2 In the Beginning…

Fig. 1 makes clear the fact that, in the early days, software development took place "in-the-head" of a person (a term that we prefer to "any-which-way"). That person had a problem that, typically, consisted of some complex computation needed to obtain some important result (e.g. the next best move during a game of chess) that the person would consume upon termination. To solve that problem, the person would develop a program to run on a particular machine.

2.1 From Programming "in-the-head" to "in-the-small"

Programming took place "in-the-head" in the sense that it did not concern anybody else except the programmer: only the results of the execution were needed, the program being just a means to that end. The program thus built would reflect very closely the architecture of the machine available to run it. The programmer would often have to resort to all sorts of "tricks" to get around the limitations of memory and speed. This is why it seems unjust to qualify this activity as programming "any-which-way" as it often required a deep knowledge of the target machine. In any case, programming was a one-off activity best performed by virtuosi in absolute control of the execution infrastructure and with the final result of the execution as the primary goal of the activity.

This changed when, instead of the result of the execution, the programmer had the solution (as embodied in the program itself) as a business goal. For instance, instead

1960 ± 5 Programming-any-which-way	*1970 ± 5 Programming-in-the-small*	*1980 ± 5 Programming-in-the-large*	*1990 ± 5 Programming-in-the-world*
Mnemonics, precise use of prose	Simple input-output specifications	Systems with complex specifications	Distributed systems with open-ended, evolving specs
Emphasis on small programs	Emphasis on algorithms	Emphasis on system structure, management	Emphasis on subsystem interactions
Representing structure, symbolic information	Data structures and types	Long-lived databases	Data & computation independently created, come and go
Elementary understanding of control flow	Programs execute once and terminate	Program systems execute continually	Suites of independent processes cooperate

Fig. 1

of a chess fanatic developing a program for his pocket calculator to compute the next move on a given configuration, we are now talking of a scenario in which a chess-playing program is developed to be sold to clients who will run it themselves on their machines for their own purposes. A crucial landmark is thus reached: programs, instead of the results of their executions become the goods. In other words, software becomes a product.

In order to make commercial sense, it is essential to develop programs that can be run in different machines. Programming becomes an activity that cannot be purely conducted "in-the-head" as it needs to take into account that the resulting software is to be commercialised. The separation between program and code executable on a particular computer is supported by machine-independent programming languages and compilers. In fact, this separation consists of an abstraction step in which the program written by the programmer is seen as a higher-level abstraction of the code that runs on the machine.

A crucial aspect of this abstraction process is the ability to work with data structures that do not necessarily mirror the organisation of the memory of the machine in which the code will run. This process can be taken even further by allowing the data structures to reflect the organisation of the solution to the problem, even if they are not available in the target programming language. Specification languages support the definition of such data structures and the high-level programs that use them.

Program development methodologies [6,] further address the problem of developing "real" programs from such high-level descriptions. They help the software developer arrive to a solution to the original problem regardless of the fact that the resulting program is for self-consumption of for sale. For instance, in the 70s, so-called *structured programming* provided abstractions for controlling execution that introduced a totally new discipline into software development by separating control flow from the text of programs. Before, control flow was largely defined in terms of GOTO state-

ments that transfer execution to a label in the program text. Structured programming provides constructs such as "if-then-else" and "while-do" for creating a variety of control execution patterns that can be understood independently of the order in which the program text is written.

2.2 Program Architectures

Through methods supporting programming in-the-small we are led to notions of program component and architecture that allow us to tackle the complexity of controlling the flow of execution. In Fig. 2, we present the architecture of a run length encoder[1] in JSP [15], one of the methods that introduced structured programming. In JSP, program development follows a top-down approach in the sense that blocks are identified and put together according to given control structures (sequential composition, iteration, etc). Each block is then developed in the same way, independently of the other blocks. The criteria for decomposition derive from the structure of the data manipulated by the program.

The advantage of this architectural representation is that it decomposes control flow according to the structure of the input data into well-identified components. Each of these can be individually programmed and put together using the primitives of the specific programming language that is chosen for a particular implementation. Different methodologies lead to different architectures, of course.

In what concerns the software industry, it is clear that programming methodologies have a significant impact in the delivery time and cost of the final product. By allowing the programmer to work at higher levels of abstraction, results from the theory of algorithms and complexity can be used for controlling performance in space and time, which is an important factor of quality. When seconded by mathematical semantics, a method and associated language can even assist the proof of the correctness of the product with respect to a high-level specification of its functionality as given, for instance, through input/output specifications. This further adds to the quality of the final product.

It is not our purpose in this paper to promote any specific such language and method, especially because the debate on what consists good support for program construction is not closed and new methods/languages keep being proposed. Nevertheless, we would like to mention *artificial intelligence* as a methodology that provides abstractions for programming that derive from the way humans solve problems, and *object-oriented programming* as a discipline based on the packaging of data and functionality together into units called objects.

Finally, we would like to point out that we have been discussing abstractions for handling the complexity of solving a given problem in terms of a computer program. In this activity, the complexity lies more in the nature of the problem that needs to be understood and the process of coding it than in the resulting solution (application). For instance, programs that play chess, unlike some of their mechanical ancestors, are

[1] A run length encoder is a program that takes as input a stream of bytes and outputs a stream of pairs consisting of a byte along with a count of the byte's consecutive occurrences in the input stream.

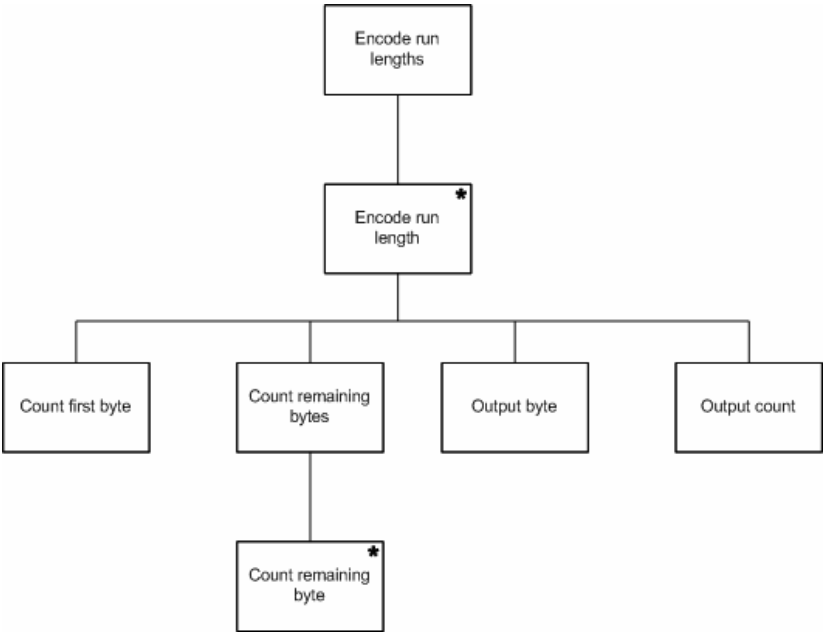


Fig. 2

not known for the complexity of their structure in terms of the modules/parts that they are assembled from. They give headaches to coders, not to project managers and their maintenance groups. Indeed, chess playing is seen more as a testing ground for artificial intelligence than software engineering.

3 The "Software Crisis"

In September 1994, an article in the Scientific American alerted to the "Software's chronicle crisis": software was recognised to be manufactured as in a "cottage industry" and not according to the "industrial standards of mass production and reliability. The article identifies the underlying problem as being one of "complexity":

> *The challenge of complexity is not only large but also growing. […]. To keep up with such demand, programmers will have to change the way that they work. "You can't build skyscrapers using carpenters," Curtis quips.*

3.1 Programming in-the-large

The problem identified in this article was known for many years when it was published, certainly since the famous 1968 NATO conference in Garmisch-Partenkirschen. What is significant about this article is the fact that it appeared in the Scientific American, a publication that reaches an audience much wider than computer scientists and software engineers. The reason it deserved being published in

such a journal was that the general public had just been hit by one of the most famous software-related failures: the luggage delivery system that kept the brand new Denver airport shut for months at a huge expense. In other words, it is not that the problem had suddenly started to give headaches to software developers but that it became clear that it was hurting the economy, i.e. reaching into people's pockets.

Indeed, as the scope and role of software in business grew, so did the size of programs: software applications were (and still are) demanded to perform more and more tasks in the business domain and, as a consequence, they grew very quickly into millions of lines of code. Sheer size compromised quality: delivery times started to suffer and so did performance and correctness due to the fact that applications became unmanageable for the lone programmer. The analogy with building skyscrapers using carpenters is a very powerful one. Engineering principles were quickly identified to be required to face the complexity of the product and the term programming "in-the-large" was coined to reflect the fact that software development needed another activity to be supported: one that could break the task into manageable pieces [5].

> *We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.[…]*
>
> *We argue that structuring a large collection of modules to form a "system" is an essentially distinct and different intellectual activity from that of constructing the individual modules. That is, we distinguish programming-in-the-large from programming-in-the-small.*

This is where a second important landmark in the history of Software Engineering is normally placed. Please note that these are just milestones: the columns in Fig. 1 should not be taken as disjoint periods in this history. In fact, one should ignore the reference to the decades (60s, 70s, 80s and 90s) as they are neither accurate nor identifiers of periods in the history of Software Engineering. Indeed, there is still a role for programming-in-the-small, as recognised above, and for programming-in-the-head, e.g. for software embedded in some critical systems.

3.2 Module Interconnection Languages

It should be clear that programming in-the-large addresses an altogether different notion of complexity, one that occurs at "design" or "compile" time. We are not so much concerned with the flow of execution of a computation but with "workflow" in a development process.

A different kind of decomposition is, therefore, at stake: one that addresses the global structure of a software application in terms of what its modules and resources are and how they fit together in the system. The resulting components (modules) are interconnected not to ensure that the computation progresses towards the required output, but that, in the final system, all modules are provided with the resources they need (e.g. the parsing module of a compiler is connected to the symbol table). In other words, it is the flow of resources among modules that is of concern. Therefore, one tends to use primitives such as *export/provide/originate* and *import/require/use* when designing individual modules.

The conclusions of Parnas' landmark paper [20] are even clearer in distinguishing program complexity/architecture from the complexity that is associated with programming "in-the-large":

> *We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.*

That is to say, we cannot hope and should not attempt to address the complexity of software systems as products with the mechanisms that were developed for structuring complex computations. That is why so-called *module interconnection languages* (MILs) were developed for programming in-the-large [22].

In the architectures that are described in such languages, dependencies between components concern access to and usage of resources, not control flow. Whereas program architectures make it much simpler to understand and prove the correctness of the code with respect to input/output specifications, module interconnection architectures are essential for project management, namely for testing and maintenance support: they enforce system integrity and inter-modular compatibility; they support incremental modification as modules can be independently compiled and linked, and thus full recompilation of a modified system is not needed; and they enforce version control as different versions (implementations) of a module can be identified and used in the construction of a system.

We should also make clear that problems of "size" do not just arise during software design. Even if specification languages can factor down the size of code by at least one order of magnitude, they do not factor out complexity altogether. For instance, in algebraic specifications, which consist of sets of sentences (axioms) in a given logic [17], the need to structure these sets in manageable pieces was recognised as early as 1977 in a famous article by Burstall and Goguen whose title is, precisely, "putting theories together to make specifications" [4]. This concern for the complexity of specifications signalled the advent of category theory [8] as a mathematical toolbox offering techniques for structuring logical theories into what became known as the "theory of institutions" [14]. Modularisation principles were extensively explored in this setting that contributed to the maturation of software development as an engineering discipline [13,25].

Other problems of "size" are also reflected in Fig. 1, e.g. in the data that some software applications are required to manipulate, which led to the development of database technologies. The same applies to control structures in the sense that termination ceased to be a correctness factor and properties of on-going execution like responsiveness started to emerge in applications such as operating and air traffic control systems.

3.3 The Case for Object-Oriented and Component-Based Development

The article in the Scientific American goes one step further and offers possible ways out of the crisis:

> *Musket makers did not get more productive until Eli Whitney figured out how to manufacture interchangeable parts that could be assembled by any skilled workman. In like manner, software parts can, if properly standardized, be reused at many different scales. […]*
>
> *In April [1994], NIST announced that it was creating an Advanced Technology Program to help engender a market for component-based software.*

Indeed, the publication of this article is also marked by the advent of object-oriented (OO) and component-based development. In the context of the "small"/"large" divide, OO makes important contributions:

- State encapsulation provides a criterion for modularising code: software is organised in classes that group together in methods all the operations that are allowed on a given piece of the system state.
- Programming-in-the-small is used within a class to define its methods.
- Clientship is used for interconnecting objects: an object can be declared to be a client of another object, and methods of the client can invoke the execution of methods of the server as part of their code.
- Inheritance is used for classifying and organising classes in hierarchies that facilitate reuse.

Object-orientation cannot be taken primarily as a means of "programming-in-the-large". In fact, one can argue that, in spite of grouping functionalities in classes, OO development could do with additional mechanisms for managing huge collections of classes… Organising classes in inheritance hierarchies is a step in that direction but many would argue that it does not take software development deep enough into an engineering practice.

Still, one has to recognise that OO software development has brought a significant improvement to the management of the complexity of software development. When one considers alternative mechanisms for modularising imperative programming such as those introduced over Pascal to produce Modula, it is clear that OO is much richer in "methodological" contents in the sense that classes as software modules and interconnection via clientship, even if providing only for a rather fine grain of decomposition, organise systems according to structures that can be recognised in the problem domain.

4 The Crisis 10 Years Later

In spite of the recognised progress towards the management of the complexity of constructing large applications, an article published in May 2003 alerted to the fact that software was still under a "crisis":

> *Computing has certainly got faster, smarter and cheaper, but it has also become much more complex. Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. […]*
>
> *In the late 1990s, the internet and the emergence of e-commerce "broke IT's back". Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. […]*
>
> *Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. […]*
>
> *For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. […]*
>
> *Applications will no longer be a big chunk of software that runs on a computer but a combination of web services.*

4.1 Programming in-the-world

One has to recognise that a different notion of complexity is involved here. The article is very explicit in saying that the problem now is not one of size – "large chunks of software" – but that the complexity lies in the fact that systems are ever more distributed and heterogeneous, and that software development requires the integration and combination of possibly "incompatible" systems. Societal and economical implications of this notion of complexity are not any smaller. The fact that this article appeared not in the Scientific American but in a wider circulation publication – The Economist – shows that the debate now concerns a much more general public.

In our opinion, one can realise that this crisis is of a different nature in the fact that the discussion is no longer around the complexity of building a large application that one needs to deliver, in time and budget, to a client, but of managing an open-ended structure of autonomous components, possibly distributed and highly heterogeneous. This means developing software components that are autonomous and can be interconnected with other components, software or otherwise, and managing the interconnections themselves as new components may be required to join in and other to be removed.

In software engineering, the term *software architecture* [3,10,21,24] has recently been reserved to this different kind of complexity. Components are treated as independent entities that may interact with each other along well-defined lines of communication called architectural connectors [19]. By focusing on particular kinds of components and connectors, one can identify different architectural styles. One can even compare the architectures induced by different styles on the same system and discuss system properties without having to analyse the code.

In a sense, we are going back to the kind of architecture provided by structured programming but at a higher level of abstraction, one which often involves abstractions not directly provided by the underlying programming language: pipes, filters, event broadcast, client-server protocols, etc. In other words, it is not so much the flow of control that we want to structure but the flow of interactions.

It is interesting to note that, from a mathematical point of view, category theory [8], as the mathematics of "structure", still plays a fundamental role in formalising these architectural principles and techniques [9]. However, instead of structuring large specifications, as discussed in Sect 3.2, we are now interested in run-time configurations of complex systems [12] and in the properties that emerge from the interactions within them [7].

Notice that, in this context, object-orientation can be clearly identified with an architectural style among many others. In this respect, the article in The Economist challenges us to identify an architectural style that can address the complexity of the new generation of systems that is emerging from the internet, mobile communication, and other such "global computers"[2] in what one could label programming "in-the-world" [24]. In this context, one can indeed debate the merits of an object-oriented style and discuss the role and status of services.

4.2 The Case for Services

Object-oriented techniques offer little support for the kind of decomposition, organisation and architectural style required for programming in-the-world. Interactions in OO are based on *identities* [16], in the sense that, through clientship, objects interact by invoking the methods of specific objects (instances) to get something specific done: to use another object's services, an object needs to have the server's identity to send it a message or call the required service. This implies that any unanticipated change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established [23].

This in why some people claim that OO brought GOTOs back into fashion. One cannot but recognise that, indeed, clientship through feature calling and method invocation works for interactions in the same way as GOTOs worked for control flow. Indeed, one often has the feeling that HyperText and URLs in web-based IT applications tend to become the web designers' version of spaghetti code.

Hence, in our opinion, the challenges raised in The Economist show that a different paradigm is required to address what is clearly a different form of software complexity. This is, precisely, the paradigm that started to emerge in the guise of *web services* [1].

Web services have been often characterised as "self-contained, modular applications that can be described, published, located, and invoked over a network, generally the web" [27]. Building applications under this new paradigm is a dynamic process that consists in locating services that provide the basic functionalities that are required, and "orchestrating" them, i.e. establishing collaborations between them, at run-time, so that the desired global properties of the application can emerge from their joint behaviour, just in time.

---

[2] "A global computer is a programmable computational infrastructure distributed at worldwide scale and available globally. It provides uniform services with variable guarantees for communication, cooperation and mobility, modalities and disciplines for resource usage, security policies and mechanisms, and more." [26].

"Integration" is another keyword in this process, often found married to "orchestration" or "marshalling": application building in service-oriented architectures is based on the composition of services that have to be discovered and "marshalled" dynamically at run-time. Therefore, one of the characteristics of the service-oriented paradigm is, precisely, the ability that it requires for interconnections to be established and revised dynamically, in run-time, without having to suspend execution, i.e. without interruption of "service". This is what is usually called "late" or "just-in-time" integration (as opposed to compile or design time integration).

What marks the difference between this aspect of the complexity of software from the one addressed by programming-in-the-large is, precisely, the fact that software is not being treated as a *product* but as a *service*, as the article of The Economist makes clear. It is interesting to note that this shift from object/product to service-oriented interactions mirrors what has been happening already in the economy: more and more, business relationships are established in terms of acquisition of services (e.g. 1000 Watts of lighting for your office) instead of products (10 lamps of 100 Watts each for the office). That is, software engineering is just following the path being set for the economy in general and, thus, shifting somewhat away from the more traditional "industrial" technologies oriented to the production of goods to exhibit the problems that are characteristic of more "social" domains.

5 Concluding Remarks

We hope that the previous sections made clear that, in our opinion, the case for a new service-oriented paradigm rests, essentially, in the recognition that there is more than one dimension of complexity in the engineering of software intensive systems.

Indeed, even in our everyday life, we use the term "complex" in a variety of ways. Many times, we apply it to entities or situations that are "complicated" in the sense that they offer great difficulty in understanding, solving, or explaining. There is nothing necessarily wrong or faulty in them; they are just the unavoidable result of a necessary combination of parts or factors. For instance, the human body is a complex entity; none of its organs operates autonomously if separated from the whole but we know which vital functions each provides as part of the body, and can understand how the functioning of each of them depends on the rest to the extent that we can replace them by others of the same type.

In other circumstances, complexity derives more from the number and "open" nature of interactions that involve "autonomic" parts. Social systems are inherently complex in the sense that it is very difficult to predict what properties can emerge from the behaviours of the parts and their interactions. Regulations and regulators can be superposed to influence the way the parts interact or induce properties that one would like to see emerge, but complete control is hard to achieve.

Knowledge of the physiological structure of a part does not necessarily help in understanding how it can contribute to the functioning of a social system. For instance, the social behaviour of human beings is essentially independent of their physiology. One does not go to a psychiatrist because of a toothache (even if the toothache is driving us mad), or to the dentist complaining with stress. A car will usually have a driver's manual explaining how it can be used for social purposes (i.e. driving) and a

technical manual that a mechanic can use for fixing a faulty part: one does not consult the technical manual to find out where to switch the headlights and the mechanic does not need the driver's manual to replace a bulb. A mechanic does not need a driver's licence to repair a car, and knowing that one should stop at the red light is not something that derives from the structure of the car.

Having said this, we should point out something equally obvious: that they can very well be related. We all know that a speech impediment is likely to influence one's ability to socialise and that, without brakes, a car cannot be stopped at a red light.

In our opinion, this distinction applies to software as well. Programming in-the-large is concerned with the physiological complexity of software systems. Programming in-the-world with their social complexity. As such, the methods and techniques that best apply to one do not necessarily serve the other in the best possible way. We see component-based development as addressing in-the-large issues, as highlighted in the article of the Scientific American. We see services as addressing social complexity, as the article of The Economist clearly suggests.

Service-oriented architectures are still very much in their infancy, and still too much bound to the internet, in the guise of web services, or other specific global computers like the grid, in what are known as grid services. Service-based computing and software development is being uptaken by the IT industry in an ad-hoc and undisciplined way, raising the spectrum of a society and economy dependent on applications that have the ability to "talk" to each other but without "understanding" what they are talking about.

This scenario suggests very clearly that a scientific challenge is there to provide the mathematical, methodological and technological foundations that are required for the new paradigm to be used effectively and responsibly in the development of the generation of systems that will operate the Information Society of tomorrow. Such is the challenge that a consortium of European universities, research institutes and companies chose to address under the IST-FET-GC2 integrated project SENSORIA (Software Engineering for Service-Oriented Overlay Computers). SENSORIA is addressing the social complexity involved in service-oriented development precisely through some of the technologies that characterise programming "in-the-world": coordination languages and models [2,11], distributed reconfigurable systems [18], and software architectures [10,21,24].

References

1. G. Alonso, F. Casati, H. Kuno, V. Machiraju (2004) *Web Services*. Springer, Berlin Heidelberg New York
2. F. Arbab (1998) What do you mean, coordination? In: *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, March 1998
3. L. Bass, P. Clements, R. Kasman (1998) *Software Architecture in Practice*. Addison-Wesley, Reading MA
4. R. Burstall, J. Goguen (1977) Putting theories together to make specifications. In: R. Reddy (ed) *Proc. Fifth International Joint Conference on Artificial Intelligence*, August 1977, Cambridge, MA, pp 1045–1058
5. F. DeRemer, H. Kron (1976) Programming-in-the-Large versus programming-in-the-small. *IEEE Transactions on Software Engineering* SE-2(2): 321–327

6. E. Dijkstra (1976) *A Discipline of Programming*. Prentice Hall, London
7. J. L. Fiadeiro (1996) On the emergence of properties in component-based systems. In: M. Wirsing, M. Nivat (eds) *Algebraic Methodology and Software Technology. LNCS, vol 1101*. Springer, Berlin Heidelberg New York, pp 421–443
8. J. L. Fiadeiro (2004) *Categories for Software Engineering*. Springer, Berlin Heidelberg New York
9. J. L. Fiadeiro, A. Lopes (1997) Semantics of architectural connectors. In: M. Bidoit, M. Dauchet (eds) *Theory and Practice of Software Development*. *LNCS, vol 1214*. Springer, Berlin Heidelberg New York, pp 505–519
10. D. Garlan, D. Perry (1994) Software architecture: practice, potential, and pitfalls. In: *Proc. 16th International Conference on Software Engineering*. IEEE Computer Society Press, Silver Spring MD, pp 363–364
11. D. Gelernter, N. Carriero (1992) Coordination languages and their significance. *Communications ACM* 35(2):97–107
12. J. Goguen (1973) Categorical foundations for general systems theory. In: F. Pichler, R. Trappl (eds) *Advances in Cybernetics and Systems Research*. Transcripta Books, New York, pp 121–130
13. J. Goguen (1986) Reusing and interconnecting software components. *IEEE Computer* 19(2):16–28
14. J. Goguen, R. Burstall (1992) Institutions: abstract model theory for specification and programming. *Journal ACM* 39(1):95–146
15. M. Jackson (1975) *Principles of Program Design*. Academic Press, New York
16. W. Kent (1993) Participants and performers: a basis for classifying object models. In: *Proc. OOPSLA 1993 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*
17. J. Loeckx, H.-D. Ehrich, M. Wolf (1996) *Specification of Abstract Data Types*. Wiley, New York
18. J. Magee, J. Kramer, M. Sloman (1989) Constructing distributed systems in Conic. *IEEE TOSE* 15(6):663–675
19. N. Mehta, N. Medvidovic, S. Phadke (2000) Towards a taxonomy of software connectors. In: *Proc. 22nd International Conference on Software Engineering*. IEEE Computer Society Press, Silver Spring MD, pp 178–187
20. D. Parnas (1972) On the criteria for decomposing systems into modules. In: *Communications of the ACM* 15(12):1053–1058
21. D. Perry, A. Wolf (1992) Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes* 17(4):40–52
22. R. Prieto-Diaz, J. Neighbors (1986) Module interconnection languages. *Journal of Systems and Software* 6(4):307-334
23. M. Shaw (1996) Procedure calls are the assembly language of software interconnection: connectors deserve first-class status. In: D. A. Lamb (ed) *Studies of Software Design*. *LNCS, vol 1078*. Springer, Berlin Heidelberg New York, pp 17–32
24. M. Shaw (1996) Three patterns that help explain the development of software engineering (position paper), *Dagstuhl Workshop on Software Architecture*
25. Y. Srinivas, R. Jüllig (1995) Specware™: formal support for composing software. In: B. Möller (ed) *Mathematics of Program Construction*. *LNCS, vol 947*. Springer, Berlin Heidelberg New York, pp 399–422
26. FET-GC2 Workprogramme text. www.cordis.lu/ist/fet/gc.htm#what
27. www.ibm.com/developerworks/web/library

Integrating Variants of DC*

He Jifeng[1,**] and Jin Naiyong[2]

[1] International Institute for Software Technology, United Nations University, Macau
hjf@iist.unu.edu
[2] SEI of East China Normal University, Shanghai, China

1 Introduction

Duration Calculus (DC) [22] was introduced as a logic to specify real-time requirements of computing systems. It has been used successfully in a number of case studies. Moreover, many variants of DC were proposed to deal with various real-time systems, including communicating processes [24], sequential hybrid systems [19, 23]. imperative programming languages [2, 17, 18, 24], finite divergence [6] and liveness properties [1, 25]. This paper aims to integrate those variants, and provides a logical framework for DC-based programming, and a design calculus for mixed hardware/software systems.

The main contribution of this paper includes some novel features

(1) Weak chop inverse constructs $(l = x)\backslash A$ and $A/(l = x)$, which can be used to specify liveness properties of real-time programs, and to define the neighbourhood operators $\diamond_l$ and $\diamond_r$ introduced in [1].

(2) Higher order quantifier $\exists V$ to describe the behaviour of local program variables of real-time programs.

(3) A formal definition of substitution of state variables $A[W/V]$ which enables us to define super-dense computations.

(4) A super-dense chop operator $\circ$, defined by the hiding operator and substitution, is used to model sequential composition of imperative programming languages [24].

The language is a *conservative* extension of DC in the sense that it adopts the same semantic definition for all the elements of DC, and preserves all the laws of variants of DC (including Neighbourhood Logic [1], Duration Calculus with iteration [2], Higher-order Duration Calculus [26], DC with super-dense chop [24], Recursive Duration Calculus [17]).

Like most of interval logical languages, our language contains

(1) a set of global variables x, $y, \ldots, z$.

(2) a set of state variables V, W, $\ldots, Z$.

* The work is partially supported by the 211 Key Project of the MoE, and the 973 project (no. 2002CB312001) of the MoST of China.

** On leave from the SEI of East China Normal University, Shanghai.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 14–34, 2005.
© Springer-Verlag Berlin Heidelberg 2005

(3) a set of temporal variables. In addition to the length temporal variable l, we associate with every state variable V the following temporal variables

- point values $\mathbf{b}V$ and $\mathbf{e}V$
- neighbourhood values $\overleftarrow{V}$ and $\overrightarrow{V}$.

(4) a set of propositional temporal letters.

(5) a set of relation symbols.

(6) a set of function symbols.

Global variables and temporal variables are terms. If f is an n-place function symbol, and $t_1, \ldots t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term.

The syntax of formulae is defined as follows

$$F ::= AF \mid \neg F \mid F \wedge F \mid F \frown F \mid (l = x)\backslash F \mid F/(l = x) \mid$$
$$\exists x \bullet F \mid \exists V \bullet F \mid F[W/V] \mid \bigvee \mathcal{F}$$

where the atomic formulae AF are of kinds P and $R(t_1, \ldots, t_n)$ where P is a propositional temporal letter, and R is n-place relation symbol, and $\mathcal{F}$ stands for a set of formulae.

The remaining of the paper is organised as follows. Section 2 is devoted to the primitive features of the logical language, including logical connectives, chop operator, temporal variable l and quantifier over global variables. Section 3 introduces the weak chop inverse operators and discusses its properties. In section 4 we illustrate how to formalise the neighbourhood operators [1] in our language. Section 5 deals with state variables and the induced temporal variables point values and neighbourhood values. We tackle higher order quantifiers and substitution operators in Section 6. Section 7 introduces the super-dense chop operator. Section 8 presents the infinite disjunction operator, which enables us to model iterative constructs of real-time programming languages , and handle the recursive DC formulae [17]. Section 9 shows how to treat DC^* [2] as one of sub-languages. The paper ends with a short concluding section.

2 The Primitive Features

2.1 Time Domain

We adopt continuous time represented by reals

$$\mathbf{T} =_{df} Real$$

The set $\mathbf{I}$ of intervals is defined by

$$\mathbf{I} =_{df} \{[b, e] \mid b, e \in \mathbf{T} \wedge b \leq e\}$$

We use the temporal variable $l : \mathbf{I} \rightarrow Real$ to represent the length of interval

$$\mathcal{M}_{[b,e]}(l) =_{df} (e - b)$$

2.2 Connectives

Let A and B be formulae. Let $\mathcal{M}$ be a model, define

$$\mathcal{M}_{[b,\,e]}(A \wedge B) \;=_{df}\; \mathcal{M}_{[b,\,e]}(A)\,\mathbf{and}\,\mathcal{M}_{[b,\,e]}(B)$$
$$\mathcal{M}_{[b,\,e]}(\neg A) \;=_{df}\; \mathbf{not}\,\mathcal{M}_{[b,\,e]}(A)$$

where $\{\mathbf{and},\ \mathbf{not}\}$ are the connectives of the propositional logic. As usual, we define $A \vee B \;=_{df}\; \neg(\neg A \wedge \neg B)$.

2.3 Chop

Definition 2.1

Let A and B be formulae. Define their composition $A \frown B$ by

$$\mathcal{M}_{[b,\,e]}(A \frown B) \;=_{df}\; \exists m : [b,\,e] \bullet (\mathcal{M}_{[b,\,m]}(A)\,\mathbf{and}\,\mathcal{M}_{[m,\,e]}(B)) \qquad \square$$

The chop operator enjoys the following properties.

($\frown$-1) (associativity) $\quad (A \frown B) \frown C \;\Leftrightarrow\; A \frown (B \frown C)$

($\frown$-2) (unit) $\quad (l = 0) \frown A \;\Leftrightarrow\; A \;\Leftrightarrow\; A \frown (l = 0)$

($\frown$-3) (zero) $\quad \mathbf{false} \frown A \;\Leftrightarrow\; \mathbf{false} \;\Leftrightarrow A \frown \mathbf{false}$

($\frown$-4) (distributivity) $\quad (A1 \vee A2) \frown B \;\Leftrightarrow\; (A1 \frown B) \vee (A2 \frown B)$

$$A \frown (B1 \vee B2) \;\Leftrightarrow\; (A \frown B1) \vee (A \frown B2)$$

($\frown$-5) (monotonicity) $\quad$ If $A1 \Rightarrow A2$, then

$$(A1 \frown B) \Rightarrow (A2 \frown B)$$
$$(B \frown A1) \Rightarrow (B \frown A2)$$

The validity of the above laws is based on the following facts:
(1) the chop operator is defined as the lift of the catenation operator of intervals.
(2) the catenation operator is associative, and has the point intervals as its unit.

2.4 Temporal Variable l

The temporal variable l is governed by the following laws
(l-1) $l \geq 0$,

(l-2) $x,\, y \geq 0 \Rightarrow (l = x) \frown (l = y) \;\Leftrightarrow\; (l = x + y)$

Definition 2.2

Let $c \geq 0$. Define

$$\mathcal{R}_c(A) \;=_{df}\; A \frown (l = c)$$
$$\mathcal{L}_c(A) \;=_{df}\; (l = c) \frown A \qquad \square$$

Fact 2.3 $\mathcal{M}_{[b-c,\,e]}(\mathcal{L}_c(A)) \;=\; \mathcal{M}_{[b,\,e]}(A) \;=\; \mathcal{M}_{[b,\,e+c]}(\mathcal{R}_c(A))$ $\qquad \square$

($\mathcal{L}_c\mathcal{R}_c$-1) (Injectiveness) $\quad \mathcal{L}_c(A) \Rightarrow \mathcal{L}_c(B)\ \ \mathbf{iff}\ A \Rightarrow B$

$$\mathcal{R}_c(A) \Rightarrow \mathcal{R}_c(B)\ \ \mathbf{iff}\ A \Rightarrow B$$

Proof $\mathbf{not}(A \Rightarrow B)$ $\{$Def of $\Rightarrow\}$

$\Longrightarrow$ $\exists \mathcal{M}, b,\ e \bullet (\mathcal{M}_{[b,\,e]}(A) = tt)$ **and** $(\mathcal{M}_{[b,\,e]}(B) = ff)$ $\{$Fact 2.3$\}$

$\Longrightarrow$ $\exists \mathcal{M}, b,\ e \bullet (\mathcal{M}_{[b,\,e+c]}(\mathcal{R}_c(A)) = tt)$ **and** $(\mathcal{M}_{[b,\,e+c]}(\mathcal{R}_c(B)) = ff)$ $\{$Def of $\Rightarrow\}$

$\Longrightarrow$ $\mathbf{not}(\mathcal{R}_c(A) \Rightarrow \mathcal{R}_c(B))$ □

($\mathcal{L}_c\mathcal{R}_c$-2) (Conjunctivity) $\mathcal{R}_c(A \wedge B) \Leftrightarrow \mathcal{R}_c(A) \wedge \mathcal{R}_c(B)$

$\mathcal{L}_c(A \wedge B) \Leftrightarrow \mathcal{L}_c(A) \wedge \mathcal{L}_c(B)$

Theorem 2.4

$\mathcal{R}_c(A) \Leftrightarrow$ **false iff** $A \Leftrightarrow$ **false iff** $\mathcal{L}_c(A) \Leftrightarrow$ **false**

Proof From ($\frown$-3) $\mathcal{L}_c(\mathbf{false}) \Leftrightarrow \mathbf{false} \Leftrightarrow \mathcal{R}_c(\mathbf{false})$. The conclusion follows from ($\mathcal{L}_c\mathcal{R}_c$-1). □

Theorem 2.5

(1) $\mathcal{R}_c(\neg A) \Leftrightarrow (l \geq c) \wedge \neg \mathcal{R}_c(A)$

(2) $\mathcal{L}_c(\neg A) \Leftrightarrow (l \geq c) \wedge \neg \mathcal{L}_c(A)$

Proof $\mathcal{R}_c(\neg A)$ $\{(\mathcal{L}_c\mathcal{R}_c - 2)$ and Theorem 2.4$\}$

$\Leftrightarrow \mathcal{R}_c(\neg A) \wedge \neg \mathcal{R}_c(A)$ $\{$Predicate Logic$\}$

$\Leftrightarrow (\mathcal{R}_c(\neg A) \vee \mathcal{R}_c(A)) \wedge \neg \mathcal{R}_c(A)$ $\{(\frown -4)\}$

$\Leftrightarrow (l \geq c) \wedge \neg \mathcal{R}_c(A)$ □

2.5 Quantifier

Definition 2.6

Let x be a global variable. Define

$$\mathcal{M}_{[b,\,e]}(\exists x \bullet A) = tt \quad \textbf{if} \quad \exists M' \bullet M'_{[b,\,e]}(A) = tt \textbf{ and } M \equiv_x M'$$

where $M \equiv_x M' \ =_{df} \ \forall y \neq x \bullet M(y) = M'(y)$ □

($\exists$-1) (Extension of the scope) If x is not free in A then

$$\exists x \bullet (A \frown B) \Leftrightarrow A \frown (\exists x \bullet B)$$

$$\exists x \bullet (B \frown A) \Leftrightarrow (\exists x \bullet B) \frown A$$

3 Inverse of Chop

Definition 3.1 (Weak Chop Inverse)

Let $c \geq 0$. Define

$$\mathcal{M}_{[b,\,e]}(A/(l = c)) =_{df} \mathcal{M}_{[b,\,e+c]}(A)$$

$$\mathcal{M}_{[b,\,e]}((l = c)\backslash A) =_{df} \mathcal{M}_{[b-c,\,e]}(A)$$ □

The following law shows that $/(l = c)$ ($\backslash(l = c)$ resp.) is the inverse of $\mathcal{R}_c$ ($\mathcal{L}_c$ resp).

(WCI-1) $c \geq 0 \;\Rightarrow\; \mathcal{R}_c(A/(l = c)) \;\Leftrightarrow\; A \wedge (l \geq c)$

$c \geq 0 \;\Rightarrow\; \mathcal{L}_c((l = c)\backslash A) \;\Leftrightarrow\; A \wedge (l \geq c)$

Proof $\mathcal{M}_{[b,\,e]}(\mathcal{R}_c(A/(l = c))) = tt$ {Def 2.1}

iff $e \geq b + c \wedge \mathcal{M}_{[b,\,e-c]}(A/(l = c)) = tt$ {Def 3.1}

iff $e \geq b + c \wedge \mathcal{M}_{[b,\,e]}(A) = tt$ {Def of $A \wedge B$}

iff $\mathcal{M}_{[b,\,e]}(A \wedge (l \geq c)) = tt$ □

Theorem 3.2

(1) $\mathcal{R}_c(X) \Rightarrow A$ **iff** $X \Rightarrow A/(l = c)$

(2) $\mathcal{R}_c(A) \wedge B \;\Leftrightarrow$ **false** **iff** $A \wedge (B/(l = c)) \;\Leftrightarrow$ **false**

(3) $(l = y)\backslash\mathcal{R}_x(A \wedge (l \geq y)) \;\Leftrightarrow\; \mathcal{R}_x((l = y)\backslash(A \wedge (l \geq y))$

(4) $(A/(l = x))/(l = y) \;\Leftrightarrow\; A/(l = x + y)$

Proof of (1) $\mathcal{R}_c(X) \Rightarrow A$ {$(\frown -5)$}

$\Rightarrow$ $\mathcal{R}_c(X) \Rightarrow (A \wedge (l \geq c))$ {(WCI-1) and $(\mathcal{L}_c\mathcal{R}_c - 1)$}

$\Rightarrow$ $X \Rightarrow A/(l = c)$ {$(\frown -5)$ and (WCI-1)}

$\Rightarrow$ $\mathcal{R}_c(X) \Rightarrow A$

(2) $(A \wedge (B/(l = c)) \;\Leftrightarrow$ **false** {Theorems 2.4}

$\Leftrightarrow$ $\mathcal{R}_c(A) \wedge \mathcal{R}_c(B/(l = c)) \;\Leftrightarrow$ **false** {(WCI-1)}

$\Leftrightarrow$ $\mathcal{R}_c(A) \wedge B \wedge (l \geq c) \;\Leftrightarrow$ **false** {$(\frown -5)$}

$\Leftrightarrow$ $\mathcal{R}_c(A) \wedge B \;\Leftrightarrow$ **false**

(4) $X \Rightarrow (A/(l = x))/(l = y)$ {Conclusion (1)}

$\Leftrightarrow$ $((X \frown (l = y)) \frown (l = x)) \Rightarrow A$ {$(\frown -1)$, $(l - 2)$}

$\Leftrightarrow$ $(X \frown (l = x + y)) \Rightarrow A$ {Conclusion (1)}

$\Leftrightarrow$ $X \Rightarrow (A/(l = x + y))$ □

Theorem 3.3

(1) $\mathcal{L}_c(X) \Rightarrow A$ **iff** $X \Rightarrow (l = c)\backslash A$

(2) $\mathcal{L}_c(A) \wedge B \;\Leftrightarrow$ **false** **iff** $A \wedge ((l = c)\backslash B) \;\Leftrightarrow$ **false**

(3) $\mathcal{L}_x((A \wedge (l \geq y)))/(l = y) \;\Leftrightarrow\; \mathcal{L}_x((A \wedge (l \geq y))/(l = y))$

(4) $(l = y)\backslash((l = x)\backslash A) \;\Leftrightarrow\; (l = x + y)\backslash A$ □

Theorem 3.4 (distributivity)

(1) $\mathbf{true}/(l = c) \Leftrightarrow \mathbf{true}$

(2) $(A \wedge B)/(l = c) \Leftrightarrow (A/(l = c)) \wedge (B/(l = c))$

(3) $\neg(A/(l = c)) \Leftrightarrow (\neg A)/(l = c)$

(4) $(\exists x \bullet A)/(l = c)) \Leftrightarrow \exists x \bullet (A/(l = c))$

(5) $(A \frown (B \wedge (l \geq d))/(l = d) \Leftrightarrow A \frown ((B \wedge (l \geq d))/(l = d))$, where $d \geq 0$

Proof of (3)

$$\begin{aligned}
& \mathcal{R}_c(\neg(A/(l = c)) && \{\text{Theorem 2.5(1)}\} \\
\Leftrightarrow\ & (l \geq c) \wedge \neg\mathcal{R}_c(A/(l = c)) && \{(WCI - 1)\} \\
\Leftrightarrow\ & (l \geq c) \wedge \neg(A \wedge (l \geq c)) && \{\text{Predicative logic}\} \\
\Leftrightarrow\ & (\neg A) \wedge (l \geq c) && \{(WCI - 1)\} \\
\Leftrightarrow\ & \mathcal{R}_c(\neg A/(l = c)) && \square
\end{aligned}$$

Corollary 3.5

(1) $\mathbf{false}/(l = c) \Leftrightarrow \mathbf{false}$

(2) $(A \vee B)/(l = c) \Leftrightarrow (A/(l = c)) \vee (B/(l = c))$

Proof From Theorem 3.4 (1)-(3). $\square$

Theorem 3.6

(1) $(l = c)\backslash\mathbf{true} \Leftrightarrow \mathbf{true}$

(2) $(l = c)\backslash(A \wedge B)) \Leftrightarrow ((l = c)\backslash A) \wedge ((l = c)\backslash B)$

(3) $\neg((l = c)\backslash A) \Leftrightarrow (l = c)\backslash(\neg A)$

(4) $(l = c)\backslash(\exists x \bullet A) \Leftrightarrow \exists x \bullet ((l = c)\backslash A)$

(5) $(l = c)\backslash((A \wedge (l \geq c)) \frown B) \Leftrightarrow ((l = c)\backslash(A \wedge (l \geq c))) \frown B$, where $c \geq 0$ $\square$

Corollary 3.7

(1) $(l = c)\backslash\mathbf{false} \Leftrightarrow \mathbf{false}$

(2) $(l = c)\backslash(A \vee B) \Leftrightarrow ((l = c)\backslash A) \vee ((l = c)\backslash B)$ $\square$

Theorem 3.8

$(l = x)\backslash(A/(l = y)) \Leftrightarrow ((l = x)\backslash A)/(l = y)$

Proof From ($\frown -1$), Theorems 3.2(1) and 3.3(1). $\square$

4 Neighbourhood Logic

This section illustrates how to define the neighbourhood operators [1] in our framework.

Definition 4.1

Define

$$\diamond_r A \ =_{df} \ \mathbf{true} \frown (\exists c \geq 0 \bullet ((A \wedge (l = c))/(l = c)))$$
$$\diamond_l A \ =_{df} \ (\exists c \geq 0 \bullet (l = c) \backslash (A \wedge (l = c))) \frown \mathbf{true}$$

Define $\Box_l \ =_{df} \ \neg \diamond_l \neg$ and $\Box_r \ =_{df} \ \neg \diamond_r \neg$ □

($\diamond$-1) If $x \geq 0$ then

$$\diamond_r (l = x)$$
$$\diamond_l (l = x)$$

Proof $\diamond_r (l = x)$ {Def 4.1}

$\Leftrightarrow \mathbf{true} \frown \exists c \geq 0 \bullet (((l = x) \wedge (l = c))/(l = c))$ {Corollary 3.5(1)}

$\Leftrightarrow \mathbf{true} \frown ((l = x)/(l = x))$ {$(\mathcal{L}_c \mathcal{R}_c - 1)$}

$\Leftrightarrow \mathbf{true} \frown (l = 0)$ {$(\frown -2)$}

$\Leftrightarrow \mathbf{true}$ □

($\diamond$-2) $\diamond_r (A \vee B) \ \Leftrightarrow \ \diamond_r A \vee \diamond_r (B)$

$\diamond_l (A \vee B) \ \Leftrightarrow \ \diamond_l A \vee \diamond_l (B)$

Proof From $(\frown -4)$ and Corollary 3.5(2). □

($\diamond$-3) If $A \Rightarrow B$, then $\diamond_r (A) \Rightarrow \diamond_r (B)$ and $\diamond_l (A) \Rightarrow \diamond_l (B)$

Proof From $(\frown -5)$. □

($\diamond$-4) $\diamond_r (\exists x \bullet A) \ \Leftrightarrow \ \exists x \bullet \diamond_r A$

$\diamond_l (\exists x \bullet A) \ \Leftrightarrow \ \exists x \bullet \diamond_l A$

Proof From Theorems 3.4(4) and 3.6(4). □

Lemma 4.2

(1) $A \wedge \diamond_r B \ \Leftrightarrow \ \exists x \geq 0 \bullet (A \frown (B \wedge (l = x))/(l = x)$

(2) $A \wedge \diamond_l B \ \Leftrightarrow \ \exists x \geq 0 \bullet (l = x) \backslash ((B \wedge (l = x)) \frown A)$

Proof $A \wedge \diamond_r B$ {Def 4.1 and $(\exists - 1)$}

$\Leftrightarrow \ A \wedge \exists x \geq 0 \bullet (\mathbf{true} \frown (B \wedge (l = x))/(l = x)$ {Predicative logic}

$\Leftrightarrow \ \exists x \geq 0 \bullet A \wedge ((\mathbf{true} \frown (B \wedge (l = x))/(l = x))$ {Theorem 3.4(5)}

$\Leftrightarrow \ \exists x \geq 0 \bullet ((A \frown (l = x))/(l = x)) \wedge$

$((\mathbf{true} \frown (B \wedge (l = x))/(l = x))$ {Theorem 3.4(2)}

$\Leftrightarrow \ \exists x \geq 0 \bullet (A \frown (B \wedge (l = x)))/(l = x)$ □

($\diamond$-5) $\quad \diamond_l \diamond_r A \;\Rightarrow\; \Box_l \diamond_r A$
$\diamond_r \diamond_l A \;\Rightarrow\; \Box_r \diamond_l A$

Proof $\diamond_l \diamond_r (A)$
{Def 4.1}
$\Leftrightarrow\; \exists x \geq 0 \bullet (l = x)\backslash(\diamond_r(A) \wedge (l = x)) \frown \mathbf{true}$
{Lemma 4.2 and Theorem 3.6(4)}
$\Leftrightarrow\; \exists x,\, y \geq 0 \bullet ((l = x)\backslash(((l = x) \frown (A \wedge (l = y)))/(l = y))) \frown \mathbf{true}$
{Theorems 3.6(5) and 3.8}
$\Leftrightarrow\; \exists x,\, y \geq 0 \bullet ((A \wedge (l = y))/(l = y)) \frown \mathbf{true}$
{Predicative logic}
$\Leftrightarrow\; \exists y \geq 0 \bullet ((A \wedge (l = y))/(l = y)) \frown \mathbf{true}$
{Theorem 3.4(2) and ($\frown$ $-$5)}
$\Rightarrow\; \exists y \geq 0 \bullet (\neg(((\neg A) \wedge (l = y))/(l = y))) \frown \mathbf{true}$

In a similar way we can show that

$$\neg\Box_l\diamond_r A \;\Leftrightarrow\; \forall y \geq 0 \bullet (((\neg A) \wedge (l = y))/(l = y)) \frown \mathbf{true}$$

from which it follows the conclusion. □

($\diamond$-6) $\quad (l = x) \Rightarrow (A \;\Leftrightarrow\; \diamond_l \diamond_r (A \wedge (l = x)))$
$(l = x) \Rightarrow (A \;\Leftrightarrow\; \diamond_r \diamond_l (A \wedge (l = x)))$

Proof $(l = x) \wedge \diamond_l \diamond_r (A \wedge (l = x))$ {Lemma 4.2}
$\Leftrightarrow\; \exists u \geq 0 \bullet (l = u)\backslash((\diamond_r(A \wedge (l = x)) \wedge (l = u)) \frown (l = x))$
{Lemma 4.2}
$\Leftrightarrow\; \exists u,\, v \geq 0 \bullet (l = u)\backslash$
$((((l = u) \frown (A \wedge (l = x) \wedge (l = v)))/(l = v)) \frown (l = x))$
{Theorem 3.5(1)}
$\Leftrightarrow\; \exists u \geq 0 \bullet (l = u)\backslash(((l = u) \frown (A \wedge (l = x)))/(l = x)) \frown (l = x)$
{(WCI-1), Thm 3.8}
$\Leftrightarrow\; A \wedge (l = x)$ □

Lemma 4.3

(1) $\diamond_r(A \wedge (l = x + y)) \;\Leftrightarrow\; \diamond_r ((A \wedge (l = x + y))/(l = y))$, where $x,\, y \geq 0$

(2) $\diamond_l(A \wedge (l = x + y)) \;\Leftrightarrow\; \diamond_l ((l = y)\backslash(A \wedge (l = x + y)))$

Proof $\diamond_r(A \wedge (l = x + y))$ {Def 4.1}

$\Leftrightarrow\ \exists z \geq 0 \bullet (\mathbf{true} \frown ((A \wedge (l = x + y)) \wedge (l = z)))/(l = z)$

{$(\frown -3)$, and Corollary 3.5(1)}

$\Leftrightarrow\ (\mathbf{true} \frown (A \wedge (l = x + y)))/(l = x + y)$

{Theorems 3.2(4) and 3.4(5)}

$\Leftrightarrow\ (\mathbf{true} \frown ((A \wedge (l = x + y))/(l = y)))/(l = x)$ {Theorem 3.4(2)}

$\Leftrightarrow\ (\mathbf{true} \frown (((A \wedge (l = x + y))/(l = y)) \wedge (l = x)))/(l = x)$

{$(\frown -3)$, Theorem 3.5(2)}

$\Leftrightarrow\ \exists z \geq 0 \bullet (\mathbf{true} \frown (((A \wedge (l = x + y))/(l = y)) \wedge (l = z)))/(l = z)$

{Def 4.1}

$\Leftrightarrow\ \diamond_r((A \wedge (l = x + y))/(l = y))$ □

Theorem 4.4 (Nested neighbourhood)

(1) $\diamond_r(A \wedge \diamond_r B)\ \equiv\ \diamond_r (A \frown B)$

(2) $\diamond_l(A \wedge \diamond_l B)\ \equiv\ \diamond_l (B \frown A)$

Proof $\diamond_r(A \wedge \diamond_r B)$ {Lemma 4.2}

$\Leftrightarrow\ \diamond_r(\exists x \geq 0 \bullet (A \frown (B \wedge (l = x))/(l = x))$ $\{(\diamond - 4)\}$

$\Leftrightarrow\ \exists x \geq 0 \bullet \diamond_r((A \frown (B \wedge (l = x)))/(l = x))$ {Lemma 4.3}

$\Leftrightarrow\ \exists x \geq 0 \bullet \diamond_r(A \frown (B \wedge l = x))$ $\{(\diamond - 4)\}$

$\Leftrightarrow\ \diamond_r(\exists x \geq 0 \bullet (A \frown (B \wedge (l = x))))$ $\{(\exists - 1)\}$

$\Leftrightarrow\ \diamond_r(A; (B \wedge (l \geq 0)))$ {Theorem 2.8}

$\Leftrightarrow\ \diamond_r(A \frown B)$ □

($\diamond$-7) If x, $y \geq 0$, then

(1) $\diamond_r((l = x) \wedge \diamond_r((l = y) \wedge \diamond_r A))\ \Leftrightarrow\ \diamond_r ((l = x + y) \wedge \diamond_r A)$

(2) $\diamond_l((l = x) \wedge \diamond_l((l = y) \wedge \diamond_l A))\ \Leftrightarrow\ \diamond_l ((l = x + y) \wedge \diamond_l A)$

Proof $\diamond_r((l = x) \wedge \diamond_r((l = y) \wedge \diamond_r A))$ {Theorem 4.4}

$\Leftrightarrow\ \diamond_r((l = x) \wedge \diamond_r((l = y) \frown A))$ {Theorem 4.4}

$\Leftrightarrow\ \diamond_r((l = x) \frown ((l = y) \frown A))$ $\{(\frown -1)$ and $(l - 2)\}$

$\Leftrightarrow\ \diamond_r((l = x + y) \frown A)$ {Theorem 4.4}

$\Leftrightarrow\ \diamond_r((l = x + y) \wedge \diamond_r A)$ □

Theorem 4.5 (Conjunctivity)

(1) $\diamond_r(A \wedge (l = x)) \wedge \diamond_r(B \wedge (l = x))\ \Leftrightarrow\ \diamond_r (A \wedge B \wedge (l = x))$

(2) $\diamond_l(A \wedge (l = x)) \wedge \diamond_l(B \wedge (l = x))\ \Leftrightarrow\ \diamond_l (A \wedge B \wedge (l = x))$

Proof $\diamond_r(A \wedge (l = x)) \wedge \diamond_r(B \wedge (l = x))$ {Theorem 3.5(1) and Def 4.1}
$\Leftrightarrow ((\mathbf{true} \frown (A \wedge (l = x)))/(l = x)) \wedge ((\mathbf{true} \frown (B \wedge (l = x)))/(l = x))$
{Theorems 3.4(2)}
$\Leftrightarrow \mathbf{true} \frown ((A \wedge B \wedge (l = x))/(l = x))$ {Theorems 3.5(1)}
$\Leftrightarrow \diamond_r(A \wedge B \wedge (l = x))$ □

($\diamond$-8) $\diamond_r((l = x) \wedge A) \Rightarrow \Box_r((l = x) \Rightarrow A)$
$\diamond_l((l = x) \wedge A) \Rightarrow \Box_l((l = x) \Rightarrow A)$

Proof $\diamond_r((l = x) \wedge A) \wedge \neg\Box_r((l = x) \Rightarrow A)$ {Def 4.1}
$\Leftrightarrow \diamond_r((l = x) \wedge A) \wedge \diamond_r((l = x) \wedge \neg A)$ {Theorem 4.5}
$\Leftrightarrow \diamond_r(\mathbf{false})$ {Corollary 3.5(1)}
$\Leftrightarrow \mathbf{false}$ □

Theorem 4.6
(1) $A \frown (B \wedge \diamond_r(C)) \Leftrightarrow (A \frown B) \wedge \diamond_r(C)$
(2) $(A \wedge \diamond_r(C)) \frown (B \wedge (l = 0)) \Leftrightarrow (A \frown (B \wedge (l = 0)) \wedge \diamond_r(C)$

Proof $A \frown (B \wedge \diamond_r C)$ {Lemma 4.2}
$\Leftrightarrow A \frown (\exists x \geq 0 \bullet (B \frown (C \wedge (l = x)))/(l = x)$ {$(\exists -1)$}
$\Leftrightarrow \exists x \geq 0 \bullet A \frown ((B \frown (C \wedge (l = x)))/(l = x))$ {Theorem 3.4(5)}
$\Leftrightarrow \exists x \geq 0 \bullet (A \frown B \frown (C \wedge (l = x)))/(l = x)$ {Lemma 4.2}
$\Leftrightarrow (A \frown B) \wedge \diamond_r C$ □

Theorem 4.7
(1) $(\diamond_l(C) \wedge A) \frown B \Leftrightarrow \diamond_l(C) \wedge (A \frown B)$
(2) $(A \wedge (l = 0)) \frown (\diamond_l(C) \wedge B) \Leftrightarrow \diamond_l(C) \wedge ((A \wedge (l = 0)) \frown B)$ □

5 Induced Temporal Variables

5.1 Point Value

Definition 5.1 (Point value)
Let V be a state variable. Define

$$\mathcal{M}_{[b,\,e]}(\mathbf{b}V) =_{df} \mathcal{M}(V)(b)$$
$$\mathcal{M}_{[b,\,e]}(\mathbf{e}V) =_{df} \mathcal{M}(V)(e)$$ □

(*point*-1) $(A \wedge p(\mathbf{e}V)) \frown B \Leftrightarrow A \frown (p(\mathbf{b}V) \wedge B)$

Theorem 5.2

(1) $(p(\mathbf{b}V) \wedge A) \frown B \;\Leftrightarrow\; p(\mathbf{b}V) \wedge (A \frown B)$

(2) $(p(\mathbf{b}V) \wedge A)/(l = c) \;\Leftrightarrow\; p(\mathbf{b}V) \wedge (A/(l = c))$

(3) $l = 0 \Rightarrow p(\mathbf{b}v) \Leftrightarrow p(\mathbf{e}v)$

Proof of (1)

$$
\begin{array}{lll}
 & (p(\mathbf{b}V) \wedge A) \frown B & \{(\frown -2) \text{ and } (point-1)\} \\
\Leftrightarrow & ((l = 0 \wedge p(\mathbf{e}V)) \frown A) \frown B & \{(\frown -1) \text{ and } (point-1)\} \\
\Leftrightarrow & (l = 0) \frown (p(\mathbf{b}V) \wedge (A \frown B)) & \{(\frown -2)\} \\
\Leftrightarrow & p(\mathbf{b}v) \wedge (A \frown B) & \\
(2) & \mathcal{R}_c(p(\mathbf{b}V) \wedge (A/(l = c))) & \{(1) \text{ and (WCI-1)}\} \\
\Leftrightarrow & p(\mathbf{b}V) \wedge A \wedge (l \geq c) & \{\text{(WCI-1)}\} \\
\Leftrightarrow & \mathcal{R}_c(p(\mathbf{b}V) \wedge A) &
\end{array}
$$

which together with ($\mathcal{L}_c\mathcal{R}_c$-1) implies the conclusion. □

Theorem 5.3

(1) $A \frown (B \wedge p(\mathbf{e}V)) \;\Leftrightarrow\; (A \frown B) \wedge p(\mathbf{e}V)$

(2) $(l = c) \backslash (A \wedge p(\mathbf{e}V)) \;\Leftrightarrow\; ((l = c) \backslash A) \wedge p(\mathbf{e}V)$

Proof Dual to Theorem 5.2. □

(*point*-2) Let $S1$ and $S2$ be Boolean state variables.

If $S1 \Rightarrow S2$, then $\mathbf{b}S1 \Rightarrow \mathbf{b}S2$ and $\mathbf{e}S1 \Rightarrow \mathbf{e}S2$.

Remark 5.4

The following laws related to the point value of Boolean state variables are part of the mathematical theory which underlies the logical framework discussed in this paper.

$$
\begin{aligned}
\mathbf{b}(S1 \vee S2) &= \mathbf{b}S1 \vee \mathbf{b}S2 \\
\mathbf{b}(S1 \wedge S2) &= \mathbf{b}S1 \wedge \mathbf{b}S2 \\
\mathbf{b}(\neg S) &= \neg \mathbf{b}S
\end{aligned}
$$

Definition 5.5

We lift a Boolean state variable S to a formula by defining

$$\lfloor S \rfloor \;=_{df}\; \neg((l > 0) \frown \mathbf{b}(\neg S) \frown (l > 0))$$ □

Theorem 5.6

(1) $\lfloor true \rfloor \;\Leftrightarrow\; \mathbf{true}$

(2) $\lfloor false \rfloor \;\Leftrightarrow\; (l = 0)$

(3) $\lfloor S1 \wedge S2 \rfloor \;\Leftrightarrow\; \lfloor S1 \rfloor \wedge \lfloor S2 \rfloor$

(4) $\lfloor S \rfloor \;\Leftrightarrow\; \Box(\lfloor S \rfloor)$

Proof of (4) $\Box(\lfloor S \rfloor)$ {Def of $\Box$}

$\Leftrightarrow$ $\neg(\mathbf{true} \frown (\neg\lfloor S \rfloor) \frown \mathbf{true})$ {Def 5.5 and $(\frown -1)$}

$\Leftrightarrow$ $\neg((l > 0) \frown \mathbf{b}\neg S \frown (l > 0))$ {Def 5.5}

$\Leftrightarrow$ $\lfloor S \rfloor$ □

5.2 Neighbourhood Value

Definition 5.7 (Finite variability)

A state variable V has finite variability if its value can only change finite number of times in any interval, i.e., the formula

$$\Box(\exists c \bullet \diamond_l(\lfloor V = c \rfloor) \wedge \exists d \bullet \diamond_r(\lfloor V = d \rfloor))$$

holds. □

Definition 5.8 (Neighbourhood values)

Let V be a state variable with finite variability. Define

$$\mathcal{M}_{[b,\,e]}(\overrightarrow{V}) =_{df} c \quad \textbf{if} \quad \exists \delta > 0 \bullet \mathcal{M}_{[e,\,e+\delta]}(\lfloor V = c \rfloor) = tt$$

$$\mathcal{M}_{[b,\,e]}(\overleftarrow{V}) =_{df} c \quad \textbf{if} \quad \exists \delta > 0 \bullet \mathcal{M}_{[b-\delta,\,b]}(\lfloor V = c \rfloor) = tt$$ □

The neighbourhood values are captured by the following law:

(nbhood-1) $\overrightarrow{V} = c \;\Leftrightarrow\; \diamond_r(\lfloor V = c \rfloor)$

$\overleftarrow{V} = c \;\Leftrightarrow\; \diamond_l(\lfloor V = c \rfloor)$

Theorem 5.9 (Left neighbourhood value)

(1) $(p(\overleftarrow{V}) \wedge A) \frown B \;\Leftrightarrow\; p(\overleftarrow{V}) \wedge (A \frown B)$

(2) $(A \wedge (l = 0)) \frown (p(\overleftarrow{V}) \wedge B) \;\Leftrightarrow\; p(\overleftarrow{V}) \wedge ((A \wedge (l = 0)) \frown B)$

Proof of (1) $(p(\overleftarrow{V}) \wedge A) \frown B$ {Predicate Calculus}

$\Leftrightarrow$ $(\exists c \bullet p(c) \wedge \overleftarrow{V} = c \wedge A) \frown B$ $\{(\exists - 1)\}$

$\Leftrightarrow$ $\exists c \bullet (p(c) \wedge ((\overleftarrow{V} = c \wedge A) \frown B))$ $\{(nbhood - 1)\}$

$\Leftrightarrow$ $\exists c \bullet (p(c) \wedge ((\diamond_l(\lfloor V = c \rfloor) \wedge A) \frown B))$ {Theorem 4.7(1)}

$\Leftrightarrow$ $\exists c \bullet (p(c) \wedge \diamond_l(\lfloor V = c \rfloor)) \wedge (A \frown B)$ $\{(nbhood - 1)\}$

$\Leftrightarrow$ $p(\overleftarrow{V}) \wedge (A \frown B)$ □

Theorem 5.10 (Right neighbourhood value)

(1) $A \frown (B \wedge p(\overrightarrow{V})) \;\Leftrightarrow\; (A \frown B) \wedge p(\overrightarrow{V})$

(2) $(A \wedge p(\overrightarrow{V})) \frown (B \wedge (l = 0)) \;\Leftrightarrow\; (A \frown (B \wedge (l = 0)) \wedge p(\overrightarrow{V})$

Proof From Theorem 4.6 and (nbhood-1). □

Theorem 5.11 (Removal of neighbourhood value)

(1) $(A \wedge (\overrightarrow{V} = c)) \frown (B \wedge (l > 0)) \;\Leftrightarrow\; A \frown (B \wedge (\lfloor V = c \rfloor^+; \mathbf{true}))$

(2) $(A \wedge (l > 0)) \frown ((\overleftarrow{V} = c) \wedge B) \;\Leftrightarrow\; (A \wedge \lfloor V = c \rfloor^+; \mathbf{true})) \frown B$

where $\lfloor S \rfloor^+ \;=_{df}\; (l > 0) \wedge \lfloor S \rfloor$ □

6 Higher Order Quantifiers

6.1 Local State Variable

Definition 6.1

Let V be a state variable. Define

$$\mathcal{M}_{[b,\,e]}(\exists V \bullet F) \;=_{df}\; tt \quad \textbf{if } \exists \mathcal{M}' \bullet \mathcal{M}'_{[b,\,e]}(F) = tt \textbf{ and } \mathcal{M} \equiv_V \mathcal{M}'$$ □

(hiding-1) (Extension of the scope) If V is not free in A, then

$$\exists V \bullet (A \frown B) \;\Leftrightarrow\; A \frown (\exists V \bullet B)$$
$$\exists V \bullet (B \frown A) \;\Leftrightarrow\; (\exists V \bullet B) \frown A$$

Theorem 6.2

(1) $\exists V \bullet (A/(l = c)) \;\Leftrightarrow\; (\exists V \bullet A)/(l = c)$

(2) $\exists V \bullet ((l = c)\backslash A) \;\Leftrightarrow\; (l = c)\backslash(\exists V \bullet A)$

Proof of (1)

	$\mathcal{R}_c(LHS)$	{(hiding-1)}
$\Leftrightarrow$	$\exists V \bullet \mathcal{R}_c(A/(l = c))$	{(WCI-1)}
$\Leftrightarrow$	$\exists V \bullet (A \wedge (l \geq c))$	{Predicate Calculus}
$\Leftrightarrow$	$(\exists V \bullet A) \wedge (l \geq c)$	{(WCI-1)}
$\Leftrightarrow$	$\mathcal{R}_c(RHS)$	□

Corollary 6.3

(1) $\exists V \bullet \diamond_l(A) \;\Leftrightarrow\; \diamond_l (\exists V \bullet A)$

(2) $\exists V \bullet \diamond_r(A) \;\Leftrightarrow\; \diamond_r (\exists V \bullet A)$ □

From Lemma 4.2 and Theorem 4.6 and 4.7, we will confine ourselves in the following laws to those formulae which do not use the weak inverse operators $(l = c)\backslash$ and $/(l = c)$ and their derived operators $\diamond_l$ and $\diamond_r$ in the remaining of this paper.

(hiding-2) (Match of point values) If neither A nor B uses $\overleftarrow{V}$ and $\overrightarrow{V}$, then

$$\exists V \bullet (A \frown B) \;\Leftrightarrow\; \exists c \bullet (\exists V \bullet A \wedge \mathbf{e}V = c) \frown (\exists V \bullet B \wedge \mathbf{b}V = c)$$

(hiding-3) (Separation of neighbourhood values) If A does not refer to $\overleftarrow{V}$, and B does not mention $\overrightarrow{V}$, then

(1) $\exists V \bullet (p(\overleftarrow{V}) \wedge A) \;\Leftrightarrow\; (\exists V \bullet p(\overleftarrow{V})) \wedge (\exists V \bullet A)$

(2) $\exists V \bullet (p(\overrightarrow{V}) \wedge B) \;\Leftrightarrow\; (\exists V \bullet p(\overrightarrow{V})) \wedge (\exists V \bullet B)$

(hiding-4) (Separation of point values) If A does not use $\mathbf{b}V$, and B does not refer to $\mathbf{e}V$, then

(1) $\exists V \bullet (p(\mathbf{b}V) \wedge (A \wedge (l > 0)) \;\Leftrightarrow\; (\exists V \bullet p(\mathbf{b}V)) \wedge (\exists V \bullet A \wedge (l > 0))$

(2) $\exists V \bullet ((B \wedge (l > 0)) \wedge p(\mathbf{e}V)) \;\Leftrightarrow\; (\exists V \bullet (B \wedge (l > 0)) \wedge (\exists V \bullet p(\mathbf{e}V))$

Theorem 6.4

If neither A nor B uses $\mathbf{b}V$, $\mathbf{e}V$, $\overleftarrow{V}$, and $\overrightarrow{V}$, then

$$\exists V \bullet (A \frown B) \;\Leftrightarrow\; (\exists V \bullet A) \frown (\exists V \bullet B)$$

Proof $\exists V \bullet (A \frown B)$ $\{(hiding-2)\}$

$\Leftrightarrow\; \exists c \bullet (\exists V \bullet A \wedge \mathbf{e}V = c) \frown (\exists V \bullet \mathbf{b}V = c \wedge B)$

$\{(hiding-3)\,\&\,(hiding-4)\}$

$\Leftrightarrow\; \exists c(\exists V \bullet A \wedge \exists V \bullet \mathbf{e}V = c)) \frown (\exists V \bullet \mathbf{b}V = c) \wedge (\exists V \bullet B)$

{Predicate Calculus}

$\Leftrightarrow\; (\exists V \bullet A) \frown (\exists V \bullet B)$ □

The axioms (HD1) and (HD2) for higher-order quantifications in [26] follow directly from (hiding-3). We will establish the axiom (HD3) in the next section.

6.2 Substitution

Definition 6.5 (Substitution)

Let W be a state variable. The notation $A[W/V]$ stands for the result of substituting all free occurrences of V in A by W.

$$\mathcal{M}_{[b,\,e]}(A[W/V]) \;=_{df}\; tt \textbf{ if}$$
$$\exists \mathcal{M}' \bullet \mathcal{M}'_{[b,\,e]}(A) \;=\; tt \textbf{ and } \mathcal{M}' \equiv_V \mathcal{M} \textbf{ and } \mathcal{M}'(V) = \mathcal{M}(W) \qquad \square$$

(sub-1) $A[V/V] \;\Leftrightarrow\; A$

(sub-2) $A[W/V] \;\Leftrightarrow\; \exists V \bullet (A \wedge \mathbf{eq}(V, W))$

where W is distinct from V, and

$$\mathbf{eq}(U, W) \;=_{df}\; \overleftarrow{U}{=}\overleftarrow{W} \;\wedge\; \|U = W\| \;\wedge\; \overrightarrow{U}{=}\overrightarrow{W}$$
$$\|S\| \;=_{df}\; \mathbf{b}S \;\wedge\; \lfloor S \rfloor \;\wedge\; \mathbf{e}S$$

Theorem 6.6

(1) $\mathbf{eq}(U, W) \;\Rightarrow\; (A[U/V] \Leftrightarrow A[W/V])$

(2) $(\forall V \bullet A) \Rightarrow A[W/V]$

Proof Direct from (sub-1) and (sub-2). □

Corollary 6.7

Let W be a fresh state variable not used in A.

(1) $A \;\Leftrightarrow\; \exists W \bullet (A[W/V] \wedge \mathbf{eq}(W, V))$,

(2) If A does not use $\overrightarrow{V}$, then

$$A \;\Leftrightarrow\; \exists W \bullet (A[W/V] \wedge \overleftarrow{V}{=}\overleftarrow{W} \wedge \llbracket V = W \rrbracket)$$

(3) If A does not mention $\overleftarrow{V}$, then

$$A \;\Leftrightarrow\; \exists W \bullet (A[W/V] \wedge \overrightarrow{V}{=}\overrightarrow{W} \wedge \llbracket V = W \rrbracket)$$

(4) If neither $\overleftarrow{V}$ nor $\overrightarrow{V}$ occurs in A, then

$$A \;\Leftrightarrow\; \exists W \bullet (A[W/V] \wedge \llbracket V = W \rrbracket)$$

Proof From Theorem 6.6 and (hiding-3). □

Now we are going to show (HD3) in [26].

Theorem 6.8 If neither A nor B mentions point values $\mathbf{b}V$ and $\mathbf{e}V$, then

$$\left(\begin{array}{l}(\exists V \bullet A \wedge (\mathbf{true} \frown \lfloor V = x1 \rfloor^+) \wedge (\overrightarrow{V}{=} x2)) \frown \\ (\exists V \bullet B \wedge (\lfloor V = x2 \rfloor^+ \frown \mathbf{true}) \wedge (\overleftarrow{V}{=} x1))\end{array}\right) \Rightarrow\; \exists V \bullet (A \frown B)$$

Proof Define $A1 \;=_{df}\; \exists W \bullet A[W/V] \wedge \llbracket W = V \rrbracket \wedge (\overleftarrow{V}{=}\overleftarrow{V}) \wedge (\overrightarrow{W}{=} x2)$

$$B1 \;=_{df}\; \exists W \bullet B[W/V] \wedge \llbracket W = V \rrbracket \wedge (\overrightarrow{V}{=}\overrightarrow{V}) \wedge (\overleftarrow{W}{=} x1)$$

From Corollary 6.7 it follows that

(c1) $(A \wedge (\overrightarrow{V}{=} x2)) \Leftrightarrow (A1 \wedge (\overrightarrow{V}{=} x2))$

(c2) $(B \wedge (\overleftarrow{V}{=} x1)) \Leftrightarrow (B1 \wedge (\overleftarrow{V}{=} x1))$

$$\begin{array}{lll} & RHS & \{\text{Pred. Calculus}\} \\ \Leftarrow & \exists V \bullet ((A \wedge (\mathbf{true} \frown \lfloor V = x1 \rfloor^+) \wedge (\overrightarrow{V}{=} x2)) \frown & \\ & (B \wedge (\lfloor V = x2 \rfloor^+ \frown \mathbf{true}) \wedge (\overleftarrow{V}{=} x1))) & \{\text{(c1) and (c2)}\} \\ \Leftrightarrow & \exists V \bullet ((A1 \wedge (\mathbf{true} \frown \lfloor V = x1 \rfloor^+) \wedge (\overrightarrow{V}{=} x2)) \frown & \\ & (B1 \wedge (\lfloor V = x2 \rfloor^+ \frown \mathbf{true}) \wedge (\overleftarrow{V}{=} x1))) & \{\text{Theorem 5.11}\} \\ \Leftrightarrow & \exists V \bullet ((A1 \wedge (\mathbf{true} \frown \lfloor V = x1 \rfloor^+)) \frown (B1 \wedge (\lfloor V = x2 \rfloor^+ \frown \mathbf{true}))) & \\ & & \{\text{Theorem 6.4}\} \\ \Leftrightarrow & (\exists V \bullet A1 \wedge (\mathbf{true} \frown \lfloor V = x1 \rfloor^+)) \frown & \\ & (\exists V \bullet B1 \wedge (\lfloor V = x2 \rfloor^+ \frown \mathbf{true}))) & \{\text{Pred. Calculus}\} \end{array}$$

$$\Leftarrow (\exists V \bullet A1 \wedge (\mathbf{true} \frown \lfloor V = x1 \rfloor^+) \wedge (\overrightarrow{V} = x2)) \frown$$
$$(\exists V \bullet B1 \wedge (\lfloor V = x2 \rfloor^+ \frown \mathbf{true}) \wedge (\overleftarrow{V} = x1)) \qquad \{(c1) \text{ and } (c2)\}$$
$$\Leftrightarrow LHS \qquad \square$$

7 Chopping Points

Definition 7.1 (Super-dense chop)

Let V be the state variables used by formulae A and B. Define

$$A \circ B \ =_{df} \ \exists x, V_1, V_2 \bullet$$
$$(A[V_1/V] \ \wedge \ [\![V = V_1]\!] \ \wedge \ (\overleftarrow{V} = \overleftarrow{V_1}) \ \wedge \ (\overrightarrow{V_1} = x)) \frown$$
$$(B[V_2/V] \ \wedge \ [\![V = V_2]\!] \ \wedge \ (\overrightarrow{V} = \overrightarrow{V_2}) \ \wedge \ (\overleftarrow{V_2} = x)) \qquad \square$$

The chop operator $\frown$ is used to compose the continuously evolving hybrid systems, whereas the relational composition operator is used to model the sequential composition of imperative programming languages. The following theorem states that $\circ$ can be seen as the product of the chop operator $\frown$ and the relational composition operator.

Theorem 7.2

If $\overleftarrow{V}$ and $\overrightarrow{V}$ do not occur in A and B, then

$$(A \wedge p(\overleftarrow{V}, \overrightarrow{V})) \circ (q(\overleftarrow{V}, \overrightarrow{V}) \wedge B) \ = \ (A \frown B) \ \wedge \ \exists x \bullet (p(\overleftarrow{V}, x) \wedge q(x, \overrightarrow{V}))$$

Proof LHS $\{(hiding - 3)\}$

$$\Leftrightarrow \ \exists x \bullet (\exists V_1 \bullet (A[V_1/V] \wedge [\![V = V_1]\!]) \wedge$$
$$\exists V_1 \bullet ((\overleftarrow{V} = \overleftarrow{V_1}) \wedge (\overrightarrow{V_1} = x) \wedge p(\overleftarrow{V_1}, \overrightarrow{V_1}))) \frown$$
$$(\exists V_2 \bullet (B[V_2/V] \wedge [\![V = V_2]\!]) \wedge$$
$$\exists V_2 \bullet ((\overrightarrow{V} = \overrightarrow{V_2}) \wedge (\overleftarrow{V_2} = x) \wedge q(\overleftarrow{V_2}, \overrightarrow{V_2}))) \qquad \{\text{Corollary 6.7}\}$$
$$\Leftrightarrow \ \exists x \bullet ((A \wedge p(\overleftarrow{V}, x)) \frown (B \wedge q(x, \overrightarrow{V}))) \qquad \{\text{Theorems 5.9–5.10}\}$$
$$\Leftrightarrow \ RHS \qquad \square$$

Theorem 7.3 ($\circ$ and $\frown$)

If A does not refer to $\overrightarrow{V}$ and B does not use $\overleftarrow{V}$, then

$$A \circ B \ = \ A \frown B$$

Proof $A \circ B$ $\{(hiding-3)\}$

$\Leftrightarrow\ \exists x \bullet (\exists V_1 \bullet (A[V_1/V] \wedge [\![V = V_1]\!] \wedge (\overleftarrow{V}{=}\overleftarrow{V_1})) \wedge$
$(\exists V_1 \bullet (\overrightarrow{V_1}{=}x))) \frown (\exists V_2 \bullet (B[V_2/V] \wedge [\![V = V_2]\!] \wedge$
$(\overrightarrow{V}{=}\overrightarrow{V_2}))\ \wedge\ (\exists V_2 \bullet (\overleftarrow{V_2}{=}x)))$ $\{\text{Corollary 6.7}\}$

$\Leftrightarrow\ \exists x \bullet (A \frown B)$ $\{x \text{ is not free in } A \text{ and } B\}$

$\Leftrightarrow\ A \frown B$ □

$\circ$ also enjoys the following familiar algebraic laws:

Theorem 7.4

(1) (associativity) $(A \circ B) \circ C\ \Leftrightarrow\ A \circ (B \circ C)$

(2) (unit) $A \circ I\ \Leftrightarrow\ A\ \Leftrightarrow\ I \circ F$, where $I\ =_{df}\ (l = 0)\ \wedge\ (\overrightarrow{V}{=}\overleftarrow{V})$

(3) (disjunctivity) $A \circ (B1 \vee B2)\ \Leftrightarrow\ (A \circ B1) \vee (A \circ B2)$
$(A1 \vee A2) \circ B\ \Leftrightarrow\ (A1 \circ B) \vee (A2 \circ B)$

(4) (zero) $A \circ \mathbf{false}\ \Leftrightarrow\ \mathbf{false}\ \Leftrightarrow\ \mathbf{false} \circ A$

(5) (initial stable state) $(p(\overleftarrow{V}) \wedge A) \circ B\ \Leftrightarrow\ p(\overleftarrow{V}) \wedge (A \circ B)$

(6) (final stable state) $A \circ (B \wedge q(\overrightarrow{V}))\ \Leftrightarrow\ (A \circ B) \wedge q(\overrightarrow{V})$

(7) (consistency) $(A \wedge r(\overrightarrow{V})) \circ B\ \Leftrightarrow\ A \circ (r(\overleftarrow{V}) \wedge B)$

(8) (invisible stable state) If A does not use $\overrightarrow{V}$ and B does not mention $\overleftarrow{V}$

$$(A \wedge (\overrightarrow{V} = y)) \circ B\ \Leftrightarrow\ A \frown B$$
$$A \circ ((\overleftarrow{V} = y) \wedge B)\ \Leftrightarrow\ A \frown B$$
$$(A \wedge (\overrightarrow{V} = y)) \circ ((\overleftarrow{V} = z) \wedge B)\ \Leftrightarrow\ (A \frown B) \wedge (y = z)$$

Proof $(A \wedge (\overrightarrow{V}{=}y)) \circ ((\overleftarrow{V}{=}z) \wedge B)$ $\{(hiding-3)\}$

$\Leftrightarrow \exists x \bullet ((\exists V_1 \bullet (A[V_1/V] \wedge [\![V = V_1]\!] \wedge (\overleftarrow{V}{=}\overleftarrow{V_1}))$
$\wedge\ (\exists V_1 \bullet ((\overrightarrow{V_1}{=}y) \wedge (\overrightarrow{V_1}{=}x))))\ \frown$
$(\exists V_2 \bullet (B[V_2/V] \wedge [\![V = V_2]\!] \wedge (\overrightarrow{V}{=}\overrightarrow{V_2}))))$
$\wedge\ ((\exists V_2 \bullet ((\overrightarrow{V_2}{=}x) \wedge (\overleftarrow{V_2}{=}z)))$ $\{\text{Corollary 6.7}\}$

$\Leftrightarrow (\exists x \bullet ((x = y) \wedge (x = z)))\ \wedge\ (A \frown B)$ $\{\text{Pred. cal}\}$

$\Leftrightarrow (y = z)\ \wedge\ (A \frown B)$ □

8 Infinite Disjunction

Definition 8.1

Let $\mathcal{A}$ be a set of formulae. Define

$$\mathcal{M}_{[b,\,e]}(\bigvee \mathcal{A}) = tt \quad \textbf{if } \exists A \in \mathcal{A} \bullet \mathcal{M}_{[b,\,e]}(A) = tt$$

Define $(\bigwedge \mathcal{A}) \ =_{df} \ \neg(\bigvee\{\neg A \mid A \in \mathcal{A}\})$. □

$\bigvee \mathcal{A}$ is the *greatest lower bound* of $\mathcal{A}$ with respect to the order $\Rightarrow$ as shown in the following two laws.

($\bigvee$-1) (lower bound) If $A \in \mathcal{A}$, then $A \Rightarrow \bigvee \mathcal{A}$

($\bigvee$-2) (greatest lower bound) If $A \Rightarrow B$ for all members A of $\mathcal{A}$, then $\bigvee \mathcal{A} \Rightarrow B$

The following laws enable us to push $\bigvee$ outwards.

($\bigvee$-3) $(\bigvee \mathcal{A}) \frown B \ \Leftrightarrow \ \bigvee \{(A \frown B) \mid A \in \mathcal{A}\}$

$B;(\bigvee \mathcal{A}) \ \Leftrightarrow \ \bigvee \{(B \frown A) \mid A \in \mathcal{A}\}$

($\bigvee$-4) $\exists x \bullet (\bigvee \mathcal{A}) \ \Leftrightarrow \ \bigvee\{\exists x \bullet A \mid A \in \mathcal{A}\}$

$\exists V \bullet (\bigvee \mathcal{A}) \ \Leftrightarrow \ \bigvee\{\exists V \bullet A \mid A \in \mathcal{A}\}$

Theorem 8.2

$(\bigvee \mathcal{A}) \Rightarrow B$ **iff** $\forall A \in \mathcal{A} \bullet (A \Rightarrow B)$

Proof $(\bigvee \mathcal{A}) \Rightarrow B$ $\{(\bigvee -1)$ transitivity of $\Rightarrow\}$

$\Longrightarrow \ \forall A \in \mathcal{A} \bullet A \Rightarrow B$ $\{(\bigvee -2)\}$

$\Longrightarrow \ (\bigvee \mathcal{A}) \Rightarrow B$ □

Theorem 8.3

(1) $(\bigvee \mathcal{A})/(l = c) \ \Leftrightarrow \ \bigvee \{A/(l = c) \mid A \in \mathcal{A}\}$

(2) $(l = c)\backslash(\bigvee \mathcal{A}) \ \Leftrightarrow \ \bigvee \{(l = c)\backslash A \mid A \in \mathcal{A}\}$

Proof From (WCI-1). □

Definition 8.4 (Weakest fixed point)

Let F be a monotonic mapping on formulae. Define

$$\mu X \bullet F \ =_{df} \ \bigvee\{A \mid A \Rightarrow F(A)\}$$ □

Theorem 8.5

(1) $\mu X \bullet F \ \Leftrightarrow \ F(\mu X \bullet F)$

(2) If $A \Rightarrow F(A)$, then $A \Rightarrow \mu X \bullet F$ □

9 Duration Calculus with Iteration

This section shows how to embed DC^* [2] into our framework.

Definition 9.1 (Iteration)
Define $A^* \ =_{df} \ \bigvee \{A^n \mid n \in NAT\}$
where $A^0 \ =_{df} \ (l = 0)$ and $A^{n+1} \ =_{df} \ A \frown A^n$ □
(DC_1^*) $(l = 0) \Rightarrow A^*$

Proof From $(\bigvee -1)$. □
(DC_2^*) $(A^* \frown A) \Rightarrow A^*$.

Proof
$$\begin{array}{lll} & A^* \frown A & \{(\bigvee -3)\} \\ \Leftrightarrow & \bigvee \{A^{n+1} \mid n \in NAT\} & \{(\bigvee -1) \text{ and } (\bigvee -2)\} \\ \Rightarrow & A^* & \end{array}$$ □

(DC_3^*) $(A^* \wedge B) \frown \mathbf{true} \Rightarrow (B \wedge l = 0) \frown \mathbf{true} \ \vee \ (((A^* \wedge \neg B) \frown A) \wedge B) \frown \mathbf{true}$

Proof
$$\begin{array}{lll} & (A^{n+1} \wedge B) \frown \mathbf{true} & \{(\frown -4)\} \\ \Longrightarrow & (((A^n \wedge \neg B) \frown A) \wedge B) \frown \mathbf{true} \ \vee ((A^n \wedge B) \frown A) \wedge B) \frown \mathbf{true} & \\ & & \{(\bigvee -1) \text{ and } (\frown -5)\} \\ \Longrightarrow & (((A^* \wedge \neg B) \frown A) \wedge B) \frown \mathbf{true} \vee & \\ & ((A^n \wedge B) \frown \mathbf{true}) \wedge \mathbf{true}) \frown \mathbf{true} & \{(\frown -2) \text{ and } (\frown -5)\} \\ \Longrightarrow & RHS \ \vee \ ((A^n \wedge B) \frown \mathbf{true}) & \end{array}$$

By induction and the fact that $(A^0 \wedge B) \frown \mathbf{true} \Rightarrow RHS$ we conclude that for all $n \in NAT$

$$(A^{n+1} \wedge B) \frown \mathbf{true} \Rightarrow RHS$$

from which and $(\bigvee -3)(1)$ follows the conclusion. □

Theorem 9.2 (Induction)
If $(l = 0) \Rightarrow B$ and $(A \frown B) \Rightarrow B$, then $A^* \Rightarrow B$

Proof We are going to show that $A^n \Rightarrow B$ for all $n \in NAT$.

$n = 0$: From the assumption.

$$\begin{array}{lll} n = 1 : & A & \{(\frown -2)\} \\ \Leftrightarrow & A \frown (l = 0) & \{(\frown -5)\} \\ \Rightarrow & A \frown B & \{\text{assumption}\} \\ \Rightarrow & B & \end{array}$$

$$
\begin{array}{lll}
n = k+1 : A^{k+1} & & \{\text{Def of } A^{k+1}\} \\
\quad \Leftrightarrow & A \frown A^k & \{(\frown -5) \text{ and induction hypothesis}\} \\
\quad \Rightarrow & B & \square
\end{array}
$$

Corollary 9.3

$A^* \;\Leftrightarrow\; (l = 0) \vee A \frown A^*$

Proof From (DC^*-1) and (DC^*-2) it follows that $RHS \Rightarrow LHS$. The opposite inequation follows from Theorem 9.2. □

10 Conclusion

This paper presents a logical language which integrates many variants of DC, and acts a wide spectrum language covering specification, design and programming of real-time computing systems. We have investigated the links of our language with some well-known variants of DC in the previous sections. Our language provides a mathematically sound basis for real-time refinement calculus and as well as proof systems for time-critical computing systems. It has been successfully used in formalising a specification language TRSL [10], mixed hardware/software systems [7] and Sequential Hybrid Systems [8].

References

1. Rana Barua and Zhou Chanchen. Neighbourhood Logics. UNU/IIST Report No 120, (1997)
2. Dang Van Hung and D.P. Guelev. Completeness and Decidability of a Fragment of Duration Calculus with Iteration. LNCS 1742, 139–150, (1999)
3. E.W. Dijkstra. A Discipline of Programming. Prentice Hall, (1976).
4. Mike Gordon. *The Semantic Challenge of Verilog HDL.* in Proc. of LICS'95, San Diego, California. (1995).
5. Mike Gordon. *Event and Cyclic Semantics of Hardware Description Languages* Technical Report of the Verilog Formal Equivalence Project of Cambridge Computing Laboratory, (1995).
6. M. Hansen, P. Pandya and Zhou Chaochen. Finite divergence Theoretical Computer Science 138, 113–139, (1995).
7. He Jifeng A Common Framework for Mixed Hardware/Software Systems. In Proc. of IFM'99, Springer-Verlag, 1–25, (1999).
8. He Jifeng and Xu Qiwen. Advance Features of DC and their applications in Sequential Hybrid Systems Formal Aspect of Computing 15: 84-99, (2003)
9. C.A.R. Hoare and He Jifeng Unifying Theories of Programming. Prentice Hall, (1998)
10. Li Li and He Jifeng A DC Semantic for Timed RSL. In Proc. of RTCSA'99, 492-504, (1999)
11. O. Maler, Z. Manna and A. Pnueli. From timed to hybrid systems. LNCS 600, (1992).

12. Carroll Morgan Programming from Specifications. Second Edition. Prentice Hall, (1994).
13. B.C. Moszkowski. A temporal logic for multi-level reasoning about hardware IEEE Computer 18(2), 10–19.
14. B.C. Moszkowski. Executing Temporal Logic Programs. Cambridge University Press, (1986).
15. B.C. Moszkowski. Compositional reasoning about projected and infinite time. In Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems, 238–245, (1995).
16. Open Verilog International (OVI). Verilog Hardware Description Language Reference Manual. Version 1, (1994)
17. P. Pandya and Y. Ramakrishna. A recursive duration calculus Technical Report, CS-95/3, Computer Science Group, TIFR, Bombay, (1995)
18. P. Pandya and V.H. Dang. Duration calculus with weakly monotonic time. In Proc. of FTRTFT'98, LNCS 1486, (1998)
19. P. Pandya, H.P. Wang and Q.W. Xu. Towards a Theory of Sequential Hybrid Programs. In Proc. of IFIP Conference, Chapman & Hall, (1998).
20. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics Vol 5: 285–309, (1955)
21. D.E. Thomas and P. Moorby. The VERILOG Hardware Description Language. Kluwer Publisher, (1991).
22. Zhou Chaochen, C.A.R. Hoare and A.P. Ravn. A calculus of duration. Information Processing Letters 40(5): 269–275, (1991).
23. Zhou Chaochen, A. Ravn and M. Hansen. An extended duration calculus for hybrid systems Lecture Notes for Computer Sciences 736, 36–59, (1993)
24. Zhou Chaochen and Li Xiaoshan. A Mean Value Calculus of Durations. In A.W Roscoe (ed): "A Classical Mind: Essays in Honour of C.A.R. Hoare", Prentice-Hall, 431–451, (1994)
25. Zhou Chaochen and M. Hansen. Chopping a point. In Proc. of BCS FACS 7th Refinement Workshop, Electronic Workshop in Computer Sciences, Springer-Verlag, (1996).
26. Zhou Chaochen, D.P. Guelev and Zhan Naijun. A Higher-Order Duration Calculus. In J.Davies et al (eds) "Millennial Perspectives in Computer Science", 407–416, (1999)

Challenges in Increasing Tool Support for Programming

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. Programming in the large is difficult, in part because the number of details that one must get right is enormous. Various tools can assist programmers in managing the details. These tools include a methodology that formalizes useful programming idioms, a language in which programmer design decisions can be expressed, and static and dynamic checkers that look for errors or attempt to prove the absence thereof. In this talk, I will discuss challenges in each of these areas. I will also give a short demo of a prototype of the Spec# programming system, which takes on these challenges and is designed to be used in practice.

Joint work with Mike Barnett, Robert DeLine, Manuel Fähndrich, Peter Müller, David A. Naumann, Wolfram Schulte, and Herman Venter.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, p. 35, 2005.
© Springer-Verlag Berlin Heidelberg 2005

A Predicate Spatial Logic and Model Checking for Mobile Processes*

Huimin Lin

Laboratory for Computer Science,
Institute of Software, Chinese Academy of Sciences
lhm@ios.ac.cn

Abstract. Mobile processes involve not only in time but also in *space*. Traditionally model checking has mainly concerned with temporal properties of processes, but recently proposals have been put forwarded to check spatial properties as well. In this talk we shall first present a modal logic for describing temporal as well as spatial properties of mobile processes expressed in the asynchronous π-calculus. The logic is first-order and has quantifies including the *fresh name quantifier* for handling the name restriction construct in the π-calculus. The novelty of the logic is that fixpoint formulas are constructed as *predicates* which are functions from names to propositions. The semantics of the logic is developed and shown to be monotonic, thus guarantees the existence of fixpoints. We then propose an algorithm to automatically check if a process has properties described as formulas in the logic, and establish its correctness.

* Supported by research grants from Natural Science Foundation of China and Chinese Academy of Sciences.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, p. 36, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes*

— **Extended Abstract** —

Erika Ábrahám[2], Marcello M. Bonsangue[3], Frank S. de Boer[4], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] University Freiburg, Germany
[3] University Leiden, The Netherlands
[4] CWI Amsterdam, The Netherlands

Abstract. The concurrent object calculus has been investigated as a core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects. The combination of this form of concurrency with objects corresponds to features known from the popular language *Java*. One distinctive feature, however, of the concurrent object calculus is that it is *object-based*, whereas the mainstream of object-oriented languages is *class-based*.

This work explores the semantical consequences of introducing classes to the calculus. Considering classes as part of a component makes instantiation a possible interaction between component and environment. A striking consequence is that to characterize the observable behavior we must take *connectivity* information into account, i.e., the way objects may have knowledge of each other. In particular, unconnected environment objects can neither determine the absolute order of interaction and furthermore cannot exchange information to compare object identities.

We formulate an operational semantics that incorporates the connectivity information into the scoping mechanism of the calculus. As instantiation itself is unobservable, objects are instantiated only when accessed for the first time ("*lazy instantiation*").

Furthermore we use a corresponding trace semantics for full abstraction wrt. a may-testing based notion of observability.

Keywords: multithreading, class-based object-oriented languages, formal semantics, full abstraction.

1 Introduction

The notion of component is well-advertised as structuring concept for software development. Even if there is not too much agreement about what constitutes a

* Part of this work has been financially supported by the IST project Omega (IST-2001-33522) and the NWO/DFG project Mobi-J (RO 1122/9-1/2).

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 37–51, 2005.
© Springer-Verlag Berlin Heidelberg 2005

component in concrete software engineering terms, one aspect should go undisputed: At the bottom line, a component means a "program fragment" being *composed,* which raises the question what the semantics of a component is. A natural approach is to take an observational point of view: two components are observably equivalent, when no observing context can tell them apart.

In the context of concurrent, *object-based* programs and starting from may-testing as a simple notion of observation, Jeffrey and Rathke [7] provide a fully abstract trace semantics for the language. Their result roughly states that, given a component as a set of objects and threads, the fully abstract semantics consists of the set of traces at the boundary of the component, where the traces record incoming and outgoing calls and returns. At this level, the result is as one would expect, since intuitively in the chosen setting, the only possible way to observe something about a set of objects and threads is by exchanging messages.

The result in [7] is developed within the concurrent object calculus [5], an extension of the sequential ν-calculus [10] which stands in the tradition of various object calculi [1] and also of the π-calculus [9, 11]. The chosen language has been proposed as core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects, but distinctive feature is that it is *object-based,* which in particular means that there are no *classes* as templates for new objects. This is in contrast to the mainstream of object-oriented languages where the code is organized in classes, one well-known example being *Java*. This work addresses therefore the following question:

> What changes when switching from an object-based to a class-based setting, a setting which corresponds to features as found in a language like multithreaded *Java* or $C^{\#}$?

Considering the observable behavior of a component, we have to take into account that in addition to objects, which are the passive entities containing the instance state, and threads, which are the active entities, *classes* come into play. Classes serve as a blueprint for their instances and can be conceptually understood as particular objects supporting just a method which allows to generate instances. Indeed, ultimately, the observer consists only of classes since the program code is structured into classes, and objects exist only at run-time.

Crucial in our context is that now the division between the program fragment under observation and its environment also separates *classes:* There are classes internal to the component and those belonging to the environment. As a consequence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well. This possibility of *cross-border instantiation* is absent in the object-based setting: Objects are created by directly providing the code of their implementation, not referring to the name of a class, which means that the component creates only component-objects and dually the environment only environment objects.

To understand the bearing of this change on what is observable, we concentrate on the issue of instantiation across the demarcation line between component and its environment. The environment is considered as the observing context which tries to determine the behavior of the component or program

under observation. So imagine that the component creates an instance of an environment class, and the first question is: does this yield a component object or an environment object? As the code of the object is in the hand of the observer, namely being provided by the external class, the further interaction between the component and the newly created object can lead to observable effects and must thus be part of the behavior at the component's interface. In other words, instances of environment classes belong to the environment, and dually those of internal classes to the component.

To obtain a semantics which is abstract enough, it is crucial not just to cover all possible interface behavior —there is little doubt that sequences of calls, returns, and instantiations with enough information at the labels would do— but to capture it *exactly,* i.e., to exclude impossible environment interaction. As an obvious example: two consecutive calls from the same thread without outgoing communication in between cannot be part of the component behavior.

Whereas in the above situation, the object is instantiated to be part of the environment, the *reference* to it is kept at the creator, for the time being. So in case an object of the program, say o_1 instantiates two objects o_2 and o_3 of the environment, the situation informally looks as shown in Figure 1, where the dotted bubbles indicate the scope of o_2, respectively of o_3.

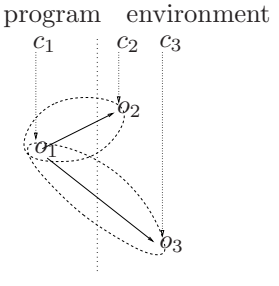


Fig. 1. Instances of external classes

In this situation, an incoming call from the environment carrying both names o_2 and o_3 is impossible, as the only entity aware of both references is o_1. Unless the component gives away the references to the environment, o_2 and o_3 are and remain completely separated.

Thus, to exclude impossible combinations of object references in the communication labels, the component must keep track of which objects of the environment are connected. The component has, of course, by no means full information about the complete system; after all it can at most trace what happens at the interface, and the objects of the environment can exchange information "behind the component's back". Therefore, the component must conservatively overapproximate the potential knowledge of objects in the environment, i.e., it must make *worst-case assumptions* concerning the proliferation of knowledge, which means it must assume that

1. once a name is out, it is never forgotten, and
2. if there is a possibility that a name is leaked from one environment object to another, this will happen.

Sets of environment objects which can possibly be in contact with each other form therefore equivalence classes of names —we call them *cliques*— and the formulation of the semantics must contain a representation of them. New cliques can be created, as new objects can be instantiated without contact to others, and

furthermore cliques can merge, if the component leaks the identity of a member of one clique to a member of another.

This paper investigates a class-based variant of the object calculus, formalizing the ideas sketched above about cliques of objects. Instantiation itself, even across the environment-program boundary, is unobservable, since the calculus does not have constructor methods. In the semantics, an externally instantiated object is created only at the point when it is actually accessed the for the first time, which we call "*lazy instantiation*". For want of space, we concentrate here on the intuition and stress the differences to the object-based setting. For deeper coverage we refer to the technical reports [2] and [3].

The paper is organized as follows. Section 2 contains the syntax of the calculus in which the result is presented, and a sketch of its semantics. In particular, the notions of lazy instantiation and connectivity of objects are formalized. Afterwards, Section 3 elaborates on the trace semantics, Section 4 fixes the notion of observability, and Section 5 states the full abstraction result. Finally in Section 6, we discuss related work.

2 A Concurrent Class Calculus

In this section, we present the calculus used in our development. As we concentrate on the semantical issues of connectivity of objects and the interface behavior of a component, we only sketch the syntax, ignore typing issues and also omit structural equivalence rules, as they are rather standard. As mentioned, the reader will find details in the accompanying technical report.

The calculus is a syntactic extension of the concurrent object calculus from [5, 7]. The basic change is the introduction of *classes,* where a class is a named collection of methods. In contrast to object references, class names are literals introduced when defining the class; they may be hidden using the ν-binder but unlike object names, the scopes for class names are *static.* Object names, on the other hand, are first-order citizens of the calculus in that they can be stored in variables, passed to other objects as method parameters, making the scoping *dynamic,* and especially they can be created freshly by instantiating a class.

A program is given by a collection of classes. A class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields. An object $o[c, F]$ stores the current value of the fields or instance variables and keeps a reference to the class it instantiates. A method $\varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_n{:}T_k).t$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. Besides named objects and classes, the dynamic configuration of a program can contain as active entities *named threads* $n\langle t\rangle$, which, like objects, can be dynamically created. Unlike objects, threads are not instantiated by some statically named entity (a "thread class" as in *Java*), but directly created by providing the code. A thread basically is either a value (especially a reference to another named entity) or a sequence of expressions, notably method calls (written $o.l(\boldsymbol{v})$) and creation of new objects and new threads ($new\, c$ and $new\langle t\rangle$ where c is a class name and t a thread). We will generally use n and its syntactic

Table 1. Abstract syntax

C	$::= \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F] \mid n\langle t\rangle$	program
O	$::= M, F$	object
M	$::= l = m, \ldots, l = m$	method suite
F	$::= l = f, \ldots, l = f$	fields
m	$::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
f	$::= \varsigma(n{:}T).\lambda().v$	field
t	$::= v \mid \mathit{stop} \mid \mathit{let}\ x{:}T = e\ \mathit{in}\ t$	thread
e	$::= t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e$	expr.
	$\mid\ v.l(v, \ldots, v) \mid n.l := v \mid \mathit{currentthread}$	
	$\mid\ \mathit{new}\, n \mid \mathit{new}\langle t\rangle$	
v	$::= x \mid n$	values

variants as name for threads (or just in general for names), o for objects, and c for classes. Furthermore we will use f specifically for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update[1] as $f.x := v$.

Concerning the *operational semantics* of the calculus, the basic steps are mainly given in two levels: *internal* steps whose effect is confined within a component, and those with external effect. Interested mainly in the external behavior we elide the definition of the internal steps.

The *external* behavior of a component is given in terms of labeled transitions describing the communication at the interface of an *open* program. For the completeness of the semantics, it is crucial ultimately to consider only communication traces realizable by an actual program context which, together with the component, yields a well-typed closed program.

The concentration on actually realizable traces has various aspects, e.g., the transmitted values need to adhere to the static typing assumptions, only publicly known objects can be called from the outside, and the like. Being concerned with the dynamic relationship among objects, we omit also these aspects here. Besides that, this part is rather standard and also quite similar to the one in [7].

2.1 Connectivity Contexts and Cliques

The informal discussion in the introduction argued that in the presence of internal and external classes and cross-border instantiation, the component must *keep track* of which identities it gives away to which objects in order to exclude impossible behavior as described for instance in connection with Figure 1. The external semantics is formalized as labeled transitions between judgments of the form

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \ , \tag{1}$$

where $\Delta; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta; E_\Theta$ the *commitments*; alternative names are the required and the pro-

[1] We don't use general method update as in the object-based calculus.

vided interface of the component. The assumptions consist of a part Δ concerning the existence (plus static typing information) of *named entities* in the environment. For the book-keeping of which objects of the environment have been told which identities, a well-typed component must take into account the *relation* of object names from the assumption context Δ amongst each other, and the knowledge of objects from Δ about those exported by the component, i.e., those from Θ. [2] In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component:

$$E_\Delta \subseteq \Delta \times (\Delta + \Theta) \ . \tag{2}$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$. We write $o_1 \hookrightarrow o_2$ ("o_1 may know o_2") for pairs from these relations. As mentioned, the component does not have full information about the complete system and thus it must make worst-case assumptions concerning the proliferation of knowledge. These worst-case assumptions are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of objects *from* Δ the component maintains. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

$$\leftrightharpoons \ \triangleq \ (\hookrightarrow\!\downarrow_\Delta \cup \hookleftarrow\!\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ . \tag{3}$$

Note that we close $\hookrightarrow$ only wrt. environment objects, but not wrt. objects at the *interface,* i.e., the part of $\hookrightarrow \subseteq \ \Delta \times \Theta$. We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\!\hookrightarrow$ for that union. As judgment, we use $\Delta; E_\Delta \vdash v_1 \leftrightharpoons v_2 : \Theta$, respectively $\Delta; E_\Delta \vdash v_1 \leftrightharpoons\!\!\hookrightarrow v_2 : \Theta$. For Θ, E_Θ, and Δ, the definitions are applied dually.

The relation $\leftrightharpoons$ is an equivalence relation on the objects from Δ and partitions them into equivalence classes. As a manner of speaking, we call a set of object names from Δ (or dually from Θ) such as for all objects o_1 and o_2 from that set, $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$, a *clique*, and if we speak of *the* clique of an object we mean the whole equivalence class.

2.2 External Steps

The external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments (cf. Table 3). Besides internal steps a component exchanges information with the environment via *calls* and *returns.* Using a lazy instantiation scheme for cross-border object creation, there are no separate external labels for *new*-steps. Thus, core labels γ are of the form $n\langle \mathit{call}\ o.l(\boldsymbol{v})\rangle$ and $n\langle \mathit{return}(v)\rangle$. Names may occur bound in a label $\nu(n{:}T).\gamma$, and receiving and sending labels are written as $\gamma?$ and $\gamma!$. In this extended abstract, we omit the typing premises in the operational rules ("only values consistent with the static typing assumptions may be

[2] Besides the relationships amongst objects, we need to keep one piece of information concerning the "connectivity" of *threads*. To exclude situations where a known thread leaves the component into one clique of objects but later returns to the component coming from a different clique without connection to the first, we remember for each thread that has left the component the object from Δ it has left into.

Table 2. Labels

$\gamma ::= n\langle call\ o.l(\boldsymbol{v})\rangle \mid n\langle return(v)\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	receive and send labels

Table 3. External steps

$$\frac{a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\boldsymbol{v})\rangle? \quad dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\boldsymbol{v})\rangle) \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; n \hookrightarrow o_2 \hookrightarrow \boldsymbol{v}) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \quad \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \boldsymbol{v}, o_2 : \acute{\Theta} \quad t_{blocked} = let\ x{:}T' = o_2'\ blocks\ for\ o_1'\ in\ t}{\begin{array}{l}\Delta; E_\Delta \vdash C \parallel n\langle t_{blocked}\rangle : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_2.l(\boldsymbol{v})\ in\ o_2\ return\ to\ o_1\ x; t_{blocked}\rangle : \acute{\Theta}; \acute{E}_\Theta\end{array}}\ \textsc{CallI}_2$$

$$\frac{a = \nu(\Theta', \Delta').\ n\langle return(v)\rangle! \quad (\Theta', \Delta') = fn(v) \cap \Phi \quad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; (n \hookrightarrow o_1 \hookrightarrow v) \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(\acute{C}, \Theta') \setminus n}{\begin{array}{l}\Delta; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_2\ return\ to\ o_1\ v\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle t\rangle) : \acute{\Theta}; \acute{E}_\Theta\end{array}}\ \textsc{RetO}$$

$$\frac{a = \nu(\Theta', \Delta').\ n\langle call\ o_2.l(\boldsymbol{v})\rangle! \quad (\Theta', \Delta') = fn(n\langle call\ o_2.l(\boldsymbol{v})\rangle \cap \Phi \quad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \quad o_2 \in dom(\acute{\Delta}) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; (n \hookrightarrow o_2 \hookrightarrow \boldsymbol{v}) \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(\acute{C}, \Theta') \setminus n}{\begin{array}{l}\Delta; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = [o_1]\ o_2.l(\boldsymbol{v})\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle) : \acute{\Theta}; \acute{E}_\Theta\end{array}}\ \textsc{CallO}$$

$$\frac{a = \nu(\Delta', \Theta').\ n\langle return(v)\rangle? \quad dom(\Delta', \Theta') \subseteq fn(v) \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta', (n \hookrightarrow o_1 \hookrightarrow v) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta', (o_2 \hookrightarrow (\Delta', \Theta')) \setminus n \quad \acute{\Delta}; \acute{E}_\Delta \vdash o_2 \leftrightharpoons\!\hookrightarrow v : \acute{\Theta}}{\Delta; E_\Delta \vdash C \parallel n\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel n\langle t[v/x]\rangle : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{RetI}$$

$$\frac{c \in dom(\Delta)}{\Delta; E_\Delta \vdash n\langle let\ x{:}c = new\ c\ in\ t\rangle : \Theta; E_\Theta \rightsquigarrow \Delta; E_\Delta \vdash \nu(o_3{:}c).n\langle let\ x{:}c = o_3\ in\ t\rangle : \Theta; E_\Theta}\ \textsc{NewO}_{lazy}$$

received" and the like) as they are straightforward and we concentrate on the novel aspects, namely the connectivity information.

Connectivity Assumptions and Commitments. As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ overapproximates the actual connectivity of the component, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. *Incoming* calls update the commitment context E_Θ in that it remembers that the callee o_2 now knows (or rather may know) the arguments $\boldsymbol{v}$, and furthermore that the thread n has entered o_2. For incoming communication (cf.

rules CALLI$_2$ and RETI)[3] we require that the sender be acquainted with the transmitted arguments.

For the role of the caller identity o_1, a few more words are in order. The antecedent of the call-rules requires, that the caller o_1 is acquainted with the callee o_2 and with all of the arguments. However, the caller is *not* transmitted in the label which means that it remains anonymous to the callee.[4] To gauge, whether an incoming call is possible and to adjust the book-keeping about the connectivity appropriately, in particular when returning later, the transition chooses among possible sources of the call. With the sole exception of the initial (external) step, the scope of at least *one* object of the calling clique must have escaped to the component, for otherwise there would be now way of the caller to address o_2 as callee. In other words, for at least one object o_1 from the clique of the actual caller (which remains anonymous), the judgment $\Delta \vdash o_1{:}c$ holds prior to the call. Furthermore it must be checked that the incoming thread originates from a group of objects in connection with the one to which the thread had left the component the last time: $\acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 : \acute{\Theta}$. Once chosen, the assumed identity of the caller is remembered as part of the return-syntax.

It is worth mentioning that in rule RETI the proviso that the callee o_2 knows indirectly the caller o_1, i.e., $\Delta; E_\Delta \vdash o_2 \leftrightharpoons\!\!\hookrightarrow o_1 : \Theta$ is not needed. Neither is it necessary to require in analogy to the situation for the incoming call that the thread is acquainted with the callee. If fact, both requirements will be automatically assured for traces where calls and returns occur in correct manner.

A commonality for incoming communications from a thread n is that the (only) pair $n \hookrightarrow o$ for some object reference o is removed from E_Δ, for which we write $E_\Delta \setminus n$. While E_Δ imposes restrictions for incoming communication, the commitment context E_Θ is *updated* when receiving new information. For instance in CALLI$_2$, the commitment $\acute{E}_\Theta$ after reception marks that now the callee o_2 is acquainted with the received arguments and furthermore that the thread n is visiting (for the time being) the callee o_2. For *outgoing* communication, the E_Δ and E_Θ play dual roles. In the respective rules, $E(\acute{C}, \Theta')$ stands for the actual connectivity of the component after the step, which needs to be made public in the commitment context, in case new names escape to the environment.

Scoping and Lazy Instantiation. In the explanation so far, we omitted the handling of bound names, in particular bound object references. In the presence of classes, a possible interaction between component and environment is instantiation. Without constructor methods and assuming an infinite heap space,

[3] We omit rules dealing with the initial situation where the first thread crosses the interface between environment and component.

[4] Of course, the caller may transmit its identity to the callee as part of the arguments, but this does not reveal to the callee who "actually" called. Indeed, the actual identity of the caller is not needed; it suffices to know the *clique* of the caller. As representative for the clique, an equivalence class of object identities, we simply pick one object.

instantiation itself has no immediate, observable side-effect. An observable effect is seen only at the point when the object is accessed.

Rule NEWO$_{lazy}$ describes the local instantiation of an external class. Instead of exporting the newly created name of the object plus the object itself immediately to the environment, the name is kept local until, if ever, it gets into contact with the environment. When this happens, the new instance will not only become known to the environment, but the object will also be instantiated in the environment.

For incoming calls, for instance, the binding part is of the form (Δ', Θ') where we mean by convention, that Δ' are the names being added to Δ, and analogously for Θ' and Θ. For object names, the distinction is based on the class types. For thread names, the reference is contained in Δ' and Θ', and class names are never transmitted. For the object names in the incoming communication Δ' contains the external references which are freshly introduced to the component by scope extrusion. Θ' on the other hand are the objects which are *lazily instantiated* as side-effect of this step, and which are from then on part of the component. In the rules, the newly instantiated objects are denoted as $C(\Theta')$.

Note that whereas the acquaintance of the caller with the arguments transmitted free is checked against the current assumption, acquaintance with the ones transmitted bound is added to the assumption context.

3 Trace Semantics and Ordering on Traces

Next we present the semantics for well-typed components, which, as in the object-based setting, takes the sequences of external steps of the program fragment as starting point.

Not surprisingly, a major complication now concerns the connectivity of objects. In this context, the caller identity, while not visible by the callee, plays a crucial role in keeping track of assumed connectivity, in particular to connect the effect of a return to a possible caller clique. To this end, the operational semantics hypothesizes about the originator of incoming calls and remembers the guess as "auxiliary" annotation in the code for return (cf. rule L-CALLI$_2$ from Table 3).

The (hypothetical) connectivity of the environment influences what is observable. Very abstractly, the fact the observer falls into a number of independent cliques increases the "uncertainty of observation". We can point to two reasons responsible for this effect. One is that separate observer cliques cannot determine the relative order of events concerning only one of the environment cliques. To put it differently: a clique of objects can only observe the order of events *projected* to its own members. We will worry about this later when describing the all possible reorderings or interleavings of a given trace. Secondly, separate observers cannot cooperate to *compare identities.* This means, as long as separated, the observers cannot find out whether identities sent to each of them separately are the same or not. In terms of projections to the observing clique it means that local projections are considered up to α-conversion, only.

The above discussion should not mislead us to think that the behavior of two observing cliques is completely independent. One thing to keep in mind is that the observers can merge. This means that identities, separate and local prior to the merge, become comparable, and the now joint clique can find out whether local interaction of the past used the same identities or not. The absolute order of local events of the past, however, cannot be reconstructed after merging.

Another more subtle point, independent from merging of observers, is that to a certain degree, the events local to one clique *do influence* interaction concerning another clique. This in other words implies that considering *only* the separate local projections of a global behavior to the observers is too abstract to be sound.

To understand the point, consider as informal example a situation of a component C_1 with two observing cliques in the environment and a sequence s of labels at the interface of the component being observed. Assume further that s_1 is the projection of s to the first observer and s_2 the projection to the second, and assume that $s = s_1 s_2$ meaning that s_1 precedes s_2 when considered as global behavior. For sake of the argument, assume additionally that C_1 is not able to perform the interaction in the swapped order $s_2 s_1$. Given a second component C_2 being more often successful, i.e., that $C_1 \sqsubseteq_{may} C_2$, what does this imply for C_2's behavior? The definition of may-preorder is given in Section 4. For the moment, being successful can be thought of being able to reach some predefined point which counted as success.

Since the environment can be programmed in such a way that it reports success only after completing s_1 resp. s_2, it is intuitively clear that C_2 must be able to exhibit s_1 resp. s_2. But the environment cannot observe whether C_2 performs s_1 and s_2 *in the same run,* as does C_1. We can only be sure that there is a run of C_2 which is able to do s_1 and a (potentially different) one which does s_2, each of which is taken as independent sign of success. This does not mean, however, that the order of $s_1 s_2$ does not play a role at all. Consider for illustration the situation where C_2 can perform $s_2 s_1$ but not $s_1 s_2$ as C_1: In this case, $C_1 \not\sqsubseteq_{may} C_2$, i.e., C_2 is not successful while C_1 is, namely in an environment where s_2 is possible and reports success but s_1 *can be hindered from completion.* In other words, taking the behavior $s_1 s_2$ of C_1 as starting point we cannot consider in isolation the fact that s_2 is possible by C_2 as well, the order of s_1 preceding s_2 is important inasmuch as it s_1 can *prevent* success for s_2. So $C_1 \not\sqsubseteq_{may} C_2$ and the fact that C_1 performs the sequence $s_1 s_2$ means, that C_2 can perform s_2 after a prefix of s_1. Since the common environment has already proven in cooperation with C_1 that it is able to perform s_1, it cannot prevent success of C_2 by blocking.

To sum up and independently of the details: to capture the observable behavior appropriately, we need to be able to define the projection of the external steps to the observer cliques. Now the labels for method calls in the external semantics do not contain information concerning the caller, which means given a

trace as a sequence of labels, we have no indication for incoming calls concerning the originating environment clique.[5]

A way to remedy this lack of information is to augment the labels as recorded in the traces by the missing information. So instead of the call label described in Section 2.2, we use $n\langle[o_1]\mathit{call}\ o_2.l(\boldsymbol{v})\rangle$ as annotated call label, where o_1 denotes the caller, respectively the clique of the caller. The augmented transitions are generated simply by using the caller rules from Table 3 where the caller is added to the transition labels in the obvious way.

A trace of a well-typed component is a sequence s of external steps, where we write $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{s}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$. For $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{\epsilon}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$, we write shorter $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \Longrightarrow \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$.

With this information we can define the *projection* of a trace onto a clique as the part of the sequence containing all the labels with objects from that clique. Remember that a clique of an object $o \in \Theta$ consists of all objects from Θ acquainted with o. Thus the equivalence $\leftrightharpoons$ partitions Θ into equivalence classes, and formally we could write $[o]_{/_{E_\Theta}}$ or $[o]_{/_{\leftrightharpoons}}$ for that equivalence class. For simplicity, we often just write $[o]$.

The definition of projection of an (augmented) trace onto a clique of environment objects is straightforward, one simply jettisons all actions not belonging to that clique. One only has to be careful dealing with exchange of bound names, i.e., scope extrusion, since names sent for the first time to a clique are to be considered as *locally fresh,* even if the name may globally be known to other environment cliques.

We can now define the order on traces as follows.

Definition 1. *$\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, if the following holds. If $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \stackrel{sa}{\Longrightarrow} \Delta'; E'_\Delta \vdash C'_1 : \Theta'; E'_\Theta$, then $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta \stackrel{t}{\Longrightarrow} \Delta''; E''_\Delta \vdash C''_2 : \Theta''; E''_\Theta$ such that*

- *$t \downarrow_{[o'']} = sa \downarrow_{[o_a]}$ for some clique $[o'']$ according to $\Theta''; E''_\Theta$ and when $[o_a]$ is the environment clique to which label a belongs, and*
- *for all cliques $[o'']$ according to $\Delta''; E''_\Delta$, there exists a clique $[o']$ according to $\Delta'; E'_\Delta$ such that $t \downarrow_{[o'']} \preccurlyeq sa \downarrow_{[o']}$.*

4 Notion of Observation

Full abstraction is a comparison between two semantics, where the reference semantics to start from is traditionally *contextually* defined and based on a some notion of *observability.*

[5] For outgoing calls, the relevant environment clique is mentioned explicitly as the receiver of the call. Concerning returns, the concerned environment clique is determined by the matching call.

As starting point we choose, as [7], a (standard) notion of semantic equivalence or rather semantic implication —one program allows at least the observations of the other— based on a particular, simple form of contextual observation: being put into a context, the component, together with the context, is able to *reach* a defined point, which is counted as the successful observation. A context $\mathcal{C}[_]$ is a program "with a hole". In our setting, the hole is filled with a program fragment consisting of a *component* C in the syntactical sense, i.e., consisting of the parallel composition of (named) classes, named objects, and named threads, and the context is the rest of the programs such that $\mathcal{C}[C]$ gives a well-typed *closed* program $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$, where closed means that it can be typed in the empty contexts, i.e., $\vdash C' : ()$.

To report success, we assume an external class with a particular success-reporting method. So assume a class c_b of type $[\![succ : () \to none]\!]$, abbreviated as barb. A component C *strongly barbs on* c_b, written $C \downarrow_{c_b}$, if $C \equiv \nu(\boldsymbol{n}{:}\boldsymbol{T}, b{:}c_b).C' \parallel n\langle let\ x{:}none = b.succ()\ in\ t\rangle$, i.e., the call to the success-method of an instance of c_b is enabled. Furthermore, C *barbs on* c_b, written $C \Downarrow_{c_b}$, if it can reach a point which strongly barbs on c_b, i.e., $C \Longrightarrow C' \downarrow_{c_b}$. We can now define may testing preorder [6] as in [7].

Definition 2 (May testing). *Assume* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta$ *and* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$. *Then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$, *if* $(C_1 \parallel C) \Downarrow_{c_b}$ *implies* $(C_2 \parallel C) \Downarrow_{c_b}$ *for all* $\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C : \Delta; E_\Delta$.

5 Full Abstraction

The proof that may-testing coincides with order on traces given in Definition 1 has two directions: compared to $\sqsubseteq_{may}$, the relation $\sqsubseteq_{trace}$ is neither too abstract (soundness) nor too concrete (completeness).

For lack of space, we simply state the soundness result here. The proof is rather similar to the one for the object-based case [7] and rests on the ability to compose a component and an environment, performing complementary traces, into one global program (plus the dual property of decomposition). We refer to the full version [3] for details.

Proposition 1 (Soundness). *If* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, *then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$.

Completeness asserts the reverse direction:

Proposition 2 (Completeness). *If* $\Delta; E_\Delta \models C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$, *then* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$.

Concerning completeness, we sketch here one core aspect part of the argument. At the heart, completeness is a constructive argument: given a trace s, construct a component C_s that exhibits the trace s and not simply realize the trace, but realize it *exactly,* up-to unavoidable reordering and prefixing.

Table 4. Legal traces

$$\frac{}{\Delta; E_\Delta \vdash r \rhd \epsilon : \mathit{trace}\,\Theta; E_\Theta}\ \text{L-Empty}$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle \mathit{call}\ o_2.l(\boldsymbol{v})\rangle? \qquad \Delta \vdash o_1 : c_1 \qquad \Delta \vdash r \rhd a : \Theta \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; n \hookrightarrow o_2 \hookrightarrow \boldsymbol{v}) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \boldsymbol{v}, o_2 : \acute{\Theta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a_{o_1} \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E_\Theta} \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\,\Theta; E_\Theta}\ \text{L-CallI}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle \mathit{return}(v)\rangle! \qquad \mathit{pop}\ n\ r = \nu(\Delta'', \Theta'').\ n\langle [o_1]\mathit{call}\ o_2.l(\boldsymbol{v})\rangle? \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; n \hookrightarrow o_1 \hookrightarrow v \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow (\Theta', \Delta') \setminus n \\ \acute{\Theta}; \acute{E}_\Theta \vdash o_2 \leftrightharpoons\!\hookrightarrow v : \acute{\Delta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\,\Theta; E_\Theta}\ \text{L-RetO}$$

Legal Traces. To do so, we must first characterize which traces (the "legal" ones) can occur at all, and again the crucial difference to the object-based case is to take connectivity into account to exclude impossible combinations of transmitted object names and threads.

The legal traces are specified by a system for judgments of the form $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\,\Theta; E_\Theta$ stipulating that under the type and relational assumptions Δ and E_Δ and with the commitments Θ and E_Θ, the trace s is legal. Three exemplary rules for legal traces are shown in Table 4; not shown are two dual rules for outgoing calls and incoming returns, and furthermore two rules specifying the situation for the initial calls, which are similar to L-CallI. For simplicity, we omit premises dealing with static aspects of typing, as we did for the external semantics. As in the operational semantics, the caller identity, even if not part of the label, is guessed and remembered, here in the history r. The premise $\Delta \vdash r \rhd a : \Theta$ asserts that after r, the action a is enabled, and $\mathit{pop}\,n\ r$ picks the call matching the return in question. See [3] for details.

6 Conclusion

Inspired by the work of [7], we presented an operational semantics of a class-based, object-oriented calculus with multithreading. The seemingly innocent step from an *object-based* setting as in [7] to a framework with classes requires quite some extension in the operational semantics to characterize the possible behavior of a component. In particular it is necessary to keep track of the potential *connectivity* of objects of the environment to exclude impossible communication labels. It is therefore instructive, to review the differences in this conclusion, especially to try to understand how the calculus of [7] can be understood as a special case of the framework explored here.

The fundamental dichotomy underlying the observational definition of equivalence is the one between the inside and the outside: program or component vs. environment or observer, or in game-theoretical terms, player vs. opponent. This

leads to the crucial difference between object-based languages, instantiating from objects, and class-based language, instantiating from classes: In the class-based setting, instantiation may *cross the demarcation line between component and environment,* while in the object-based setting, this is not possible: the program only instantiates program objects, and the environment only objects belonging to the environment. All other complications follow from this difference, the most visible being that it is necessary to represent the dynamic object structure into the semantics, or rather an approximation of the connectivity of the environment objects. Another way to see it is, that in the setting of [7], there is only *one clique* in the environment, i.e., in the worst case, which is the relevant one, all environment objects are connected with each other. Since the component cannot create environment objects (or vice versa), new isolated cliques are never created. The object-based case can therefore be understood by invariantly (and trivially) taking $E_\Delta = \Delta \times (\Delta + \Theta)$, while in our setting, E_Δ may be more specific.

Further Related Work. [12] investigates the full abstraction problem in an object calculus with subtyping. The setting is a bit different from the one as used here as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. [4] presents a full abstraction result for the π-calculus, the standard process algebra for name passing and dynamically changing process structures. The extensional semantics is given as a domain-theoretic, categorical model, and using bisimulation equivalence as starting point, not may testing resp. traces as here. [13] gives equational full abstraction for standard translation of the polyadic π-calculus into the monadic one. Without additional information, the translation is not fully abstract, and [13] introduces graph-types as an extension to the π-calculus sorting to achieve full abstraction. The graph types abstract the *dynamic* behavior of processes. In capturing the dynamic behavior of interaction, Yoshida's graph types are rather different from the graph abstracting the connectivity of objects presented here. Recently, Jeffrey and Rathke [8] extended their work on trace-based semantics from an object-based setting to a core of *Java* (called *Java Jr.*), including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*.

Acknowledgements. We thank Andreas Grüner for careful reading, discussing, helping to clarify and improving a number of half-baken previous versions of the document. Likewise Karsten Stahl and Harald Fecher for "active listening" even to the more Byzantine details and dead ends of all this. We are also indepted to Ben Lukoschus for helping with some of the more arcane TEX-stunts and to Willem-Paul de Roever for spotting a number of sloppy points. Finally, we thank the reviewers for their insightful remarks.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Aug. 2003.
3. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. Preliminary technical report, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Jan. 2005.
4. M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus (extended abstract). In *Proceedings of LICS '96*, pages 43–54. IEEE, Computer Society Press, July 1996.
5. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
6. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
7. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.
8. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. 2005. Submitted for publication.
9. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
10. A. M. Pitts and D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In A. M. Borzyszkowski and S. Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Sept. 1993.
11. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
12. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.
13. N. Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.

Specifying Software Connectors

Marco Antonio Barbosa and Luís Soares Barbosa

Departamento de Informática – Universidade do Minho,
Campus de Gualtar 4710-057 – Braga – Portugal
{marco.antonio, lsb}@di.uminho.pt

Abstract. Orchestrating software components, often from independent suppliers, became a central concern in software construction. Actually, as relevant as components themselves, are the ways in which they can be put together to interact and cooperate in order to achieve some common goal. Such is the role of the so-called software connectors: external coordination devices which ensure the flow of data and enforce synchronization constraints within a component's network. This paper introduces a new model for software connectors, based on relations extended in time, which aims to provide support for light inter-component dependency and effective external control.

1 Introduction

The expression *software connector* was coined by software architects to represent the interaction patterns among components, the latter regarded as basic computational elements or information repositories. Their aim is to mediate the communication and coordination activities among components, acting as a sort of glueing code between them. Examples range from simple channels or pipes, to event broadcasters, synchronisation barriers or even more complex structures encoding client-server protocols or hubs between databases and applications.

Although component-based development [19, 25, 15] became accepted in industry as a new effective paradigm for Software Engineering and even considered its cornerstone for the years to come, there is still a need for precise ways to document and reason about the high-level structuring decisions which define a system's software architecture.

Conceptually, there are essentially two ways of regarding *component-based* software development. The most wide-spread, which underlies popular technologies like, *e.g.*, CORBA [24], DCOM [14] or JAVABEANS [16], reflects what could be called the *object orientation* legacy. A component, in this sense, is essentially a collection of objects and, therefore, component interaction is achieved by mechanisms implementing the usual *method call* semantics. As F. Arbab stresses in [3] this

> induces an asymmetric, unidirectional semantic dependency of users (of services) on providers (...) which subverts independence of components, contributes to the breaking of their encapsulation, and leads to a level of inter-dependence among components that is no looser than that among objects within a component.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 52–67, 2005.
© Springer-Verlag Berlin Heidelberg 2005

An alternative point of view is inspired by research on coordination languages [13, 21] and favors strict component decoupling in order to support a looser inter-component dependency. Here computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controlled. This model is (partially) implemented in JavaSpaces on top of Jini [20] and fundamental to a number of approaches to component-ware which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, Reo [3] or Piccola [23, 18].

Adopting the latter point of view, this paper focuses on the specification of software connectors either as *relations* over a temporarily labelled data domain (representing the flow of messages) or as *relations extended in time*, *i.e.*, defined with respect to a a memory of past computations encoded as an internal state space. The latter model extends the former just as a labelled transition system extends a simple relation. Formally, we resort to coalgebraic structures [22] to model such *extended relations*, pursuing a previous line of research on applying coalgebra theory to the semantics of component-based software development (see, eg, [5, 6, 17].).

The paper is organized as follows: a semantic model for software connectors is introduced in section 2 and illustrated with the specification of one of the most basic connectors: the *asynchronous channel*. The model is further developed in section 3 which introduces a systematic way of building connectors by *aggregation* of *ports* as well as two combinators encoding, respectively, a form of *concurrent* composition and a generalization of *pipelining*. Section 4 illustrates the expressiveness of this model through the discussion of some typical examples from the literature. Finally, section 5 summarizes what has been achieved and enumerates a few research questions for the future.

Notation. The paper resorts to standard mathematical notation emphasizing a *pointfree* specification style (as in, *e.g.*, [9]) which leads to more concise descriptions and increased calculation power. The underlying mathematical universe is the category of sets and set-theoretic functions whose composition and identity are denoted by $\cdot$ and id, respectively. Notation $(\phi \rightarrow f, g)$ stands for a conditional statement: if ϕ then apply function f else g. As usual, the basic set constructs are *product* $(A \times B)$, *sum*, or disjoint union, $(A + B)$ and *function space* (B^A). We denote by $\pi_1 : A \times B \longrightarrow A$ the first projection of a product and by $\iota_1 : A \longrightarrow A + B$ the first embedding in a sum (similarly for the others). Both $\times$ and $+$ extend to functions in the usual way and, being universal constructions, a canonical arrow is defined to $A \times B$ from any set C and, symmetrically, from $A + B$ to any set C, given functions $f : C \longrightarrow A, g : C \longrightarrow B$ and $l : A \longrightarrow C, h : B \longrightarrow C$, respectively. The former is called a *split* and denoted by $\langle f, g \rangle$, the latter an *either* and denoted by $[l, h]$, satisfying

$$k = \langle f, g \rangle \;\Leftrightarrow\; \pi_1 \cdot k = f \;\wedge\; \pi_2 \cdot k = g \tag{1}$$

$$k = [l, h] \;\Leftrightarrow\; k \cdot \iota_1 = l \;\wedge\; k \cdot \iota_2 = h \tag{2}$$

Notation B^A is used to denote *function space*, *i.e.*, the set of (total) functions from A to B. It is also characterized by an universal property: for all function

$f : A \times C \longrightarrow B$, there exists a unique $\overline{f} : A \longrightarrow B^C$, called the *curry* of f, such that $f = \mathsf{ev} \cdot (\overline{f} \times C)$. Finally, we also assume the existence of a few basic sets, namely $\emptyset$, the empty set and $\mathbf{1}$, the singleton set. Note they are both 'degenerate' cases of, respectively, *sum* and *product* (obtained by applying the iterated version of those combinators to a nullary argument). Given a value v of type X, the corresponding constant function is denoted by $\underline{v} : \mathbf{1} \longrightarrow x$. Of course all set constructions are made up to isomorphism. Therefore, set $\mathbb{B} = \mathbf{1} + \mathbf{1}$ is taken as the set of boolean values true and false. Finite sequences of X are denoted by X^*. Sequences are observed, as usual, by the head (head) and tail (tail) functions, and built by singleton sequence construction (singl) and concatenation ($\frown$).

2 Connectors as Coalgebras

2.1 Connectors

According to Allen and Garlan [1] an expressive notation for software connectors should have three properties. Firstly, it should allow the specification of common patterns of architectural interaction, such as remote call, pipes, event broadcasters, and shared variables. Secondly, it should scale up to the description of complex, eventually dynamic, interactions among components. For example, in describing a client–server connection we might want to say that the server must be initialized by the client before a service request becomes enabled. Finally, it should allow for fine-grained distinctions between small variations of typical interaction patterns.

In this paper a connector is regarded as a *glueing device* between software components, ensuring the flow of data and synchronization constraints. Software components interact through anonymous messages flowing through a connector network. The basic intuition, borrowed from the coordination paradigm, is that connectors and components are independent devices, which make the latter amenable to external coordination by the former.

Connectors have *interface points*, or *ports*, through which messages flow. Each port has an *interaction polarity* (either *input* or *output*), but, in general, connectors are blind with respect to the data values flowing through them. Consequently, let us assume $\mathbb{D}$ as the generic type of such values. The simplest connector one can think of — the *synchronous channel* — can be modelled just as a *function* $[\![\bullet \longmapsto \bullet]\!] : \mathbb{D} \longrightarrow \mathbb{D}$. The corresponding temporal constraint — that input and output occur simultaneously — is built-in in the very notion of a function. Such is not the case, however, of an *asynchronous channel* whose synchronization constraints entails the need for the introduction of some sort of temporal information in the model. Therefore, we assume that, on crossing the borders of a connector, every data value becomes labelled by a *time stamp* which represents a (rather weak) notion of time intended to express *order of occurrence*. As in [3], temporal *simultaneity* is simply understood as *atomicity*, in the sense that two equally tagged input or output events are supposed to occur in an atomic way, that is, without being interleaved by other events.

In such a setting, the semantics of a connector C, with m input and n output ports, is given by a relation

$$[\![C]\!] : (\mathbb{D} \times \mathbb{T})^n \longrightarrow (\mathbb{D} \times \mathbb{T})^m \tag{3}$$

The *asynchronous channel*, in particular, is specified by

$$[\![\bullet \longmapsto \bullet]\!] \subseteq (\mathbb{D} \times \mathbb{T}) \times (\mathbb{D} \times \mathbb{T}) \;=\; \{((d,t),(d',t')) \mid d' = d \;\wedge\; t' > t\}$$

This simple model was proposed by the authors in [7], where its expressive power and reasoning potential is discussed. Note that with the explicit representation of a temporal dimension one is able to model non trivial synchronization restrictions. Relations, on the other hand, cater for non deterministic behaviour. For example, a *lossy* channel, *i.e.*, one that can loose information, therefore modeling unreliable communication, is specified by a correflexive relation over $\mathbb{D} \times \mathbb{T}$, *i.e.*, a subset of the identity $\mathsf{Id}_{\mathbb{D}\times\mathbb{T}}$.

On the other hand it seems difficult to express in this model the FIFO requirement usually associated to an asynchronous channel. The usual way to express such constraints, requiring a fine-grain control over the flow of data, resorts to *infinite* data structures, typically *streams*, *i.e.*, infinite sequences, of messages (as in [4, 3] or [8]). An alternative, more operational, approach, to be followed in the sequel, is the introduction of some form of internal memory in the specification of connectors. Let U be its type, which, in the asynchronous channel example, is defined as a sequence of $\mathbb{D}$ values, *i.e.*, $U = \mathbb{D}^*$, representing explicitly the buffering of incoming messages. The asynchronous channel is, then, given by the specification of two ports to which two operations over $\mathbb{D}^*$, corresponding to the reception and delivery of a $\mathbb{D}$ value, are associated. The rationale is that the operations are activated by the arrival of a data element (often referred to as a message) to the port. Formally,

$$\begin{aligned} \mathsf{receive} \;&:\; \mathbb{D}^* \times D \;\rightarrow\; \mathbb{D}^* \\ &= \frown \cdot (\mathsf{id} \times \mathsf{singl}) \\ \mathsf{deliver} \;&:\; \mathbb{D}^* \;\rightarrow\; \mathbb{D}^* \times (\mathbb{D} + \mathbf{1}) \\ &= \langle \mathsf{tl}, \mathsf{hd} \rangle \end{aligned}$$

Grouping them together leads to a specification of the channel as an elementary transition structure over $\mathbb{D}^*$, *i.e.*, a pointed *coalgebra* $\langle [] \in \mathbb{D}^*, c : \mathbb{D}^* \longrightarrow (\mathbb{D}^* \times (\mathbb{D} + \mathbf{1}))^{(\mathbb{D}+\mathbf{1})} \rangle$ where

$$\begin{aligned} \overline{c} \;=\; \mathbb{D}^* \times (\mathbb{D}+1) \;&\xrightarrow{\;\mathsf{dr}\;}\; \mathbb{D}^* \times \mathbb{D} + \mathbb{D}^* \;\xrightarrow{\;\mathsf{receive}+\mathsf{deliver}\;}\; \mathbb{D}^* + \mathbb{D}^* \times (\mathbb{D}+1) \\ &\xrightarrow{\;\simeq\;}\; \mathbb{D}^* \times 1 + \mathbb{D}^* \times (\mathbb{D}+1) \;\xrightarrow{\;[\mathsf{id}\times\iota_2, \mathsf{id}]\;}\; \mathbb{D}^* \times (\mathbb{D}+1) \end{aligned}$$

Note how this specification meets all the exogenous synchronization constraints, including the enforcing of a strict FIFO discipline. The temporal dimension, however, is no more explicit, but *built-in* in coalgebra dynamics. We shall come back to this in section 5. For the moment, however, let us elaborate on this example to introduce a general model of software connectors as coalgebras.

2.2 The General Model

A software connector is specified by an interface which aggregates a number of *ports* represented by operations which regulate its behaviour. Each operation encodes the port reaction to a data item crossing the connector's boundary. Let U be the type of the connector's internal state space and $\mathbb{D}$ a generic data domain for messages, as before. In such a setting we single out three kinds of ports with the following signatures:

$$\mathsf{post} \ : \ U \longrightarrow U^{\mathbb{D}} \tag{4}$$

$$\mathsf{read} \ : \ U \longrightarrow (\mathbb{D} + 1) \tag{5}$$

$$\mathsf{get} \ : \ U \longrightarrow U \times (\mathbb{D} + 1) \tag{6}$$

where

- post is an input operation analogous to a write operation in conventional programming languages (see *e.g.*, [2, 21, 3]). Typically, a post port accepts data items and store them internally, in some form.
- read is a non-destructive output operation. This means that through a read port the environment might 'observe' a data item, but the connector's state space remains unchanged. Of course read is a partial operation, because there cannot be any guarantee that data is available for reading.
- get is a destructive variation of the read port. In this case the data item is not only made externally available, but also deleted from the connector's memory.

As mentioned above, connectors are formed by the aggregation of a number of post, read and get ports. According to their number and types one specific connectors with well-defined behaviours may be defined. Let us consider some possibilities.

Sources and Sinks. The most elementary connectors are those with a unique port. According to its orientation they can be classified as

- Data *sources*, specified by a single read operation

$$\Diamond_d \ = \ \langle d \in \mathbb{D}, \iota_1 : \mathbb{D} \to \mathbb{D} + \mathbf{1} \rangle \tag{7}$$

  defined over state space $U = \mathbb{D}$ and initialized with value d.
- Data *sinks*, ie, connectors which are always willing to accept any data item, discarding it immediately. The state space of data sinks is irrelevant and, therefore, modeled by the singleton set $\mathbf{1} = \{*\}$. Formally,

$$\blacklozenge \ = \ \langle * \in \mathbf{1}, ! : \mathbf{1} \to \mathbf{1}^{\mathbb{D}} \rangle \tag{8}$$

  where ! is the (universal) map from any object to the (final) set $\mathbf{1}$.

Binary Connectors. Binary connectors are built by the aggregation of two ports, assuming the corresponding operations are defined over the same state space. This, in particular, enforces mutual execution of state updates.

– Consider, first, the aggregation of two read ports, denoted by read_1 and read_2, with possibly different specifications. Both of them are (non destructive) observers and, therefore, can be simultaneously offered to the environment. The result is a coalgebra simply formed by their *split*:

$$c \;=\; \langle u \in U, \langle \mathsf{read}_1, \mathsf{read}_2 \rangle : U \to (\mathbb{D}+1) \times (\mathbb{D}+1) \rangle \tag{9}$$

– On the other hand, aggregating a post to a read port results in

$$c \;=\; \langle u \in U, \langle \mathsf{post}, \mathsf{read} \rangle : U \to U^{\mathbb{D}} \times (\mathbb{D}+1) \rangle \tag{10}$$

– Replacing the read port above by a get one requires an additive aggregation to avoid the possibility of simultaneous updates leading to

$$c \;=\; \langle u \in U, \gamma_c : U \to (U \times (\mathbb{D}+1))^{\mathbb{D}+\mathbf{1}} \rangle \tag{11}$$

where[1]

$$\begin{aligned} \overline{\gamma_c} \;=\; U \times (\mathbb{D}+1) &\xrightarrow{\;\mathsf{dr}\;} U \times \mathbb{D} + U \xrightarrow{\;\overline{\mathsf{post}}+\mathsf{get}\;} U + U \times (\mathbb{D}+1) \\ &\xrightarrow{\;\simeq\;} U \times 1 + U \times (\mathbb{D}+1) \xrightarrow{\;[\mathsf{id}\times\iota_2, \mathsf{id}]\;} U \times (\mathbb{D}+1) \end{aligned}$$

Channels of different kinds are connectors of this type. Recall the asynchronous channel example above: ports identified by receive and deliver have the same signature of a post and a get, respectively. An useful variant is the *filter* connector which discards some messages according to a given predicate $\phi : \mathbf{2} \longleftarrow \mathbb{D}$. The get port is given as before, *i.e.*, $\langle \mathsf{tl}, \mathsf{hd} \rangle$, but post becomes conditional on predicate ϕ, *i.e.*,

$$\mathsf{post} \;=\; \phi \to \frown \cdot (\mathsf{id} \times \mathsf{singl}), \mathsf{id}$$

– A similar scheme is adopted for the combination of two post ports:

$$c \;=\; \langle u \in U, \gamma_c : U \to U^{\mathbb{D}+\mathbb{D}} \rangle \tag{12}$$

where

$$\begin{aligned} \overline{\gamma_c} \;= U \times (\mathbb{D}+\mathbb{D}) &\xrightarrow{\;\mathsf{dr}\;} U \times \mathbb{D} + U \times \mathbb{D} \\ &\xrightarrow{\;\overline{\mathsf{post}_1}+\overline{\mathsf{post}_2}\;} U + U \xrightarrow{\;\nabla\;} U \end{aligned}$$

[1] In the sequel dr is the right distributivity isomorphism and ∇ the codiagonal function defined as the *either* of two identities, *i.e.*, $\nabla = [\mathsf{id}, \mathsf{id}]$.

The General Case. Examples above lead to the specification of the following shape for a connector built by aggregation of P post, G get and R read ports:

$$c \;=\; \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{D}+1))^{P \times \mathbb{D} + G} \times (\mathbb{D}+1)^R \rangle \tag{13}$$

where ρ_c is the split the R read ports, *i.e.*,

$$\rho_c : \; U \longrightarrow (\mathbb{D}+1) \times (\mathbb{D}+1) \times \ldots \times (\mathbb{D}+1) \tag{14}$$

and, γ_c collects ports of type post or get, which are characterized by the need to perform state updates, in the uniform scheme explained above for the binary case. Note that this expression can be rewritten as

$$U \;=\; (\sum_{i \in P} U^{\mathbb{D}} + \sum_{j \in G} U \times (\mathbb{D}+1)) \times \prod_{k \in R} (\mathbb{D}+1) \tag{15}$$

which is, however, less amenable to symbolic manipulation in proofs.

3 Combinators

In the previous section, a general model of software connectors as pointed coalgebras was introduced and their construction by port aggregation discussed. To obtain descriptions of more complex interaction patterns, however, some forms of connector composition are needed. Such is the topic of the present section in which two combinators are defined: a form of *concurrent composition* and a generalisation of *pipelining* capturing arbitrary composition of post with either read or get ports.

3.1 Concurrent Composition

Consider connectors c_1 and c_2 defined as

$$c_i = \langle u_i \in U_i, \langle \gamma_i, \rho_i \rangle : (U_i \times (\mathbb{D}+1))^{P_i \times \mathbb{D} + G_i} \times (\mathbb{D}+1)^{R_i} \rangle$$

with P_i ports of type post, R_i of type read and G_i of type get, for $i = 1, 2$. Their concurrent composition, denoted by $c_1 \boxtimes c_2$ makes externally available all c_1 and c_2 *single* primitive ports, plus *composite* ports corresponding to the simultaneous activation of post (respectively, get) ports in the two operands. Therefore, $P' = P_1 + P_2 + P_1 \times P_2$, $G' = G_1 + G_2 + G_1 \times G_2$ and $R' = R_1 + R_2$ become available in $c_1 \boxtimes c_2$ as its interface sets. Formally, define

$$c_1 \boxtimes c_2 : U' \longrightarrow (U' \times (\mathbb{D}+1))^{P' \times \mathbb{D} + G'} \times (\mathbb{D}+1)^{R'} \tag{16}$$

where

$$\begin{aligned}
\overline{\gamma}_{c_1 \boxtimes c_2} \;=\; & U_1 \times U_2 \times (P_1 + P_2 + P_1 \times P_2) \times \mathbb{D} + (G_1 + G_2 + G_1 \times G_2) \xrightarrow{\simeq} \\
& (U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 + U_1 \times U_2 \times (P_2 \times \mathbb{D} + G_2) + U_1 \times (P_1 \times \mathbb{D} + G_1) \times U_2 \times (P_2 \times \mathbb{D} + G_2) \\
& \xrightarrow{\gamma_1 \times \mathsf{id} + \mathsf{id} \times \gamma_2 + \gamma_1 \times \gamma_2} (U_1 \times (\mathbb{D}+1)) \times U_2 + U_1 \times (U_2 \times (\mathbb{D}+1)) + (U_1 \times (\mathbb{D}+1)) \times (U_2 \times (\mathbb{D}+1)) \\
& \xrightarrow{\simeq} U_1 \times U_2 \times (\mathbb{D}+1) + U_1 \times U_2 \times (\mathbb{D}+1) + U_1 \times U_2 \times (\mathbb{D}+1)^2 \xrightarrow{\nabla + \mathsf{id}} \\
& U_1 \times U_2 \times (\mathbb{D}+1) + U_1 \times U_2 \times (\mathbb{D}+1) \times U_2 (\mathbb{D}+1) \xrightarrow{\simeq} U_1 \times U_2 \times ((\mathbb{D}+1) + (\mathbb{D}+1))^2
\end{aligned}$$

and

$$\rho_{c_1 \boxtimes c_2} = U_1 \times U_2 \xrightarrow{\rho_1 \times \rho_2} (\mathbb{D}+1)^{R_1} \times (\mathbb{D}+1)^{R_2} \xrightarrow{\simeq} (\mathbb{D}+1)^{R_1+R_2}$$

3.2 Hook

The *hook* combinator plugs ports with opposite polarity within an arbitrary connector

$$c = \langle u \in U, \langle \gamma_c, \rho_c \rangle : U \longrightarrow (U \times (\mathbb{D}+1))^{P \times \mathbb{D}+G} \times (\mathbb{D}+1)^R \rangle$$

There are two possible plugging situations:

1. Plugging a post port p_i to a read r_j one, resulting in

$$\begin{aligned}
\rho_{c \curvearrowleft_{r_j}^{p_i}} &= \langle r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_R \rangle \\
\overline{\gamma}_{c \curvearrowleft_{r_j}^{p_i}} &= U \times ((P-1) \times \mathbb{D} + G) \xrightarrow{\theta \times \mathsf{id}} U \times ((P-1) \times \mathbb{D} + G) \\
&\xrightarrow{\simeq} \textstyle\sum_{P-1} U \times \mathbb{D} + \sum_G U \xrightarrow{[p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_p]+[g_1,\ldots,g_G]} \\
&U + U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \\
&\xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} U \times (\mathbb{D}+1)
\end{aligned}$$

   where $\theta : U \to U$

$$\begin{aligned}
\theta = U &\xrightarrow{\triangle} U \times U \xrightarrow{\mathsf{id} \times r_j} U \times \mathbb{D} + 1 \\
&\xrightarrow{\simeq} U \times \mathbb{D} + U \xrightarrow{\overline{p_i} + \mathsf{id}} U + U \xrightarrow{\triangledown} U
\end{aligned}$$

2. Plugging a post port p_i to a get g_j one, resulting in

$$\begin{aligned}
\rho_{c \curvearrowleft_{r_j}^{p_i}} &= \rho_c \\
\overline{\gamma}_{c \curvearrowleft_{g_j}^{p_i}} &= U \times ((P-1) \times \mathbb{D} + (G-1)) \xrightarrow{\theta \times \mathsf{id}} \\
&U \times ((P-1) \times \mathbb{D} + (G-1)) \\
&\xrightarrow{\simeq} \textstyle\sum_{P-1} U \times \mathbb{D} + \sum_{G-1} U \\
&\xrightarrow{[p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_p]+[g_1,\ldots,g_{j-1},g_{j+1},\ldots,g_G]} \\
&U + U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times 1 + U \times (\mathbb{D}+1) \\
&\xrightarrow{[\mathsf{id} \times \iota_2, \mathsf{id}]} U \times (\mathbb{D}+1)
\end{aligned}$$

   where $\theta : U \to U$

$$\begin{aligned}
\theta = U &\xrightarrow{g_j} U \times (\mathbb{D}+1) \xrightarrow{\simeq} U \times \mathbb{D} + U \\
&\xrightarrow{\overline{p_i} + \mathsf{id}} U + U \xrightarrow{\triangledown} U
\end{aligned}$$

Note that, according to the definition above, if the result of a reaction at a port of type read or get is of type **1**, which encodes the absence of any data item to be read, the associated post is not activated and, consequently, the interaction does not become effective.

Such unsuccessful read attempt can alternatively be understood as a *pending read request*. In this case the intended semantics for interaction with the associated post port is as follows: successive read attempts are performed until communication occurs. This version of *hook* is denoted by $\Gamma_r^p\ c$ and easily obtained by replacing, in the definition of θ above, step

$$U \times \mathbb{D} + U \xrightarrow{\overline{p_i}+\mathsf{id}} U + U$$

by

$$U \times \mathbb{D} + U \xrightarrow{\overline{p_i}+\theta} U + U$$

Both forms of the *hook* combinator can be applied to a whole sequence of pairs of opposite polarity ports, the definitions above extending as expected.

The two combinators introduced in this section can also be put together to define a form of *sequential composition* in situations where all the post ports of the second operand (grouped in *in*) are connected to all the read and get ports of the first (grouped in *out*). It is assumed that hooks between two single ports extend smoothly to any product of ports (as arising from concurrent composition) in which they participate. Formally, we define by abbreviation

$$c_1 \,;\, c_2 \stackrel{\text{abv}}{=} (c_1 \boxtimes c_2)\ \curvearrowleft_{out}^{in} \tag{17}$$

and

$$c_1 \bowtie c_2 \stackrel{\text{abv}}{=} \Gamma_{out}^{in}\ (c_1 \boxtimes c_2) \tag{18}$$

4 Examples

This section discusses how some typical software connectors can be defined in the model proposed in this paper.

4.1 Broadcasters and Mergers

Our first example is the *broadcaster*, a connector which replicates in each of its two (output) ports, any input received in its (unique) entry as depicted bellow. There are two variants of this connector denoted, respectively, by $\blacktriangleleft$ and $\triangleleft$. The first one corresponds to a *synchronous* broadcast, in the sense that the two get ports are activated simultaneously. The other one is *asynchronous*, in the sense that both of its get ports can be activated independently. The definition of $\triangleleft$ is rather straightforward as a coalgebra over $U = \mathbb{D} + \mathbf{1}$ and operations

$$\begin{aligned} \overline{\mathsf{post}} &: U \times \mathbb{D} \to U = \iota_1 \cdot \pi_2 \\ \mathsf{get}_1, \mathsf{get}_2 &: U \to U \times (\mathbb{D} + 1) = \triangle \end{aligned}$$

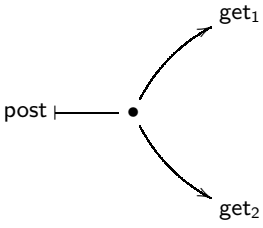


Fig. 1. The *broadcaster* connector

where $\triangle$ is the *diagonal* function, defined by $\triangle = \langle \mathsf{id}, \mathsf{id} \rangle$. The synchronous case, however, requires the introduction of two boolean flags initialized to $\langle \mathsf{false}, \mathsf{false} \rangle$ to witness the presence of get requests at both ports. The idea is that a value is made present at both the get ports if it has been previously received, as before, and there exists two reading requests pending. Formally, let $U = (\mathbb{D}+\mathbf{1}) \times (\mathbb{B} \times \mathbb{B})$ and define

$$\overline{\mathsf{post}} \ : \ U \times \mathbb{D} \ \rightarrow \ U \ = \ \langle \iota_1 \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle$$
$$\mathsf{get}_1 \ : \ U \ \rightarrow \ U \times (\mathbb{D}+1) \ = \ (=_* \cdot \pi_1 \ \rightarrow \ \langle \mathsf{id}, \pi_1 \rangle, \ \mathsf{getaux}_1)$$

where

$$\mathsf{getaux}_1 \ = \ (\pi_2 \cdot \pi_2 \ \rightarrow \ \langle (\iota_2 \cdot \underline{*}) \times (\underline{\mathsf{false}} \times \underline{\mathsf{false}}), \pi_1 \rangle, \ \langle \mathsf{id} \times (\underline{\mathsf{true}} \times \mathsf{id}), \iota_2 \cdot \underline{*} \rangle)$$

I.e., if there is no information stored flag $*$ is returned and the state left unchanged. Otherwise, an output is performed but only if there is a previous request at the other port. If this is not the case the reading request is recorded at the connector's state. This definition precludes the possibility of a reading unless there are reading requests at both ports. The fact that both requests are served depends on their interaction with the associated post ports, *i.e.*, on the chosen hook discipline (see the synchronization barrier example in subsection 4.3). The definition of get_2 is similar but for the boolean flags update:

$$\mathsf{getaux}_2 \ = \ (\pi_1 \cdot \pi_2 \ \rightarrow \ \langle (\iota_2 \cdot \underline{*}) \times (\underline{\mathsf{false}} \times \underline{\mathsf{false}}), \pi_1 \rangle, \ \langle \mathsf{id} \times (\mathsf{id} \times \underline{\mathsf{true}}), \iota_2 \cdot \underline{*} \rangle)$$

Dual to the *broadcaster* connector is the *merger* which concentrates messages arriving at any of its two post ports. The *merger*, denoted by $\triangleright$, is similar to

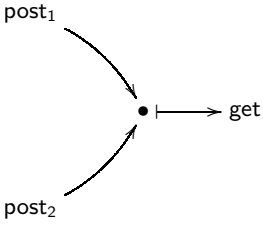


Fig. 2. The *merger* connector

an asynchronous channel, as given in section 2, with two identical post ports. Another variant, denoted by ▶, accepts one data item a time, after which disables both post ports until get is activated. This connector is defined as a coalgebra over $U = \mathbb{D} + \mathbf{1}$ with

$$\begin{aligned} \overline{\mathsf{post}_1} = \overline{\mathsf{post}_2} \;:\; & U \times \mathbb{D} \;\rightarrow\; U \\ = \;& (=_* \cdot \pi_1 \;\rightarrow\; \pi_1,\; \iota_1 \cdot \pi_2) \\ \mathsf{get} \;:\; & U \;\rightarrow\; U \times (\mathbb{D} + \mathbf{1}) \\ = \;& (=_* \rightarrow\; \langle \triangle, \mathsf{id} \rangle,\; \langle \iota_2 \cdot \underline{*}, \mathsf{id} \rangle) \end{aligned}$$

4.2 Drains

A *drain* is a symmetric connector with two inputs, but no output, points. Operationally, every message arriving to an end–point of a drain is simply lost. A drain is *synchronous* when both post operations are required to be active at the same time, and *asynchronous* otherwise. In both case, no information is saved and, therefore $U = \mathbf{1}$. Actually, drains are used to enforce synchronisation in the flow of data. Formally, an *asynchronous* drain is given by coalgebra

$$[\![\; \bullet \overset{\triangledown}{\longmapsto\!\dashv} \bullet \;]\!] \;:\; \mathbf{1} \longrightarrow \mathbf{1}^{\mathbb{D}+\mathbb{D}}$$

where both post ports are modelled by the (universal) function to $\mathbf{1}$, *i.e.*, $\mathsf{post}_1 = !_{U \times \mathbb{D}} = \mathsf{post}_2$. The same operations can be composed in a product to model the *synchronous* variant:

$$[\![\; \bullet \overset{\blacktriangledown}{\longmapsto\!\dashv} \bullet \;]\!] \;:\; U \longrightarrow U^{\mathbb{D} \times \mathbb{D}}$$

defined by

$$\mathbf{1} \times (\mathbb{D} \times \mathbb{D}) \xrightarrow{\;\cong\;} \mathbf{1} \times \mathbb{D} \times \mathbf{1} \times \mathbb{D} \xrightarrow{\overline{\mathrm{post}}_1 \times \overline{\mathrm{post}}_2} \mathbf{1} \times \mathbf{1} \xrightarrow{\;!\;} \mathbf{1}$$

There is an important point to make here. Note that in this definition two post ports were aggregated by a product, instead of resorting to the more common additive context. Such is required to enforce their simultaneous activation and, therefore, to meet the expected synchrony constraint. This type of port aggregation also appears as a result of concurrent composition. In general, when presenting a connector's interface, we shall draw a distinction between *single* and *composite* ports, the latter corresponding to the simultaneous activation of two or more of the former.

Composite ports, on the other hand, entail the need for a slight generalisation of hook. In particular it should cater for the possibility of a post port requiring, say, two values of type $\mathbb{D}$ be plugged to two (different) read or get ports. Such a generalisation is straightforward and omitted here (but used below on examples involving *drains*).

4.3 Synchronization Barrier

In the coordination literature a *synchronization barrier* is a connector used to enforce mutual synchronization between two channels (as σ_1 and σ_2 below). This is achieved by the composition of two synchronous broadcasters with two of their get ports connected by a synchronous drain. As expected, data items read at extremities o_1 and o_2 are read simultaneously. The composition pattern is depicted in figure 3, which corresponds to the following expression:

$$(\blacktriangleleft \boxtimes \blacktriangleleft) \bowtie ((\bullet \overset{\sigma_1}{\longmapsto} \bullet) \boxtimes (\bullet \longmapsto \bullet) \boxtimes (\bullet \overset{\sigma_2}{\longmapsto} \bullet)) \tag{19}$$

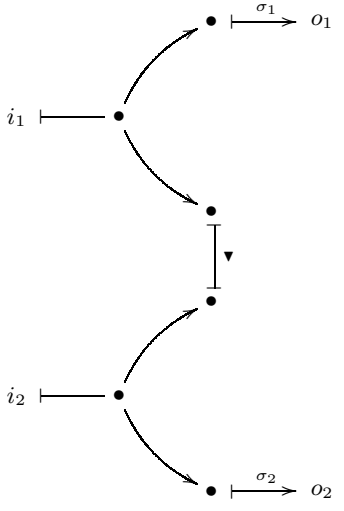


Fig. 3. A *synchronization barrier*

4.4 The Dining Philosophers

Originally posed and solved by Dijkstra in 1965, the *dinning philosophers* problem provides a good example to experiment an exogenous coordination model of the kind proposed in this paper[2]. In the sequel we discuss two possible solutions to this problem.

A* Merger-Drain *Solution. One possible solution assumes the existence of five replicas of a component *Phi*(losopher), each one with four get ports, two on the lefthand side and another two on the righthand side. The port labeled left_i

[2] The basic version reads as follows. Five philosophers are seated around a table. Each philosopher has a plate of spaghetti and needs two forks to eat it. When a philosopher gets hungry, he tries to acquire his left and right fork, one at a time, in either order. If successful in acquiring two forks, he eats for a while, then puts down the forks and continues to think.

is activated by Phi_i to place a request for the left fork. On the other hand, port leftf_i is activated on its release (and similarly for the ports on the right). Coordination between them is achieved by a software connector Fork with four post ports, to be detailed below. The connection between two adjacent philosophers through a Fork is depicted below which corresponds to the following expression in the calculus

$$(Phi_i \boxtimes \mathsf{Fork}_i \boxtimes Phi_{i+1}) \; \leftarrow^{\mathsf{rr}_i\, \mathsf{rf}_i\, \mathsf{lr}_i\, \mathsf{lf}_i}_{\mathsf{right}_i\, \mathsf{rightf}_i\, \mathsf{left}_{i+1}\, \mathsf{leftf}_{i+1}} \tag{20}$$

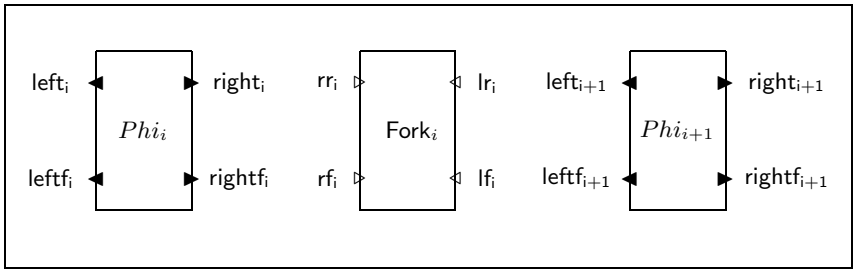


Fig. 4. Dining Philosophers (1)

The synchronization constraints of the problem are dealt by connector Fork built from two blocking mergers and a synchronous drain depicted in figure 5 and given by expression

$$(\blacktriangleright \boxplus \blacktriangleright)\,;\; \bullet \vdash\!\!\blacktriangledown\!\!\dashv \bullet \tag{21}$$

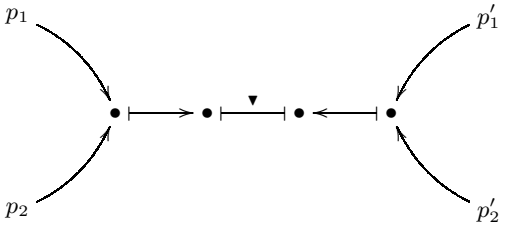


Fig. 5. A *Fork* connector (1)

A* Token *Solution. Another solution is based on a specification of Fork as an *exchange token* connector. Such a connector is given as a coalgebra over $U = \{\Cap\} + \mathbf{1}$, where $\Cap$ is a token representing the (physical) fork. From the point of view of a philosopher requesting a fork equivales to an attempt to remove $\Cap$ from the connector state space. Dually, a fork is released by returning it to the connector state space. In detail, a fork request at a philosopher port, say right, which is a post port hooked to (the get port) rr of the connector is only

successful if the token is available. Otherwise the philosopher must wait until a fork is released. The port specifications for Fork are as follows

$$\begin{aligned}\overline{\mathsf{rr}} = \overline{\mathsf{lr}} \;:\; & U \;\rightarrow\; U \times (\mathbb{D}+1)\\ & =\; (=_{\underline{\Uparrow}} \rightarrow\; (\iota_2 \cdot \underline{*}) \times (\iota_1 \cdot \underline{\Uparrow}),\; \mathsf{id} \times (\iota_2 \cdot \underline{*}))\\ \overline{\mathsf{rf}} = \overline{\mathsf{lf}} \;:\; & U \times \mathbb{D} \;\rightarrow\; U\\ & =\; \iota_1 \cdot \underline{\Uparrow}\end{aligned}$$

Again, the *Fork* connector is used as a mediating agent between any two philosophers as depict in figure 6. The corresponding expression is

$$(Phi_i \boxtimes \mathsf{Fork}_i \boxtimes Phi_{i+1}) \;\curlywedge^{\mathsf{right}_i\ \mathsf{rf}_i\ \mathsf{left}_i\ \mathsf{lf}_i}_{\mathsf{rr}_i\ \mathsf{rightf}_i\ \mathsf{lr}_{i+1}\ \mathsf{leftf}_{i+1}} \tag{22}$$

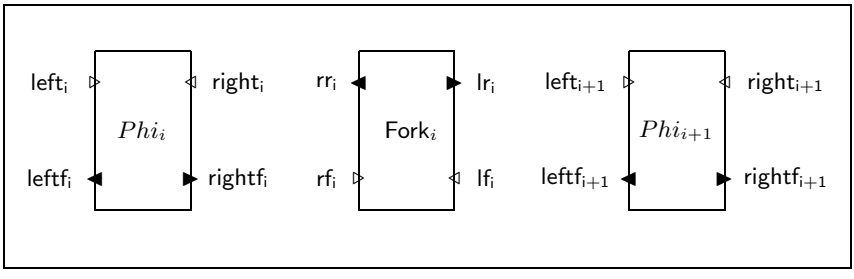


Fig. 6. Dining Philosophers (2)

5 Conclusions and Future Work

This paper discussed the formalization of software connectors, adopting a coordination oriented approach to support looser levels of inter-component dependency. Two alternative models were mentioned: relations on time-tagged domains (detailed in [7]) and (polynomial) coalgebras, regarded as relations extended in time, which is the basic issue of this paper. The close relation between the two models is still object of on-going work. In particular, how does the relational model lifts to a coalgebraic one when more complex notions of time are adopted? Note that, in most cases, the usual set-theoretic universe underlying coalgebras as used here lacks enough structure to extend such relations over (richly structured) timed universes.

Resorting to coalgebras to specify software connectors has the main advantage of being a smooth extension of the previous relational model. Actually, any relation can be seen as a coalgebra over the singleton set, *i.e.*, $U = \mathbf{1}$. Moreover, techniques of coalgebraic analysis, namely *bisimulation*, can be used to reason about connectors and, in general, architectural design descriptions. In fact, although in this paper the emphasis was placed on connector modeling and expressiveness, the model supports a basic calculus in which connector equivalence

and refinement can be discussed (along the lines of [17]). The model compares quite well to the more classic stream-based approaches (see *e.g.*, [10, 8, 3]), which can be recovered as the *final* interpretation of the coalgebraic specifications proposed here.

A lot of work remains to be done. Our current concerns include, in particular, the full development of a calculus of software connectors emerging from the coalgebraic semantic framework and its use in reasoning about typical *software architectural patterns* [1, 12] and their laws. How easily this work scales up to accommodate *dynamically re-configurable* architectures, as in, *e.g.*, [11] or [26], remains an open challenging question. We are also currently working on the development of an HASKELL based platform for prototyping this model, allowing the user to define and compose, in an interactive way, his/her own software connectors.

Acknowledgements. This research was carried on in the context of the PURE Project supported by FCT, the Portuguese Foundation for Science and Technology, under contract `POSI/ICHS/44304/2002`.

References

1. R. Allen and D. Garlan, *A formal basis for architectural connection*, ACM TOSEM **6** (1997), no. 3, 213–249.
2. F. Arbab, *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science, **14** (2004), no. 3, 329–366.
3. ———, *Abstract behaviour types: a foundation model for components and their composition*, Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02) (F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), Springer Lect. Notes Comp. Sci. (2852), 2003, pp. 33–70.
4. F. Arbab and J. Rutten, *A coinductive calculus of component connectors*, CWI Tech. Rep. SEN-R0216, CWI, Amsterdam, 2002, To appear in the proceedings of WADT'02.
5. L. S. Barbosa, *Components as processes: An exercise in coalgebraic modeling*, FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems (S. F. Smith and C. L. Talcott, eds.), Kluwer Academic Publishers, September 2000, pp. 397–417.
6. ———, *Towards a Calculus of State-based Software Components*, Journal of Universal Computer Science **9** (2003), no. 8, 891–909.
7. M.A. Barbosa and L.S. Barbosa, *A Relational Model for Component Interconnection*, Journal of Universal Computer Science **10** (2004), no. 7, 808–823.
8. K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy, *A Formal Model for Componentware*, Foundations of Component-Based Systems (Gary T. Leavens and Murali Sitaraman, eds.), Cambridge University Press, 2000, pp. 189–210.
9. R. Bird and O. Moor, *The algebra of programming*, Series in Computer Science, Prentice-Hall International, 1997.
10. M. Broy, *Semantics of finite and infinite networks of communicating agents*, Distributed Computing (1987), no. 2.
11. G. Costa and G. Reggio, *Specification of abstract dynamic data types: A temporal logic approach*, Theor. Comp. Sci. **173** (1997), no. 2.

12. J. Fiadeiro and A. Lopes, *Semantics of architectural connectors*, Proc. of TAPSOFT'97, Springer Lect. Notes Comp. Sci. (1214), 1997, pp. 505–519.
13. D. Gelernter and N. Carrier, *Coordination languages and their significance*, Communication of the ACM **2** (1992), no. 35, 97–107.
14. R. Grimes, *Profissional dcom programming*, Wrox Press, 1997.
15. He Jifeng, Liu Zhiming, and Li Xiaoshan, *A contract-oriented approach to component-based programming*, Proc. of FACS'03, (*Formal Approaches to Component Software*) (Pisa) (Z. Liu, ed.), Spetember 2003.
16. V. Matena and B Stearns, *Applying entreprise javabeans: Component-based development for the j2ee platform*, Addison-Wesley, 2000.
17. Sun Meng and L. S. Barbosa, *On refinement of generic software components*, 10th Int. Conf. Algebraic Methods and Software Technology (AMAST) (Stirling) (C. Rettray, S. Maharaj, and C. Shankland, eds.), Springer Lect. Notes Comp. Sci. (3116), 2004, pp. 506–520.
18. O. Nierstrasz and F. Achermann, *A calculus for modeling software components*, Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02) (F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, eds.), Springer Lect. Notes Comp. Sci. (2852), 2003, pp. 339–360.
19. O. Nierstrasz and L. Dami, *Component-oriented software technology*, Object-Oriented Software Composition (O. Nierstrasz and D. Tsichritzis, eds.), Prentice-Hall International, 1995, pp. 3–28.
20. S. Oaks and H. Wong, *Jini in a nutshell*, O'Reilly and Associates, 2000.
21. G. Papadopoulos and F. Arbab, *Coordination models and languages*, Advances in Computers — The Engineering of Large Systems, vol. 46, 1998, pp. 329–400.
22. J. Rutten, *Elements of stream calculus (an extensive exercise in coinduction)*, Tech. report, CWI, Amsterdam, 2001.
23. J.-G. Schneider and O. Nierstrasz, *Components, scripts, glue*, Software Architectures - Advances and Applications (L. Barroca, J. Hall, and P. Hall, eds.), Springer-Verlag, 1999, pp. 13–25.
24. R. Siegel, *CORBA: Fundamentals and programming*, John Wiley & Sons Inc, 1997.
25. C. Szyperski, *Component software, beyond object-oriented programming*, Addison-Wesley, 1998.
26. M. Wermelinger and J. Fiadeiro, *Connectors for mobile programs*, IEEE Trans. on Software Eng. **24** (1998), no. 5, 331–341.

Replicative - Distribution Rules in P Systems with Active Membranes

Tseren-Onolt Ishdorj[1,2] and Mihai Ionescu[2]

[1] Computer Science and Information Technology School,
Mongolian State University of Education,
Baga Toiruu-14, 210648 Ulaanbaatar, Mongolia
[2] Research Group on Mathematical Linguistics,
Rovira i Virgili University,
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain

Abstract. P systems (known also as *membrane systems*) are biologically motivated theoretical models of distributed and parallel computing. The two most interesting questions in the area are completeness (solving every solvable problem) and efficiency (solving a hard problem in feasible time). In this paper we define a general class of P systems covering some biological operations with membranes. We introduce a new operation, called *replicative-distribution*, into P systems with active membranes. This operation is well motivated from a biological point of view, and elegant from a mathematical point of view. It is both computationally powerful and efficient. More precisely, the P systems with active membranes using replicative-distribution rules can compute exactly what Turing machines can compute, and can solve **NP**-complete problems, particularly `SAT`, in linear time.

1 Introduction

Traditionally, theoretical computer science has played the role of a scout that explores novel approaches towards computing well in advance of other sciences. This did also occur in the case of membrane computation.

In the history of computing, electronic computers are only the latest in a long chain of man's attempts to use the best technology available for doing computations. While it is true that their appearance, some 50 years ago, has revolutionized computing, computing does not start with electronic computers, and there is no reason why it should end with them. Indeed, even electronic computers have their limitations: there is only so much data they can store and their speed thresholds determined by physical laws will soon be reached. The latest attempt to break down these barriers is to replace, once more, the tools for doing computations: instead of electrical use biological ones.

An important achievement in this direction was brought by Leonard Adleman in 1994, [1], when he surprised the scientific community by using the tools of molecular biology to solve a hard computational problem. Adleman's exper-

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 68–83, 2005.
© Springer-Verlag Berlin Heidelberg 2005

iment, solving an instance of the Directed Hamiltonian Path Problem by manipulating DNA strands marked the first instance of a mathematical problem being solved by biological means. The experiment provoked an avalanche of computer science/molecular biology/biochemistry/physics research, while generating at the same time a multitude of open problems.

The understanding of computations in nature – evolutionary computing, neural computing, and molecular computing – belong to the emerging area of *natural computing*, which is concerned with computing which go on in nature or is inspired by nature.

Membrane computing is a novel emerging branch of natural computing, introduced by Gheorghe Păun in [10]. This area starts from the observation that certain processes which take place in the complex structure of living cells can be considered as computations. P systems are a class of distributed parallel computing devices of a biochemical type, which can be seen as a general computing architecture where various types of objects can be processed by various operations. For a detailed description of various P system models we refer to [12].

In membrane computing, P systems with active membranes have a special place, because they provide biologically inspired tools to solve computationally hard problems. Using the possibility to divide or separate membranes, one can create an exponential working space in linear time, which can then be used in a parallel computation for solving, e.g., **NP**-complete problems in polynomial or even linear time. Details can be found in [2, 11, 12], as well as in the comprehensive page from the web address `http://psystems.disco. unimib.it.` Informally speaking, in P systems with active membranes one uses the following types of rules: (a_0) multiset rewriting rules, (b_0) rules for introducing objects into membranes, (c_0) rules for sending objects out of membranes, (d_0) rules for dissolving membranes, (e_0) rules for dividing elementary membranes, and (f_0) rules for dividing non-elementary membranes, see [3]. In these rules, a single object is involved. The following rules are introduced in [2]: (g_0) membrane merging rules, (h_0) membrane separation rules, and (i_0) membrane release rules, whose common feature is that they involve multisets of objects.

In this paper, we introduce a new developing rule in P systems, called *replicative-distribution rule*, which is motivated from the specific structure and functioning of living neural-cell (neuron). The universality and efficiency related to replicative-distribution rules are investigated here.

Let us briefly mention the biological background of our new developing rules. A neuron has a *body*, the *dendrites*, which form a very fine filamentary bush around the body of the neuron, and the *axon*, a unique, long filament, which in turn also ends with a fine filamentous bush; each of the filaments from the end of the axon is terminated with a small bulb. It is by means of these end-bulbs and the dendrites that the neurons are linked to each other: the impulses are sent through the axon, from the body of the neuron to the end-bulbs, and the end-bulbs transmit the impulses to the neurons whose dendrites they touch. Such a contact junction between an end-bulb of an axon and dendrites of another neuron is called cleft. An end-bulb releases an impulse into cleft, the impulse is

replicated in the cleft and distributed into the connected dendrites. Moreover, in the axon of the neuron, chemicals are replicated at the so-called Ranvier nodes and transmitted to the adjacent nodes in opposite directions through the axon. For more details about neural biology, we refer to [17].

2 Preliminaries

We assume the reader to be familiar to the fundamentals of formal language theory and complexity theory, for instance, from [9, 15, 16], as well as to the basics of membrane computing, from [12]. We only mention here some notions and results from formal language theory, complexity theory, as well as from membrane computing, which are used in this paper.

2.1 Formal Languages

An *alphabet* is a finite set of symbols (letters), and a word (string) over an alphabet Σ is a finite sequence of letters from Σ. We denote the empty word by λ, the length of a word w by $|w|$, and the number of ocuurences of a symbol a in w by $|w|_a$. The concatenation of two words x and y is denoted by $x \cdot y$ or simply xy.

A *language* over Σ is a (possibly infinite) set of words over Σ. The language consisting of all words over Σ is denoted by Σ^*, and Σ^+ denotes the language $\Sigma^* - \{\lambda\}$. A set of languages containing at least one language not equal to $\oslash$ or $\{\lambda\}$ is also called a family of languages.

We denote by `REG`, `LIN`, `CF`, `CS`, `RE` the families of languages generated by regular, linear, context-free, context-sensitive, and of arbitrary grammars, respectively (`RE` stands for recursively enumerable languages). By `FIN` we denote the family of finite languages. The following strict inclusions hold:

$$\texttt{FIN} \subset \texttt{REG} \subset \texttt{LIN} \subset \texttt{CF} \subset \texttt{CS} \subset \texttt{RE}.$$

This is the Chomsky hierarchy.

For a family `FL` of languages, `NFL` denotes the family of length sets of languages in `FL`. Therefore, `NRE` is the family of Turing computable sets of natural numbers. For $a \in \Sigma$ and $x \in \Sigma^*$ we denote by $|x|_a$ the number of occurrences of a in x. Then, for $\Sigma = \{a_1, \cdots, a_n\}$, the *Parikh mapping* associated with Σ is the mapping on Σ^* defined by $\Psi_\Sigma(x) = (|x|_{a_1}, \cdots, |x|_{a_n})$ for each $x \in \Sigma^*$. The Parikh images of languages `RE` is denoted by $PsRE$ (this is the family of all recursively enumerable sets of vectors of natural numbers).

The multisets over a given finite support (alphabet) are represented by strings of symbols. The order of symbols does not matter, because the number of copies of an object in a multiset is given by the number of occurrences of the corresponding symbol in the string. Clearly, using strings is only one of many ways to specify multisets. We suggest the readers refer to [4].

We will now introduce the notion of matrix grammars, used below in proofs.

A *matrix grammar with appearance checking* is a computationally universal rewriting system. Details can be found in [5]. For each matrix grammar there is an equivalent matrix grammar in the binary normal form.

A matrix grammar $G = (N, T, S, M, F)$ is in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \#\}$, with these three sets mutually disjoint, and the matrices in M are in one of the following forms:

1. $(S \to XA)$, with $X \in N_1, A \in N_2$,
2. $(X \to Y, A \to x)$, with $X, Y \in N_1, A \in N_2, x \in (N_2 \cup T)^*, |x| \leq 2$,
3. $(X \to Y, A \to \#)$, with $X, Y \in N_1, A \in N_2$,
4. $(X \to \lambda, A \to x)$, with $X \in N_1, A \in N_2$, and $x \in T^*, |x| \leq 2$.

Moreover, there is only one matrix of type 1 (that is why one uses to write it in the form $(S \to X_{init}A_{init})$, in order to fix the symbols X, A present in it), and F consists exactly of all rules $A \to \#$ appearing in matrices of type 3; $\#$ is a trap-symbol, because once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of a derivation.

For $w, z \in (N \cup T)^*$ we write $w \Longrightarrow z$ if there is a matrix in $m \in M$ such that applying once each rule of m to w one can obtain z. A rule can be skipped if it is in F and it is not applicable.

The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}. It is known that $MAT_{ac} = RE$.

2.2 P Systems with Active Membranes

In this subsection, we describe P systems with active membranes following the concept defined in [12], where more details can also be found.

A *membrane structure* is represented by a Venn diagram and is identified by a string of correctly matching parentheses, with a unique external pair of parentheses; this external pair of parentheses corresponds to the external membrane, called the *skin*. A membrane without any other membrane inside is said to be *elementary*. For instance, the structure in Figure 1 contains 8 membranes; membranes 3, 5, 6 and 8 are elementary. The string of parentheses identifying this structure is

$$\mu = [\,[\,[\,\,]_5[\,\,]_6]_2[\,\,]_3[\,[\,[\,\,]_8]_7]_4]_1.$$

All membranes are labeled; we have used here the numbers from 1 to 8. We say that the number of membranes is the *degree* of the membrane structure, while the height of the tree associated in the usual way with the structure is its *depth*. In the example above we have a membrane structure of degree 8 and of depth 3. The membranes delimit *regions* precisely identified by the membranes (the region of a membrane is delimited by the membrane and all membranes placed immediately inside it, if such a membrane exists). In these regions we place *objects*, which are represented by symbols of an alphabet. Several copies of the same object can be present in a region, so we work with *multisets* of objects.

We will now define the model which we work with: P systems with active membranes. A P system *with active membranes* (without electrical charges) is a construct

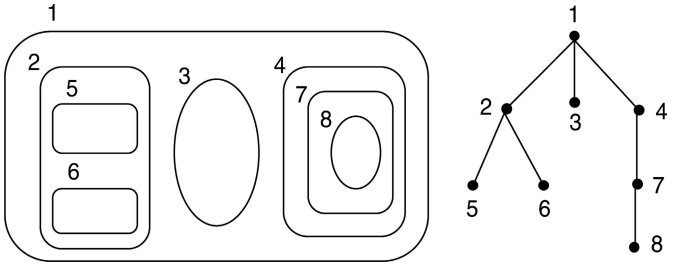


Fig. 1. A membrane structure and its associated tree

$$\Pi = (O, H, \mu, w_1, \ldots, w_m, R),$$

where:

- $m \geq 1$ is the initial degree of the system;
- O is the alphabet of *objects*;
- H is a finite set of *labels* for membranes;
- μ is a *membrane structure*, consisting of m membranes, labeled (not necessarily in a one-to-one manner) with elements of H;
- $w_1, \ldots, w_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ;
- R is a finite set of *developmental rules*, of the following forms:

(a_0) $[\ a \to v\ ]_h$, for $h \in H, a \in O, v \in O^*$
(object evolution rules, associated with membranes and depending on the label, but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them);

(b_0) $a[\ \ ]_h \to [\ b]_h$, for $h \in H, a, b \in O$
(communication rules; an object is introduced in the membrane during this process);

(c_0) $[\ a\ ]_h \to [\ \ ]_h b$, for $h \in H, a, b \in O$
(communication rules; an objects sent out of the membrane during this process);

(d_0) $[\ a\ ]_h \to b$, for $h \in H, a, b \in O$
(dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule can be modified);

(e_0) $[\ a]_h \to [\ b]_h[\ c]_h$, for $h \in H, a, b, c \in O$
(division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label; the object specified in the rule is replaced in the two new membranes by possibly new objects; and the remaining objects are duplicated);

(f_0) $[\ a]_h \to [\ b]_h[\ c]_h$, for $h \in H, a, b, c \in O$
(division rules for non-elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label; the object specified in the rule is replaced in the two new membranes by

possibly new objects; the remaining objects and membranes contained in this membrane are duplicated, and then are part of the contents of both new copies of the membrane);

(g_0) $[\,]_{h_1}[\,]_{h_2} \to [\,]_{h_3}$, for $h_i \in H, 1 \leq i \leq 3$
(merging rules for elementary membranes; in reaction of two membranes, they are merged into a single membrane; the objects of the former membranes are put together in the new membrane);

(h_0) $[\,O]_h \to [\,U]_h[\,O-U]_h$, for $h \in H, U \subset O$
(separation rules for elementary membranes; the membrane is separated into two membranes with the same labels; the objects from U are placed in the first membrane, those from $U - O$ are placed in the other membrane);

(i_0) $[\,[\,O]_{h_1}]_{h_2} \to [\,]_{h_2}O$, for $h_1, h_2 \in H$
(release rule; the objects in a membrane are released out of a membrane, surrounding it, while the first membrane disappears).

The rules of types (a_0), (b_0), (c_0), (d_0), (e_0), and (f_0) are the polarizationless version of the corresponding rules in [12]; the rules of (g_0), (h_0), and (i_0) are introduced in [2]. (In all cases, the subscript 0 indicates the fact that we do not use polarization for membranes; in [11], [13] the membranes can have one of the "electrical charges negative, positive, neutral, represented by $-$, $+$, 0, respectively. Note that, following [3], we have omitted the label of the left parenthesis from a pair of parentheses which identifies a membrane.)

The rules of type (a_0) are applied in the parallel way (all objects which can evolve by such rules have to evolve), while the rules of types (b_0), (c_0), (d_0), (e_0), (f_0), (g_0), (h_0), and (i_0) are used sequentially, in the sense that one membrane can be used by at most one rule of these types at a time. In total, the rules are used in the non-deterministic maximally parallel manner: all objects and all membranes which can evolve, should evolve.

The result of a halting computation is the vector of natural numbers describing the multiplicity of objects expelled into the environment during the computation; the set of vectors computed in this way by all possible halting computations of Π is denoted by $Ps(\Pi)$. A P system is called *deterministic* if there is a single computation. A P system is called *confluent* if all of its computations reach the same halting configuration.

By $PsOP_m(r)$ we denote the family of sets $Ps(\Pi)$ computed as described above by P systems with at most m membranes using rules of types listed in r.

When the rules of a given type (α_0) are able to change the label(s) of the involved membranes, then we denote that type of rules by (α_0').

P systems with certain combinations of these rules are universal and efficient. Further details can be found in [2, 3, 8].

To understand what solving a problem in a semi-uniform/uniform way means, we briefly recall here some related notions. Consider a decisional problem X. A family $\Pi_X = (\Pi_X(1), \Pi_X(2), \cdots)$ of P systems (with active membranes in our case) is called *semi-uniform* (*uniform*) if its elements are constructible in polynomial time starting from $X(n)$ (from n, respectively), where $X(n)$ denotes

the instance of size n of X. We say that X can be solved in polynomial (linear) time by the family Π_X if the system $\Pi_X(n)$ will always stop in a polynomial (linear, respectively) number of steps, sending out the object `yes` if and only if the instance $X(n)$ has a positive answer. For more details about complexity classes for P systems see [12, 13].

3 Replicative-Distribution Rules

The biological motivations of *replicative-distribution* operations are mentioned in Section 1. Mathematically, we capture the idea of replicative-distribution rules as following:

(k_0) $a[\,]_{h_1}[\,]_{h_2} \rightarrow [\,u]_{h_1}[\,v]_{h_2}$, for $h_1, h_2 \in H, a \in O, u, v \in O^*$
(replicative-distribution rule (for sibling membranes); an object is replicated and distributed into inner two adjacent membranes);

(l_0) $[\,a[\,]_{h_1}]_{h_2} \rightarrow [\,[\,u]_{h_1}]_{h_2} v$, for $h_1, h_2 \in H, a \in O, u, v \in O^*$
(replicative-distribution rule (for nested membranes); an object is replicated and distributed into a directly inner membrane and outside the directly surrounding membrane).

The rules are applied non-deterministically, in the maximally parallel manner. Note that the multisets u and v might be empty.

As we have mentioned before, we use the primed versions to indicate the fact that the labels of membranes can be changed. The primed versions of replicative-distribution rules are of the following form:

(k_0') $a[\,]_{h_1}[\,]_{h_2} \rightarrow [\,u]_{h_3}[\,v]_{h_4}$ for $h_i \in H, 1 \leq i \leq 4$
(the label of both or only one membrane can be changed);

(l_0') $[\,a[\,]_{h_1}]_{h_2} \rightarrow [\,[\,u]_{h_3}]_{h_4} v$, for $h_i \in H, 1 \leq i \leq 4$
(the label of both or only one membrane can be changed).

3.1 Computational Universality

P systems with active membranes and with particular combinations of several types of rules can reach universality. Here, we show that P systems with active membranes and with only one type of rules, namely (l_0'), is Turing complete. The proof is based on the simulation of matrix grammars with appearance checking.

Theorem 1. $PsOP_4(l_0') = PsRE$.

Proof. It is enough to prove that any recursively enumerable set of vectors of non-negative integers can be generated by a P system with active membranes using rules of type (l_0') and four membranes.

Consider a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the binary normal form, hence with $N = N_1 \cup N_2 \cup \{S, \#\}$ and with the matrices of the four forms introduced in Section 2.1. Assume that all matrices are injectively labeled with elements of a set B. Replace the rule $X \rightarrow \lambda$ from matrices of type 4 by $X \rightarrow f$, where f is a new symbol.

We construct the P system of degree 4

$$\begin{aligned}
\Pi &= (O, H, \mu, w_0, w_1, w_2, w_{X_{init}}, R),\\
O &= T \cup N_2 \cup \{A_m \mid A \in N_2, m \in B\} \cup \{c, c', c'', c''', \lambda, \#\},\\
H &= N_1 \cup \{X_m \mid X \in N_1, m \in B\} \cup \{0, 1, 2, f\},\\
\mu &= [\,[\,[\,[\,\,]_2]_1]_{X_{init}}]_0,\\
w_0 &= cA_{init}, w_{X_{init}} = w_1 = w_2 = \lambda.
\end{aligned}$$

and the set R containing the rules below.

The simulation of a matrix $m : (X \to Y, A \to x)$, with $X \in N_1, Y \in N_1 \cup \{f\}$, and $A \in N_2$, is done in two steps, using the following replicative-distribution rules:

1. $[\,A[\,\,]_X]_0 \to [\,[\,A_m]_{Y_m}]_0\lambda$,
2. $[\,A_m[\,\,]_1]_{Y_m} \to [\,[\,\lambda]_1]_Y x$.

The first rule of the matrix is simulated by the change of the label of membrane X, and the correctness of this operation is obvious (one cannot simulate one rule of the matrix without simulating at the same time also the other rule).

The simulation of a matrix $m : (X \to Y, A \to \#)$, with $X, Y \in N_1$, and $A \in N_2$, is done in four steps, using the rules:

3. $[\,c[\,\,]_X]_0 \to [\,[\,c']_{Y_m}]_0\lambda$,
4. $[\,c'[\,\,]_1]_{Y_m} \to [\,[\,c'']_1]_{Y'_m}\lambda$,
5. $[\,A[\,\,]_{Y'_m}]_0 \to [\,[\,\#]_f]_0\lambda$,
   $[\,c''[\,\,]_2]_1 \to [\,[\,\lambda]_2]_1 c'''$,
6. $[\,c'''[\,\,]_1]_{Y'_m} \to [\,[\,\lambda]_1]_Y c$.

By using rule 3, object c replicates to c' and λ which are distributed, in the same time, as follows: c' enters membrane X changing its label to Y_m and λ is send out of the skin membrane. The second step (rule 4) makes c' to evolve to λ and c''; c'' will be sent to membrane 1 and λ gets out of membrane Y_m changing it to Y'_m. In the next step, if any copy of A is present, then, it introduces the trap-object $\#$ and the computation never stops. Otherwise, c'' following the same replicative-distribution rule transforms into λ and c''', which enter membranes 1 and Y'_m, respectively. The last computational step produces the result we were looking for by replicating c''' to λ and c and distributing λ to membrane 1 and c to the skin membrane, changing label Y_m to Y. Now, the process can be iterated having c in the skin membrane as in its initial configuration.

We also consider the following rules (applicable in the case A is present in the skin membrane):

7. $[\,\#[\,]_1]_f \to [\,[\,\lambda]_1]_f\#$,
8. $[\,\#[\,\,]_f]_0 \to [\,[\,\#]_f]_0\lambda$.

The equality $\Psi_T(L(G)) = Ps(\Pi)$ easily follows from the above explanations. □

3.2 Efficiency Result Using Pre-computed Resources

The SAT problem (satisfiability of propositional formula in the conjunctive normal form) is probably the most known **NP**-complete problem [6]. It asks whether or not for a given formula in the conjunctive normal form there is a truth-assignment of variables such that the formula assumes the value *true*.

Let us consider a propositional formula in the conjunctive normal form:

$$\begin{aligned}\beta &= C_1 \wedge \cdots \wedge C_m,\\ C_i &= y_{i,1} \vee \cdots \vee y_{i,l_i},\ 1 \le i \le m, \text{ where}\\ y_{i,k} &\in \{x_j, \neg x_j \mid 1 \le j \le n\},\ 1 \le i \le m, 1 \le k \le l_i.\end{aligned}$$

The instance β of SAT will be encoded in the rules of P system by multisets v_j and v'_j of symbols, corresponding to the clauses satisfied by *true* and *false* assignment of x_j, respectively:

$$\begin{aligned}v_j &= \{c_i \mid x_j \in \{y_{i,k} \mid 1 \le k \le l_i\}, 1 \le i \le m\}, 1 \le j \le n,\\ v'_j &= \{c_i \mid \neg x_j \in \{y_{i,k} \mid 1 \le k \le l_i\}, 1 \le i \le m\}, 1 \le j \le n.\end{aligned}$$

A computation in a P systems with active membranes always starts from a given initial configuration, and we usually create an exponential workspace in linear time by membrane division, membrane creation, string replication, and membrane separation (all of them with biological motivation). In this subsection we use a different strategy (which is already discussed in [12]): we start from an arbitrarily large initial membrane structure, without objects placed in its regions, and we trigger a computation by introducing objects related to a given problem in a specified membrane. We use object replicative-distribution rules, as discussed in the previous sections. In this way, the number of objects can increase exponentially.

Theorem 2. *P systems with rules of types* (k'_0) *and* (l'_0)*, constructed in a semi-uniform manner, can solve* SAT *in linear time.*

Proof. Given a propositional formula β as above, we construct the P system

$$\begin{aligned}\Pi &= (O, H, \mu, w_a, w_b, w_c, w_d, w_e, w_0, w_1, w_2, w_{4n+5-m}, w_{4n+6}, R), \text{with}\\ O &= \{a_i \mid 0 \le i \le n\} \cup \{d_i \mid 1 \le i \le m\} \cup \{e_i \mid 0 \le i \le 4n+m+1\}\\ &\quad \cup \{t_i, f_i \mid 1 \le i \le n\} \cup \{c_i \mid 1 \le i \le m\} \cup \{\mathtt{yes}, \mathtt{no}, d, \lambda\},\\ \mu &= \underbrace{[\,[\,[\,[\,]_a]_b]_c[\,\cdots[\,]_d[\,]_e\cdots]_1[\,\cdots[\,]_d[\,]_e\cdots]_2]_0}_{2^{n+4}+2},\\ w_0 &= a_0, w_c = e_0, w_1 = w_2 = w_a = w_b = w_d = w_e = w_{4n+5-m} = w_{4n+6} = \lambda,\\ H &= \{0, \cdots, 4n+6, a, b, c, d, e\}.\end{aligned}$$

The membrane structure has to be a complete binary tree with $n+2$ internal levels for the constructions of truth value assignments except the membranes for global counting. In the skin membrane, 3 "nested" membranes with labels

a, b, and c are used to count the computations of the system. The "sibling membranes", those placed in the same upper membrane, directly under it, are labeled with 1 and 2. We consider the skin membrane as being level 0 (root) of our binary tree. It is obvious that on level 1 we have 2 (2^1) membranes, on level 2 we have 4 (2^2) membranes, and so on. The membranes on levels $1, \cdots, n$ are labeled 1 and 2, of level $n+1$ are labeled $4n+5-m$ and $4n+6$, and elementary membranes are labeled by d and e. The skin membrane is labeled 0.

We give the set of rules R accompanying them with their use explanations:

– Global control:

E1. $[\ e_i[\ ]_b]_c \rightarrow [\ [\ e_{i+1}]_b]_c\lambda$,

E2. $[\ e_i[\ ]_a]_b \rightarrow [\ [\ \lambda]_a]_b e_{i+1}, 0 \leq i \leq 4n+m-1$,

The control variables e_i count the computing steps in the "nested" control membranes. As we shall see at the end of the description of the whole algorithm, after $4n+m$ derivation steps in the corresponding P system Π the answer `yes` appears outside the skin membrane if the given satisfiability problem has a solution, whereas in the case that no solution exits, in one or two more steps the answer `no` appears in the environment.

The main task of the algorithm is accomplished in the generation phase where, for each possible truth assignment to the n variables. After $2n-1$ steps it will contain all the informations needed to decide whether it represents a solution to the given problem or not:

– Generation phase:

G1. $a_i[\ ]_1[\ ]_2 \rightarrow [\ a_{i+1}t_{i+1}]_3[\ a_{i+1}f_{i+1}]_4, 0 \leq i \leq n-1$,

G2. $t_i[\ ]_{i+2}[\ ]_{i+3} \rightarrow [\ t_i]_{i+4}[\ t_i]_{i+5}$,
$f_i[\ ]_{i+2}[\ ]_{i+3} \rightarrow [\ f_i]_{i+4}[\ f_i]_{i+5}, 1 \leq i \leq n$,

Starting the computation (rule G1 in skin membrane), object a_0 is replicated into objects a_1t_1 and a_1f_1, which are distributed into direct inner membranes with label 1 and 2 of level 2, changing the labels to 3 and 4.

Let us consider step i of the generation phase: By applying rule G1 in the membranes of level i, 2^i number of couples of objects a_it_i and a_if_i are produced and took place in the 2^i number of membranes, which will change the label from 1 or 2 to 3 or 4, respectively. Each membrane among the 2^{i-j} membranes on levels $i-j$, $1 \leq j \leq [\ i/2]$ of hierarchal binary tree structure contains couple of objects p_{i-j} and q_j, $(p_r, q_r \in \{t_r, f_r\})$, $1 \leq j \leq [\ i/2]$ if j is an odd number. Otherwise, in the membranes of $[\ i/2]$ th level only objects p_{i-j} are placed. Up to now, $t_1, \cdots, t_i, f_1, \cdots, f_i$, and a_i different 2^i number of objects have been produced and distributed in the membranes of levels between $i-j$ and i, $1 \leq j \leq [\ i/2]$ presenting the truth value assignments for variables $x_1, \cdots, x_i$ of β. Objects t_k, $1 \leq k \leq i$ correspond to the *true* value of variables x_k, and objects f_k correspond to the $false$ value of variables x_k, $1 \leq k \leq i$. In the next step, rule G1 is applied and objects $a_{i+1}t_{i+1}$ and $a_{i+1}f_{i+1}$ are produced and took place in inner membranes, changing the labels of them to 3 and 4. At the same time, rules

$$t_k[\ ]_{k+2}[\ ]_{k+3} \rightarrow [\ t_k]_{k+4}[\ t_k]_{k+5},$$

$$f_k[\,]_{k+2}[\,]_{k+3} \rightarrow [\,f_k]_{k+4}[\,f_k]_{k+5}$$

are applied simultaneously in each membranes of levels $i-j$, $1 \leq j \leq [\,i/2]$, objects t_k and f_k are replicated and distributed in one deeper level. Membrane labels are changed from $k+2$ and $k+3$ to $k+4$ and $k+5$, respectively, which guarantees that in each step only a single object, t_k or f_k, enters into a membrane, since active membranes work in sequential manner.

At the nth step of the computation, 2^n number of couple objects $a_n p_n$, $p_n \in \{t_n, f_n\}$, were in 2^n membranes with label 3 and 4 on nth level, then rule G1 will not apply anymore since there is no membrane with label 1 and 2 at the next level. The iteration is continued $n-1$ more steps, all objects t_k, f_k, $1 \leq k \leq n-1$ are within the membranes of level n, then, those membranes' label being $2n+1$ and $2n+2$. Therefore all possible 2^n truth assignments of variables $x_1, x_2, \cdots, x_n$ are generated and placed in the corresponding 2^n membranes. Objects t_i correspond to the *true* value of variables x_i, and objects f_i correspond to the $false$ value of variables x_i.

G3. $t_i[\,]_{4n+5-m}[\,]_{4n+6} \rightarrow [\,v_i]_{4n+5-m}[\,d]_{4n+6}$,
$f_i[\,]_{4n+5-m}[\,]_{4n+6} \rightarrow [\,v_i']_{4n+5-m}[\,d]_{4n+6}, 1 \leq i \leq n.$

By using rule G3, in n steps, every object t_i and f_i evolve into objects c_i (corresponding to clauses C_i, satisfied by the *true* or $false$ values chosen for x_i) and "dummy" object d, then they are distributed into membranes with label $4n+5-m$ and $4n+6$ in one deeper level, respectively.

– Checking phase:

C1. $[\,c_i[\,]_d]_{4n+4-m+i} \rightarrow [\,[\,c_i]_d]_{4n+5-m+i} d_i, 1 \leq i \leq m.$

In the checking phase, by using rule C1, object $c_i, 1 \leq i \leq n$, is placed in membranes labeled $4n+5-m$ of level $n+1$, and replicated into object c_i and counter object d_i. Object c_i is sent into the direct inner elementary membrane with label d, which is on the deepest level $(n+2)$ of our membrane structure, and object d_i is sent out the surrounding membrane on nth level. Meanwhile, the label of the surrounding membrane is incremented by one. If at the beginning of the checking phase $c_1, \cdots, c_i$ are present $(1 \leq i < m)$, and c_{i+1} is absent, in the membrane, after $i+1$ steps rule C1 will no longer be applicable and the membrane will never change the label again. If all objects $c_i, 1 \leq i \leq m$, are present in some membrane, then after m steps, objects d_m are produced into the membranes with label $2n+1$ and $2n+2$ of level n.

– Output phase:

O1. $[\,d_m[\,]_{2n+5+2i}]_{1+2i} \rightarrow [\,[\,d]_{2n+5+2i}]_{2n+3+2i} d_m$,
O2. $[\,d_m[\,]_{2n+6+2i}]_{2+2i} \rightarrow [\,[\,d]_{2n+6+2i}]_{2n+4+2i} d_m$, $1 \leq i \leq n$
O3. $[\,d_m[\,]_{2n+5}]_0 \rightarrow [\,[\,d]_{2n+5}]_1$yes,

If β has solutions, the process starts when objects d_m are placed in membranes with label $2n+1$ and/or $2n+2$ of level n. Object d_m is replicated to objects d and d_m, object d_m is sent out the current membrane, and "dummy" object d is sent into the inner membrane with label $2n+5+2i$. The process is recurrently

done following objects d_m through levels $n \cdots 1$ in n steps by using rule O1 and O2. During the output phase, in each membranes with label $1+2i$ and $2+2i$ possible taken at most two objects d_m, then one of them non-deterministically chosen send out the surrounding membrane, while membrane label is changed to $2n+3+2i$ and $2n+4+2i$. Then, in membranes $2n+3+2i$ and $2n+4+2i$ no rule can be applied. Thus, the system works fine in a sequential manner. However, in n steps, totally speaking in the $(4n+m-1)$th step of the computation, at most two objects d_m arrive in the skin membrane. Then rule O3 is applied, an object d_m ejects positive answer `yes` and changes skin membrane label to 1 in order to prevent further output. Thus, the formula is satisfiable and the computation stops. That was the $(4n+m)$th step of the whole computation.

E3. $[\ e_{4n+m(-1)}[\ ]_b]_c \rightarrow [\ [\ \lambda]_b]_c e_{4n+m(+1)}$,
E4. $[\ e_{4n+m(+1)}[\ ]_c]_0 \rightarrow [\ [\ \lambda]_c]_0 \texttt{no}$.

If β has no solution and if $4n+m-1$ is an odd step, counter object e_{4n+m-1} must be placed in the membrane a, then rules E3 and E4 are applied in two steps, the counter object e_{4n+m+1} will eject the correct answer `no` to the environment. Otherwise after one more step object e_{4n+m} will eject the correct answer `no` to the environment by applying rule E4. Since rule O3 did not apply (the case in which β has no solution), the label of the skin membrane is still 0, so rule E3 is applicable. The 3 "nested" control membranes guarantee that no object tries to cross the skin membrane at the same time with `yes`.

The labels of membranes of level i, in the constructing phase, are $3+2(i-1)$ and $4+2(i-1)$, $1 \leq i \leq n$, and in the output phase it would be $3+2(i+n)$ and $4+2(i+n)$, $1 \leq i \leq n$. □

Theorem 3. *P systems with rules of types (k_0') and (c_0'), constructed in a semi-uniform manner, can solve* `SAT` *in linear time.*

Proof. The proof of the theorem follows the idea of Theorem 2. Since rules of type (l_0') are not used in the proof, we do not need membranes of level $n+2$ in the checking phase, and we change the "nested" couple membrane structure by "eyes" structure for the global control.

We now construct the P system

$$\Pi = (O, H, \mu, w_a, w_b, w_0, w_1, w_2, w_{4n+5-m}, w_{4n+6}, R), \text{with}$$
$$O = \{a_i \mid 0 \leq i \leq n\} \cup \{d_i \mid 1 \leq i \leq m\} \cup \{e_i \mid 0 \leq i \leq 4n+m+1\}$$
$$\cup \{t_i, f_i \mid 1 \leq i \leq n\} \cup \{c_i \mid 1 \leq i \leq m\} \cup \{\texttt{yes}, \texttt{no}, d, \lambda\},$$
$$\mu = \underbrace{[\ [\ ]_a[\ ]_b[\cdots[\ ]_{4n+5-m}[\ ]_{4n+6}\cdots]_1[\cdots[\ ]_{4n+5-m}[\ ]_{4n+6}\cdots]_2]_0}_{2^{n+3}+1},$$
$$w_0 = a_0 e_0, w_1 = w_2 = w_a = w_b = w_{4n+5-m} = w_{4n+6} = \lambda,$$
$$H = \{0, \cdots, 4n+6, a, b\}.$$

The global control rules are as following:

– Global control in skin membrane:

E1. $e_i[\ ]_a[\ ]_b \to [\ e_{i+1}]_a[\ \lambda]_b$,
E2. $[\ e_i]_a \to [\ ]_a e_{i+1}, 0 \leq i \leq 4n+m+1$,

Here we use rules of types (k_0) and (c_0) for counting the computation steps of the system.

– Generation phase:

We reuse rules G1-G3 of the generation phase in Theorem 2, then generate 2^n number of truth-assignments in level n of the membrane structure.

– Checking phase:

C1. $[\ c_i]_{4n+4-m+i} \to [\ ]_{4n+5-m+i} d_i, 1 \leq i \leq m$.

In the checking phase of satisfiability truth-assignments of propositional formula, rules of type (c_0') are used instead of rules (l_0').

– Output phase:

O1. $[\ d_m]_{1+2i} \to [\ \ ]_{2n+3+2i} d_m$,
O2. $[\ d_m]_{2+2i} \to [\ \ ]_{2n+4+2i} d_m$, $1 \leq i \leq n$
O3. $[\ d_m]_0 \to [\ \ ]_1 \texttt{yes}$,
E3. $[\ e_{4n+m(+1)}]_0 \to [\ \ ]_0 \texttt{no}$.

If β has solutions, in n steps, totally speaking in $(4n+m-1)$th step of the computation, at most two objects d_m arrive in the skin membrane by using rules O1 and O2, then rule O3 is applied, an object d_m ejects positive answer `yes` and changes the skin label to 1 in order to prevent further output. Thus, the formula is satisfiable and the computation stops. That was the $(4n+m)$th step of the whole computation.

If β has no solution and if $4n+m-1$ is an odd step, after two more steps the counter object e_{4n+m+1} will eject the correct answer *no* to the environment. Otherwise, after one more step object e_{4n+m} will perform this operation. Since rule O3 did not apply (the case in which β has no solution), the label of the skin membrane is still 0, so rule E3 is applicable. □

3.3 Efficiency Result Using Membrane Division to Obtain Exponential Work Space

In Theorem 2 and Theorem 3 we have shown that the NP-complete problem `SAT` can be decided by a P system with active membranes in linear time with replicative-distribution rules of types (k_0') and (l_0') and replicative-distribution rules of type (k_0') and communication rules of type (c_0'), respectively, using precomputed exponential work space.

Here, we reuse the most investigated way to obtain exponential work space–membrane division. The following theorem shows that `SAT` can be solved by P systems with active membranes using the rules of types (f_0) and (l_0'), in linear time. We recall here the propositional formula β in Section 3.2.

Theorem 4. *P systems with rules of types* $(f_0), (l'_0)$*, constructed in a semi-uniform manner, can deterministically solve* SAT *in linear time with respect to the number of the variables and the number of clauses.*

Proof. We construct the P system

$$\begin{aligned}
\Pi &= (O, H, \mu, w_0, \cdots, w_7, R), \text{ with} \\
O &= \{d_i \mid 1 \le i \le m\} \cup \{a_i \mid 1 \le i \le n\} \\
&\quad \cup \{c_i \mid 1 \le i \le m\} \cup \{b_i \mid 0 \le i \le n\} \\
&\quad \cup \{e_i \mid 0 \le i \le 2n+m+4\} \cup \{\mathtt{yes}, \mathtt{no}\} \\
&\quad \cup \{t_i, f_i \mid 1 \le i \le n\}, \\
\mu &= [\,[\,[\,[\,\,]_3]_4]_2[\,[\,[\,\,]_6]_7]_5]_0, \\
w_2 &= a_1 \cdots a_n b_0, w_5 = e_0, w_0 = w_3 = w_4 = w_6 = w_7 = \lambda \\
H &= \{i \mid 0 \le i \le 9\},
\end{aligned}$$

and the following rules (we accompany them with explanations about their use):

The global control rules are as follows:

– Global control:

E1. $[\, e_i[\,\,]_7]_5 \to [\,[\, e_{i+1}]_7]_5\lambda$,

E2. $[\, e_i[\,\,]_6]_7 \to [\,[\,\lambda]_6]_7 e_{i+1}, 0 \le i \le 2n+m+1$,

The "nested" membranes with label 5,7, and 6 are used only to globally control of the computation, and rules E1 and E2 are used to count the computation steps as we used in the proof of Theorem 2.

– Generation phase:

G1. $[\, a_i]_2 \to [\, t_i]_2[\, f_i]_2$, $1 \le i \le n$,

Using rule G1, with a_i non-deterministically chosen, we produce the truth values *true* and $false$ assigned to variable x_i, placed in two separate copies of membrane 2. In this way, in n steps we assign truth values to all variables, hence we get all 2^n truth-assignments, placed in 2^n separate copies of membrane 2.

G2. $[\, b_i[\,\,]_4]_2 \to [\,[\, b_{i+1}]_4]_2\lambda$,
$[\, b_i[\,\,]_3]_4 \to [\,[\,\lambda]_3]_4 b_{i+1}$, for all $0 \le i \le n-1$,

G3. $[\, b_n[\,\,]_3]_4 \to [\,[\,\lambda]_3]_1\lambda$,

G4. $[\, b_n[\,\,]_4]_2 \to [\,[\,\lambda]_1]_2\lambda$,

Initially, object b_0 is placed in membrane 2. Rule G2 works simultaneously with division and increment the subscript of b_i by one in each step. If in the nth step of the computation, object b_n takes place in membrane 2, it was an odd number. If it was an even number, object b_n takes place in membrane 4. In the next step rule G3 or G4 perform, and change the label of membrane 4 to 1, while object b_n disappears. This ensure rule G5 will perform.

G5. $[\, t_i[\,\,]_1]_2 \to [\,[\, v_i]_1]_2\lambda$,
$[\, f_i[\,\,]_1]_2 \to [\,[\, v'_i]_1]_2\lambda$, $1 \le i \le n$.

– Checking phase:

C1. $[\ c_i[\ \ ]_3]_i \rightarrow [\ [\ c_i]_3]_{i+1} d_i, 1 \leq i \leq m.$

The checking phase idea is the same from the proof of Theorem 2.

– Output phase:

O1. $[\ d_m[\ \ ]_{m+1}]_2 \rightarrow [\ [\ \lambda]_{m+1}]_8 d_m,$

O2. $[\ d_m[\ \ ]_8]_0 \rightarrow [\ [\ \lambda]_8]_9 \texttt{yes},$

If β has solutions, after $2n + m + 2$ steps, objects d_m appear in the skin membrane using rules O1, and again one object d_m, non-deterministically chosen, ejects object `yes` into environment, while the skin label changes to 9 using rule O2 in order to prevent further output. Thus, the formula is satisfiable and the computation stops. That was the $(2n+m+3)$th step of the whole computation.

E3. $[\ e_{2n+m+2(3)}[\ \ ]_7]_5 \rightarrow [\ [\ \lambda]_7]_5 e_{2n+m+3(4)},$

E4. $[\ e_{2n+m+3(4)}[\ \ ]_5]_0 \rightarrow [\ [\ \lambda]_5]_0 \texttt{no}.$

If β has no solution and if $2n + m + 2$ is an odd step, after two more steps the counter object e_{2n+m+4} will eject the correct answer `no` to the environment. Otherwise, after one more step object e_{2n+m+3} will perform this operation. Since rule O2 did not apply (the case in which β has no solution), the label of the skin membrane is still 0, so rule E4 is applicable. □

4 Final Remarks

We have considered a new type of rules in P systems with active membranes: (k_0) and (l_0') replicative-distribution rules with deep relations to cell biology. We have illustrated here how this type of rules can solve **NP**-complete problems in linear time using pre-computed resources and obtaining an exponential work space during the computation, by membrane division. Universality was also shown here, but we want to emphasize that we have used only one type of rules in our proof. However, in the efficiency results, we have used very few types of rules compared to the previous results in [2, 3, 7, 8, 11]. This reveals the fact that replicative-distribution type of rules is a powerful and efficient tool in P systems. The following problems are expecting a future work: What simulations of other classes of P systems with active membranes using these new types of rules can be obtained? What other computational hard problems can be solved with these types of rules in feasible time and space?

Acknowledgments. The first author acknowledges the State Training Fund of the Ministry of Science, Technology, Education and Culture of Mongolia. The work of second author was supported by the FPU fellowship from the Spanish Ministry of Education, Culture and Sport.

References

1. L.M. Adlmen, Molecular computation of solutions to combinatorial problems, *Science* v.266, Nov.1994, 1021–1024.
2. A. Alhazov, T.-O. Ishdorj, Membrane Operations in P Systems with Active Membranes, In: Gh. Păun, et all (eds.) *Second Brainstorming Week on Membrane Computing*, Sevilla, 2-7 February, 2004, Research Group in Natural Computing **TR 01/2004**, University of Sevilla, 37–44.
3. A. Alhazov, L. Pan, Gh. Păun, Trading Polarizations for Labels in P Systems with Active Membranes, submitted, 2003.
4. C. Calude, Gh. Păun, G. Rozenberg, A. Salomaa (eds.): *Multiset Processing*, LNCS 2235, Springer-Verlag, Berlin, 2001.
5. J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.
6. M.R. Garey, D.J. Johnson: *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.
7. L. Pan, A. Alhazov, T.-O. Ishdorj, Further Remarks on P Systems with Active Membranes, Separation, Merging and Release Rules, *Soft Computing*, 8(2004), 1–5.
8. L. Pan, T.-O. Ishdorj, P Systems with Active Membranes and Separation Rules, *Journal of Universal Computer Science*, 10(5)(2004), 630–649.
9. Ch. P. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading, MA, 1994.
10. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences*, 61(1), (2000), 108–143, and TUCS Research Report 208, 1998 (http://www.tucs.fi).
11. Gh. Păun, P Systems with Active Membranes: Attacking NP-Complete Problems, Journal of Automata, Languages and Combinatorics, 6, 1 (2001), 75–90.
12. Gh. Păun, *Computing with Membranes: An Introduction*, Springer-Verlag, Berlin, 2002.
13. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, Complexity Classes in Models of Cellular Computation with Membranes, *Natural Computing*, 2, 3 (2003), 265–285.
14. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini, *Teoría de la Complejidad en Modelos de Computatión Celular con Membranas*, Editorial Kronos, Sevilla, 2002.
15. A. Salomaa, *Formal Languages*, Academic Press, New York, 1973.
16. A. Salomaa, G. Rozenberg (eds.), *Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997.
17. G.M. Shepherd, *Neurobiology*, Oxford University Press, NY Oxford, 1994.

A Generalisation of a Relational Structures Model of Concurrency*

Ryszard Janicki

Department of Computing and Software, McMaster University,
Hamilton, Ontario, Canada L8S 4K1
janicki@mcmaster.ca

Abstract. We show how complex concurrent behaviours can be modelled by relational structures $(X, \diamond, \sqsubset)$, where X is a set (of event occurrences), $\diamond$ (interpreted as *commutativity*), $\sqsubset$ (interpreted as *weak causality*) are binary relations on X. The paper is a continuation of the approach initiated in [6, 18, 1, 9], substantially developed in [10, 12], and recently partially generalized in [7]. For the first time an axiomatic model for the most general case is given. The results can be interpreted as a generalisation of Szpilrajn Theorem[1] [25].

1 Introduction

The classical "true concurrency" model semantics make the assumption that all relevant behavioural properties of non-sequential systems can be adequately expressed in terms of causal partial orders. This assumption is arbitrary and the model, although very successful in the majority of applications, is unable to properly describe some aspects of systems with priorities, error recovery systems, inhibitor nets, time testing etc (see for instance [12, 10, 13, 19, 26] and many others).

The solution, first introduced by Lamport [18] (improved by Abraham, Ben-David and Magodor [1]), Gaifman and Pratt [6], and Janicki and Koutny [9], later fully developed by Janicki and Koutny [10, 12], is to use relational structures, $(X, <, \sqsubset)$, with two relations. The first relation, denoted by "$<$" in [10, 12], is "causality" (i.e. an abstraction of "earlier than"), the second, denoted by "$\sqsubset$" in [10, 12] is called "weak causality" and is an abstraction of "not later than" relation. The classical "interleaving" and "true concurrency" models are distinctive special cases. The papers [10, 12] provide the theoretical foundations of the model (results of [18, 6, 1] are special cases) and prove its soundness.

The model has been successfully applied to inhibitor systems [11, 2, 14, 17], priority systems [13, 16], asynchronous races [28, 29], synthesis [22, 24], and has influenced many other approaches [3, 27]. It was shown in Janicki and Koutny [10] that relational structures of the type $(X, <, \sqsubset)$ *still cannot model the most general case* and that the most general case requires relational stuctures of the

* Partially supported by NSERC of Canada Grant.

[1] Every partial order is the intersection of all its total order extensions.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 84–98, 2005.
© Springer-Verlag Berlin Heidelberg 2005

type $(X, \diamond, \sqsubset)$, where $\diamond$, called "commutativity", is an abstraction of "interleaving" relation, and $< = \diamond \cap \sqsubset$.

An axiomatic model for the structures of the type $(X, \diamond, \sqsubset)$ was recently proposed by Guo and Janicki [7]. The model of [7] is restricted, it assumes that all the observations (runs, executions) are modelled by stratified partial orders (or, equivalently, step sequences). *In this paper an axiomatic model for the structures of the type* $(X, \diamond, \sqsubset)$ *with no restrictions is given.*

To illustrate the main ideas, let us consider the following four, very simple programs, which nonetheless are reflective of the essence of the problem. All of the programs are written using a mixture of *cobegin, coend* and a version of (concurrent) *guarded commands.*

```
P1: begin int x;
     a: x:=0;
     cobegin b: x:=x+1, c: x:=x+2 coend
     end P1.
P2: begin int x,y;
     a: begin x:=0; y:=0 end;
     cobegin
     b: x=0 → y:=y+1, c: x:=x+1,
     coend
     end P2.
P3: begin int x,y;
     a: begin x:=0; y:=0 end;
     cobegin
     b:y=0 → x:=x+1, c: x=0 → y:=y+1
     coend
     end P3.
P4: begin int x,y;
     a: begin x:=0; y:=0 end;
     cobegin b: x:=x+1, c: y:=y+1 coend
     end P4.
```

Each program is a different composition of three events (actions) called a, b and c respectively (a_i, b_i, c_i, $i = 1, ..., 4$, to be exact, but a restriction to a, b, c, does not change the validity of the analysis below, while simplifying the notation).

What *concurrent behaviours* (*concurrent histories*) are generated by the above programs? Let us concentrate on the behaviours that involve all three actions $\{a, b, c\}$ (sometimes such behaviours are called *proper*). Let $obs(P_i)$ denote the set of all program runs involving the actions $\{a, b, c\}$ that can be observed. Assume that simultaneous executions can be observed. In this simple case all runs can be modelled by *step-sequences* (or equivalently *stratified orders*), with simultaneous execution of a_1,...,a_n denoted by $\{a_1, ..., a_n\}$. Let us denote $o_1 = abc$, $o_2 = acb$, $o_3 = a\{b, c\}$. Each o_i can be seen as a partial order $o_i = (\{a, b, c\}, \stackrel{o_i}{\rightarrow})$, where, less formally: $o_1 = a \stackrel{o_1}{\rightarrow} b \stackrel{o_1}{\rightarrow} c$, $o_2 = a \stackrel{o_2}{\rightarrow} c \stackrel{o_2}{\rightarrow} b$, $o_3 = a \stackrel{o_3}{\rightarrow} b \wedge a \stackrel{o_3}{\rightarrow} c$. We can now write $obs(P_1) = \{o_1, o_2\}$, $obs(P_2) = \{o_1, o_3\}$, $obs(P_3) = \{o_3\}$, $obs(P_4) = \{o_1, o_2, o_3\}$.

Note that for each $obs(P_i)$ all runs from $obs(P_i)$ yield to exactly the same outcome (so this justify to call $obs(P_i)$'s as *concurrent histories*).

An abstract model of such outcome is called a *concurrent behaviour*, but what entity constitutes such a model? Let us start with the set $obs(P_4)$. We may say that in this case for each run, a always precedes both b and c, and there is no *casual* relationship between b and c. This *causality* relation, $<$, is a partial order defined as $< = \{(a,b),(a,c)\}$. Formally $<$ is defined as $x < y$ iff for each run o we have $x \stackrel{o}{\rightarrow} y$. One may notice that $<$ is an intersection of o_1, o_2 and o_3, and that $\{o_1, o_2, o_3\}$ is the set of all stratified extensions of the relation[2] $<$. Thus in this case the causality relation $<$ can model the concurrent behaviour that corresponds to the set of (equivalent) runs $obs(P_4)$. This is a classical case of "true" concurrency approach, where concurrent behaviours are modelled by causality relations. Before considering the remaining cases, note that the causality relation $<$ is exactly the same in all four cases, i.e. $<_i = \{(a,b),(a,c)\}$, for $i = 1, ..., 4$, so we may omit the index i. Let us consider now the set $obs(P_1)$.

The causality relation $<$ does not model the concurrent behaviour correctly[3] since o_3 does not belong to $obs(P_1)$. Let $\diamond$ be a symmetric relation, called *commutativity*, defined as $x \diamond y$ iff for each run o either $x \stackrel{o}{\rightarrow} y$ or $y \stackrel{o}{\rightarrow} x$. For the set $obs(P_1)$, the relation $\diamond_1 = \{(a,b),(b,a),(a,c),(c,a),(b,c),(c,b)\}$. A set of relations $\{\diamond_1, <\}$ and the set $obs(P_1)$ are equivalent in the sense that each one defines another (the set $obs(P_1)$ can be defined as the greatest set PO of partial orders built from a, b and c satisfying $x \diamond_1 y \Rightarrow \forall o \in PO.\ x \stackrel{o}{\rightarrow} y \vee y \stackrel{o}{\rightarrow} x$ and $x < y \Rightarrow \forall o \in PO.\ x \stackrel{o}{\rightarrow} y$). We may say that in this case the relations $\{\diamond_1, <\}$ model the concurrent behaviour described by $obs(P_1)$. Note also that $\diamond_i = < \cup <^{-1}$ for $i = 2, 3, 4$, so the set $\{\diamond_4, <\}$ models the concurrent behaviour described by $obs(P_4)$ as well.

To deal with $obs(P_2)$ and $obs(P_3)$ we need another relation, $\sqsubset$, called *weak causality*, and defined as $x \sqsubset y$ iff for each run o we have $\neg(y \stackrel{o}{\rightarrow} x)$ (x *is never executed after* y). For our four cases we have $\sqsubset_2 = \{(a,b),(a,c),(b,c)\}$, $\sqsubset_3 = \{(a,b),(a,c),(b,c),(c,b)\}$, and $\sqsubset_1 = \sqsubset_4 = <$. One may observe that for $i = 2, 3$, a set of relations $\{<, \sqsubset_i\}$ and the set $obs(P_i)$ are equivalent in the sense that each one defines another (the set $obs(P_i)$ can be defined as the greatest set PO of partial orders built from a, b and c satisfying $x < y \Rightarrow \forall o \in PO.\ x \stackrel{o}{\rightarrow} y$ and $x \sqsubset_i y \Rightarrow \forall o \in PO.\ \neg(y \stackrel{o}{\rightarrow} x)$). We may say that in this case the relations $\sqsubset_i$, $i = 2, 3$, models the concurrent behaviour described by $obs(P_i)$. Note that $\sqsubset_i$ alone is not sufficient, for instance $obs(P_2)$ and $obs(P_2) \cup \{\{a,b,c\}\}$ define the same $\sqsubset$. The relations $<$, $\diamond$, $\sqsubset$ are not independent, it can be proven ([10]) that $< = \diamond \cap \sqsubset$.

---

[2] The fact that $<$ equals to $\stackrel{o_3}{\rightarrow}$ is a coincidence, there are not so many partial orders built from three elements. No order is interpreted differently, it means no causal relationship for $<$ and simultaneous execution for $\stackrel{o_3}{\rightarrow}$.

[3] Unless we assume that simultaneity is not allowed, or not observed, in such a case $obs(P_1) = obs(P_4) = \{o_1, o_2\}$, $obs(P_2) = \{o_1\}$, $obs(P_3) = \emptyset$.

Summing up we have:

1. all $obs(P_i)$, for $i = 1, 2, 3, 4$ are modelled by appropriate pairs of relations $\{\diamond_i, \sqsubset_i\}$,
2. $obs(P_i)$, for $i = 2, 3, 4$ can also be modelled by appropriate pairs of relations $\{<, \sqsubset_i\}$,
3. $obs(P_4)$ can be modelled by the relation $<$ alone.

The theory developed in [10] provides a hierarchy of models of concurrency, where each model corresponds to a so called "paradigm", or general rule about the structures of concurrent histories. In principle, a paradigm says how simultaneity is handled in concurrent histories. The paradigms are denoted by π_1 through π_8. It appears only paradigms π_1, π_3, and π_8 are interesting from the point of view of concurrency theory. The most general paradigm, π_1, assumes no additional restrictions for concurrent histories. The most restrictive paradigm, π_8, simply says that if a set of partial orders Δ is a concurrent history then $(\exists o \in \Delta.\ x \overset{o}{\leftrightarrow} y) \iff (\exists o \in \Delta.\ x \overset{o}{\rightarrow} y) \wedge (\exists o \in \Delta.\ y \overset{o}{\rightarrow} x)$, where $\overset{o}{\leftrightarrow}$ denotes simultaneity, i.e. $x \overset{o}{\leftrightarrow} y \iff \neg(x \overset{o}{\rightarrow} y) \wedge \neg(y \overset{o}{\rightarrow} x)$. The paradigm, π_3, assumes that if a set of partial orders Δ is a concurrent history then $(\exists o \in \Delta.\ x \overset{o}{\rightarrow} y) \wedge (\exists o \in \Delta.\ y \overset{o}{\rightarrow} x) \Rightarrow (\exists o \in \Delta.\ x \overset{o}{\leftrightarrow} y)$.

In the case of P_1, P_2, P_3, P_4 programs, obviously all $obs(P_i)$, $i = 1, 2, 3, 4$, conform to paradigm π_1, $obs(P_2)$, $obs(P_3)$, $obs(P_4)$, conform to paradigm π_3, and $obs(P_4)$ conforms to π_8. It can be proven [10] that π_3 implies $\diamond = < \cup <^{-1}$, and π_8 implies $\diamond = < \cup <^{-1}$ and $<$ equals to $\sqsubset$.

The most restrictive case, π_8, corresponds to the classical "true concurrency" model where causal partial orders are sufficient to model all aspects of concurrent behaviour. In the "true concurrency" model, an equivalence of the formula that defines π_8 is called a "Diagonal Property" [4, 5].

The problem is: *What axioms the triples* $(X, \diamond, \sqsubset)$ *or* $(X, <, \sqsubset)$ *must satisfy to be considered as models of concurrent behaviours?*

The paradigm π_i is only one of the factors shaping concurrent histories (i.e. the sets $obs(P_i)$ for our example). Another important factor is the kind of partial orders that observable runs are allowed to be. It is argued in [10] that observable runs of (discrete) software systems should be modelled by initially finite interval orders, however the results of [12] cover general partial orders as well. Observable runs are frequently assumed to be stratified orders or even total orders. This makes the modelling simpler, and such assumptions are often justified. It appears that the axioms for $(X, \diamond, \sqsubset)$ and $(X, <, \sqsubset)$ depend heavily on what kind of partial orders the observable runs are allowed to be. Under the assumption that only totally ordered runs are allowed, all paradigms are equivalent, $<$ alone models concurrent behaviour and the relationship between sets of runs and the relation $<$ follows directly from Szpilrajn theorem [25]. A detailed discussion of triples $(X, <, \sqsubset)$ that model concurrent behaviours under the assumption of paradigm π_3, is given in [12].

A solution to the case $(X, \diamond, \sqsubset)$, i.e. paradigm π_1, but under the assumption that all runs must be stratified orders was recently presented in [7]. *In this paper two remaining solutions to the case* $(X, \diamond, \sqsubset)$ *are given, first, under the*

assumption that all runs are interval orders, and second, runs are just general partial orders.

There are two major different attitudes towards abstracting non-sequential behaviour, one based on *interleaving abstraction* (for instance [21]), and another based on *partially ordered causality* (for instance [4, 5, 20] etc.). Both models have been very successful. The interleaving models are very structured and compositional, the partial order models can handle fairness, confusion, etc. Both models are mathematically much less complex, far more developed than the models with two relations, and they sufffice in the majority of standard applications. Nevertheless some aspect of concurrent behaviour are difficult or almost impossible to tackle by both interleaving and partially ordered causality based models. E.g., specification of priorities, error recovery, time testing, proper treatment of simultaneity, are *in some circumstances* problematic [12, 10, 13, 19, 26].

From the purely mathematical viewpoint the results of this paper can be seen as an extension of the Szpilrajn Theorem [25] to orders that are not necessary total. Alternatively, the results show how sets of equivalent partial orders can be represented by two relations.

2 Relational Structure Model of Concurrency

In order to make this paper self-contained, we briefly recall all the main results of [10, 12].

A *partial order*, is a pair $po = (X, <)$ such that X is a non-empty set and $<$ is an irreflexive and transitive relation on X. We say taat X is the domain of po. Sometime we also say that $<$ is a partial order in X. Two distinct incomparable elements a and b of X will be denoted by $a \sim b$, and we will write $a <^{\sim} b$ if $a < b$ or $a \sim b$.

A partial order $(X, <)$ is said to be

- *total* if for all $a, b \in X$, either $a < b$ or $b < a$ or $a = b$.
- *stratified* if $\sim \cup\, id_X$, where id_X is identity on X, is an equivalence relation[4];
- *interval*[5] if for all $a, b, c, d \in X$, $a < b \wedge c < d \Longrightarrow a < d \vee c < b$.
- *initially finite* if for every $a \in X$, $\{b \mid b <^{\sim} a\}$ is finite.

It is easy to see that a total order is a stratified order and a stratified order is an interval one. Stratified orders correspond to step sequences. Modelling concurrency usually assume some form of discreteness, for instance the number of predecessors is finite, etc. This is captured by the concept of initial finiteness. It turns out many results need separate proofs under initial finiteness assumption. In general, if $\mathcal{C}$ is a class of partial orders we will denote by $\mathcal{C}_{IF}$ the subclass of all initially finite partial orders in $\mathcal{C}$. A partial order p_1 is an *extension* of another partial order p_2 if they have the same domain and $<_{p_2} \subseteq <_{p_1}$.

[4] An equivalent definition: a poset $(X, <)$ is a stratified order iff there exists a total order $(Y, \prec)$ and a mapping $\phi : X \to Y$ such that $\forall a, b \in X.\ a < b \iff \phi(a) \prec \phi(b)$.

[5] The name and intuition follow from Fishburn Theorem [8], see Theorem 3.3 in the next section. Very often Fishburn Theorem is used as a definition of interval orders.

A *run* (*observation, instance of concurrent behaviour*) is an abstract model of the execution of a concurrent system. It was argued in [10] that an observation must be an initially finite, either total, or stratified, or interval order. The results of [12] are valid for all kinds of partial orders, not necessarily initially finite nor interval, however separate proofs are frequently required for different cases. Following [10, 12], we will make a distinction in notation between general posets and those used as runs. We will use $o = (X, \rightarrow_o)$ rather than $po = (X, <)$ to denote a generic run, and use $\leftrightarrow_o$ rather than $\sim$ to denote incomparability.

A *complete set of equivalent runs* is a *concurrent history*[6]. To explain the concept assume that all runs are total orders. A set $\Delta = \{abc, cba\}$ is not a concurrent history, since it implies that there is no causal relationship between a, b, and c (as the intersection of abc and cba, denoted by $<_\Delta$ is empty). Let Δ^{cl} be the set of all total extensions of Δ, i.e. $\Delta^{cl} = \{abc, bac, acb, bca, cab, aba\}$. The set Δ^{cl} is complete as it is the set of all total extensions of $<_{\Delta^{cl}} = \emptyset$, so it can be considered as a concurrent history.

If runs are not all total orders, a definition of concurrent histories requires using two intrinsic characteristics of the runs. Let Δ be a set of partial orders with a common domain X.

Define the relations $\diamond_\Delta$ and $\sqsubset_\Delta \subseteq X \times X$ as

- $x \diamond_\Delta y \iff \forall o \in \Delta.\ (x \overset{o}{\rightarrow} y \vee y \overset{o}{\rightarrow} x)$,
- $x \sqsubset_\Delta y \iff \forall o \in \Delta.\ (x \overset{o}{\rightarrow} y \vee x \overset{o}{\leftrightarrow} y)$.

We say that a partial order (run) $o = (X, \overset{o}{\rightarrow})$ is an *extension* of $\diamond_\Delta$ if and only if

- $\forall x, y \in X.\ x \diamond_\Delta y \Rightarrow (x \overset{o}{\rightarrow} y \vee y \overset{o}{\rightarrow} x)$

and it is an *extension* of $\sqsubset_\Delta$ if and only if

- $\forall x, y \in X.\ x \sqsubset_\Delta y \Rightarrow (x \overset{o}{\rightarrow} y \vee x \overset{o}{\leftrightarrow} y)$.

Let Δ^{cl} be the set of all posets $o = (X, \overset{o}{\rightarrow})$ that are extensions of both $\diamond_\Delta$ and $\sqsubset_\Delta$.

Definition 2.1. A set of runs Δ is a *concurrent history* iff $\Delta = \Delta^{cl}$. □

All $obs(P_i)$, $i = 1, ..., 4$, satisfy $obs(P_i) = obs(P_i)^{cl}$. For detailed discussion of the above definition, see [10, 12].

The problem is how to find axioms for the relations $\diamond$ and $\sqsubset$ such that their partial order extensions could be interpreted as some Δ^{cl}. To solve this problem the notion of a *paradigm* has been introduced.

As we mentioned earlier, a paradigm is a superposition or a statement about the structure of a history involving a treatment of simultaneity. For instance, let Δ be a concurrent history. The classical causality based approach usually stipulates that if there is a run $o \in \Delta$ such that $a \overset{o}{\leftrightarrow} b$, then there must be a run such a precedes b and a run such that b precedes a. Formally, *paradigms*, $\omega \in Par$, are defined by

[6] The term "concurrent history" has been used by many authors, e.g., [5, 15, 20] and others, to denote formally different concepts (atlhough intuitively close) in the idea of concurrency. The concept used in this paper was inroduced in [9] and is close to that of [20].

$$\omega := true|false|\Psi_1|\Psi_2|\Psi_3|\neg\omega|\omega \vee \omega|\omega \wedge \omega|\omega \Rightarrow \omega,$$

where $\Psi_1(\beta,\gamma) = \exists o.\ \beta \xrightarrow{o} \gamma, \Psi_2(\beta,\gamma) = \exists o.\ \beta \xleftarrow{o} \gamma$ and $\Psi_3(\beta,\gamma) = \exists o.\ \beta \overset{o}{\leftrightarrow} \gamma$, where β, γ are some variables.

A history Δ satisfies a paradigm $\omega \in Par$ if for all distinct $a, b \in dom(\Delta)$, $\omega(a,b)$ holds. It can be shown (see [10]) that in the study of concurrent histories, we only need to consider 8 paradigms, denoted by $\pi_1, \cdots, \pi_8$. From those eight, only π_1, π_3 and π_8 are important. The most general paradigm, $\pi_1 = true$, admits all concurrent histories. The most restrictive paradigm, π_8, admits concurrent histories Δ such that

$$(\exists o \in \Delta.\ x \overset{o}{\leftrightarrow} y) \ \Leftrightarrow\ (\exists o \in \Delta.\ x \xleftarrow{o} y) \wedge (\exists o \in \Delta.\ x \xrightarrow{o} y).$$

It was proven that in this case causality, $<$, suffices to fully describe Δ, so the use of it to model concurrent behaviour is justified. The paradigm π_3, which is general enough to deal with most problems that cannot be dealt with under π_8, admits concurrent histories Δ such that

$$(\exists o \in \Delta.\ x \xleftarrow{o} y) \wedge (\exists o \in \Delta.\ x \xrightarrow{o} y)\ \Rightarrow\ (\exists o \in \Delta.\ x \overset{o}{\leftrightarrow} y).$$

It was proven that in this case, causality, $<$, and weak causality (an abstraction of "not later than"), $\sqsubset$, suffice to fully describe Δ. The axioms for relational structures $(X, <, \sqsubset)$, such that the sets of their partial order extensions can be interpreted as concurrent histories Δ^{cl}, were provided in [12]. We briefly show them below.

Definition 2.2. A *two relation structure*, or simply a *structure*, is a triple $S = (X, <, \sqsubset)$ where X is a non-empty set and $<, \sqsubset$ are two irreflexive binary relations on X such that for all $a, b \in X$, $a < b \Rightarrow \neg b \sqsubset a$. □

Note that at this point *we do not assume any other properties* of $<$ and $\sqsubset$. Until further notice $<$ and $\sqsubset$ do not have any interpretation. For any irreflexive relation R, let $R^{\sim}$ be defined as $xR^{\sim}y \iff \neg(yRx)$.

Definition 2.3. Let $S = (X, <, \sqsubset)$ be a structure. An irreflexive relation R over X is *an extension* of S, i.e. $R \in ext(S)$, if and only if for all $x, y \in X$, we have $(x < y \ \Rightarrow\ xRy) \wedge (x \sqsubset y \ \Rightarrow\ xR^{\sim}y)$. □

Let Θ be a non-empty class of structures and let $S = (X, <, \sqsubset) \in \Theta$. We define $ext_{\Theta}(S) = \{R \mid R \in ext(S) \wedge (X, R, R^{\sim}) \in \Theta\}$.

Definition 2.4. A class of structures Θ is *extension complete* if for every $S = (X, <, \sqsubset) \in \Theta$,

- $ext_{\Theta}(S) \neq \emptyset$,
- $< = \bigcap_{R \in ext_{\Theta}(S)} R$, and
- $\sqsubset = \bigcap_{R \in ext_{\Theta}(S)} R^{\sim}$. □

Let $\mathcal{T}$ be a class of structures defined as follows $S = (X, <, \sqsubset) \in \mathcal{T}$ iff $<$ is a partial order and $<$ equals to $\sqsubset$, i.e. $S = (X, <, <)$. One may easily show

that for each $S \in \mathcal{T}$, $ext_{\mathcal{T}}(S)$ consists of total orders only (if $R = R^{\sim}$ and R is a partial order, then R must be a total order). By Szpilrajn theorem [25], $\mathcal{T}$ is extension complete. The class $\mathcal{T}$ is called *total order structures*. This class is not very interesting, as its members are just partial orders but it creates a bottom of the hierarchy developed in [12].

Definition 2.5. A relational structure $S = (X, <, \sqsubset)$ is called a *stratified*, *interval* and *partial order structure* if the following conditions S1-S4, I1-I6 and P1-P4 are satisfied respectively:

$$\begin{array}{ll} S1\ \ a \not\sqsubset a & S3\ \ a \sqsubset b \sqsubset c \Rightarrow a \sqsubset c \vee a = c \\ S2\ \ a < b \Rightarrow a \sqsubset b & S4\ \ a \sqsubset b < c \vee a < b \sqsubset c \Rightarrow a < c. \end{array}$$

$$\begin{array}{ll} I1\ \ a \not\sqsubset a & I4\ \ a < b \sqsubset c \vee a \sqsubset b < c \Rightarrow a \sqsubset c \\ I2\ \ a < b \Rightarrow a \sqsubset b & I5\ \ a < b \sqsubset c < d \Rightarrow a < d \\ I3\ \ a < b < c \Rightarrow a < c & I6\ \ a \sqsubset b < c \sqsubset d \Rightarrow a \sqsubset d \vee a = d \end{array}$$

$$\begin{array}{ll} P1\ \ a \not\sqsubset a & P3\ \ a < b < c \Rightarrow a < c \\ P2\ \ a < b \Rightarrow a \sqsubset b & P4\ \ a \sqsubset b < c \vee a < b \sqsubset c \Rightarrow a \sqsubset c. \end{array}$$

□

We will denote by $\mathcal{S}, \mathcal{I}$ and $\mathcal{P}$ respectively the class of stratified, interval and partial order structures. One may verify easily that $\mathcal{T} \subset \mathcal{S} \subset \mathcal{I} \subset \mathcal{P}$. It was proven in [12] that

- for every $S \in \mathcal{S}$, all elements of $ext_{\mathcal{S}}(S)$ are *stratified orders*,
- for every $S \in \mathcal{I}$, all elements of $ext_{\mathcal{I}}(S)$ are *interval orders*,
- for every $S \in \mathcal{P}$, all elements of $ext_{\mathcal{P}}(S)$ are *partial orders*

which justifies the names. A discussion of differences between the above axioms (from [12]) and those of [18, 6] can be found in [12].

A structure $(X, <, \sqsubset)$ is said to be *initially finite* if $\{b \mid b <^{\sim} a\}$ is finite for all $a \in X$. As with partial orders, if Θ is a class of structures, we denote by $\Theta_{IF} \subseteq \Theta$ the subclass consisting of initially finite structures. The relational structures $S_2 = (\{a, b, c\}, <, \sqsubset_2)$ and $S_3 = (\{a, b, c\}, <, \sqsubset_3)$ that correspond to the programs P_2 and P_3 from the Introduction belong to $\mathcal{S}_{IF}$, and $ext_{\mathcal{S}_{IF}}(S_i) = obs(P_i)$ for $i = 2, 3$.

The main result of [12] is the following theorem.

Theorem 2.1 [12] The classes of ordered structures: $\mathcal{T}, \mathcal{S}, \mathcal{I}, \mathcal{P}, \mathcal{T}_{IF}, \mathcal{S}_{IF}, \mathcal{I}_{IF}, \mathcal{P}_{IF}$ are extension complete. □

This means the ordered structures (i.e. the triples $(X, <, \sqsubset)$) represent uniquely appropriate sets of partial orders, i.e. concurrent histories, so they can be used to model concurrent behaviours conforming to paradigm π_3. It is important to point out that the result of [12, 18, 6] are only valid under π_3, which suffices for most of the application, but it is not the most general case. Under π_3 we *have to* model the program P_1 by two sequential behaviours instead of more natural one concurrent behaviour.

3 Generalized Order Structures

This chapter is devoted to the new results. We start with refining and adapting some definitions from [7].

Definition 3.1. *A generalised structure* is a triple $G = (X, \diamond, \sqsubset)$ such that $X \neq \emptyset$, $\diamond$ and $\sqsubset$ are two irreflexive relations on X, $\diamond$ is symmetric, and $S_G = (X, <_G, \sqsubset)$, where $<_G = \diamond \cap \sqsubset$, is a structure. □

Definition 3.2. Let $G = (X, \diamond, \sqsubset)$ be a generalized structure. An irreflexive relation R over X is *an extension* of G, i.e. $R \in gext(G)$, iff for all $x, y \in X$, we have $(x \diamond y \Rightarrow x(R \cup R^{-1})y) \wedge (x \sqsubset y \Rightarrow xR^{\smile}y)$. □

Let Θ be a non-empty class of generalized structures and let $G = (X, \diamond, \sqsubset)$ is in Θ. We define $gext_\Theta(G) = \{R \mid R \in gext(G) \wedge (X, R, R^{\smile}) \in \Theta\}$.

Definition 3.3. A class of generalized structures Θ is *extension complete* if for every $G = (X, \diamond, \sqsubset) \in \Theta$,

- $gext_\Theta(S) \neq \emptyset$,
- $\diamond = \bigcap_{R \in gext_\Theta(S)} (R \cup R^{-1})$, and
- $\sqsubset = \bigcap_{R \in gext_\Theta(S)} R^{\smile}$. □

Let $\mathcal{GT}$ be a class of generalized structures defined as $G = (X, \diamond, \sqsubset) \in \mathcal{GT}$ if and only if $\diamond = \{(x, y) \mid x, y \in X \wedge x \neq y\}$ and $\sqsubset$ is a partial order. One may easily show that for each $G \in \mathcal{GT}$, $gext_{\mathcal{GT}}(G)$ consists of total orders only (the same argument as for $ext_{\mathcal{T}}(S)$ before), so by Szpilrajn theorem [25], $\mathcal{GT}$ is extension complete. The class $\mathcal{GT}$ is called *generalized total order structures*, its members are just partial orders in disguise, and it creates a bottom of our new hierarchy.

Definition 3.4. A generalized structure $G = (X, \diamond, \sqsubset)$ is called a *stratified*, *interval*, *partial order generalized structure* and *initially finite generalized structure* if the structure $S_G = (X, <_G, \sqsubset)$, where $<_G = \diamond \cap \sqsubset$, is stratified (axioms S1-S4), interval (axioms I1-I6), partial order structure (axioms P1-P4), and initially finite, respectively. □

We shall use $\mathcal{GT}$, $\mathcal{GS}$, $\mathcal{GI}$ and $\mathcal{GP}$ to denote respectively the classes of total, stratified, interval and partial order generalized structures. One may verify easily that $\mathcal{GT} \subset \mathcal{GS} \subset \mathcal{GI} \subset \mathcal{GP}$. If Θ is a class of generalized structures, we denote by $\Theta_{IF} \subseteq \Theta$ the subclass consisting of initially finite generalized structures. The relational structure $G_1 = (\{a, b, c\}, \diamond_1, \sqsubset_1)$ corresponding to the program P_1 from the Introduction belong to $\mathcal{GS}_{IF}$, and $gext_{\mathcal{GS}_{IF}}(G_1) = obs(P_1)$.

The following lemma (a generalization of Lemma 3 in [7]) gives some necessary and sufficient conditions for extension completeness.

Lemma 3.1. Let $G = (X, \diamond, \sqsubset)$ be a generalized structure and Ω is any set of relations that extends G. Then, $\diamond = \bigcap_{R \in \Omega} (R \cup R^{-1})$, and $\sqsubset = \bigcap_{R \in \Omega} R^{\smile}$ if and only if for all distinct $a, b \in X$ we have:

(a) $\neg(x \diamond y) \Rightarrow \exists R \in \Omega.\ \neg(xRy) \wedge \neg(yRx)$
(b) $\neg(x \sqsubset y) \Rightarrow \exists R \in \Omega.\ yRx.$

Proof. (Sketch) Similarly as the proof of Lemma 3 in [7]. □

The above lemma is used in proofs of our main results. The main result of [7] is the following theorem.

Theorem 3.1.[7] The classes of generalized ordered structures $\mathcal{GT}$, $\mathcal{GS}$, $\mathcal{GT}_{IF}$, $\mathcal{GS}_{IF}$ are extension complete. □

The main result of this paper is the following.

Theorem 3.2. The classes of generalized ordered structures $\mathcal{GI}$, $\mathcal{GP}$, $\mathcal{GI}_{IF}$, $\mathcal{GP}_{IF}$ are extension complete. □

Theorem 3.1 was proven using a technique developed in [12] to prove Theorem 2.1. It seems this technique is not enough to prove Theorem 3.2. In order to prove Theorem 3.2 we need the following results from [1, 8, 25].

Lemma 3.2.[25] Let $po = (X, <)$ be a partial order and $a, b \in X$, $a \sim b$. Define $Y = \{a\} \cup \{y \mid y < a\}$, $Z = \{b\} \cup \{z \mid b < z\}$.
Then $(X, <^{ab}) = (X, < \cup\, Y \times Z)$ is a partial order and $a <^{ab} b$. □

Theorem 3.3.[8]([10] for initially finite case) A partial order $po = (X, <)$ is *interval* (*interval and initially finite*) iff there exists a total (total and initially finite) order $(T, \prec)$ and two mappings $\varphi, \psi : X \to T$ such that for all $a, b \in X$, $\varphi(a) \prec \psi(a)$ and $a < b \iff \psi(a) \prec \varphi(b)$. □

Usually $\varphi(a)$ is interpreted as the beginning and $\psi(a)$ as the end of an *interval* a.

Theorem 3.4.[1] Let $S = (X, <, \sqsubset)$ be an interval order structure. Then there is a partial order $(T, \prec)$ and two mappings $\varphi, \psi : X \to T$ such that for all $a, b \in X$, $\varphi(a) \prec \psi(a)$ and:

$$a < b \iff \psi(a) \prec \varphi(b)$$
$$a \sqsubset b \iff \varphi(a) \prec \psi(b) \vee \varphi(a) = \psi(b).$$

□

It turns out the results of Theorem 3.4 also hold under additional assumption of initial finiteness.

Theorem 3.5. Let $S = (X, <, \sqsubset)$ be an *initially finite* interval order structure. Then there is an *initially finite* partial order $(T, \prec)$ and two mappings $\varphi, \psi : X \to T$ such that for all $a, b \in X$, $\varphi(a) \prec \psi(a)$ and:

$$a < b \iff \psi(a) \prec \varphi(b)$$
$$a \sqsubset b \iff \varphi(a) \prec \psi(b) \vee \varphi(a) = \psi(b).$$

Proof. (Sketch) We take $(T, \prec)$ from Theorem 3.4 and show that initial finiteness of S implies initial finiteness of $(T, \prec)$. □

The results of Thorems 3.4 and 3.5 can be extended to generalized structures.

Theorem 3.6. Let $G = (X, \diamond, \sqsubset)$ be a generalized interval order structure. Then there is a partial order $(T, \prec)$, two mappings $\varphi, \psi : X \to T$, and an acyclic relation $\lhd \subseteq T \times T$ such that for all $a, b \in X$, $\varphi(a) \prec \psi(a)$ and:

(a) $a \diamond b \iff \psi(a) \lhd \varphi(b) \vee \psi(b) \lhd \varphi(a)$
(b) $a <_G b \iff \psi(a) \prec \varphi(b)$, where $<_G = \diamond \cap \sqsubset$
(c) $a \sqsubset b \iff \varphi(a) \prec \psi(b) \vee \varphi(a) = \psi(b)$.

Furthermore, if G is initially finite then $(T, \prec)$ is also initially finite.

Proof.(Sketch) Let $\prec_t$ be any total extension of $<_G$. Define $R_t = \diamond \cap \prec_t$. Note that $\diamond = R_t \cup R_t^{-1}$ and $<_G \subseteq R_t$. Let $\prec$ be any relation that satisfies (b) and (c). Its existence follows from Theorem 3.4 or Theorem 3.5. First we set $\lhd$ equal to $\prec$. Let $(a, b) \in R_t - <_G = R_t - \sqsubset$. In such case we extend $\lhd$ by setting $\psi(a) \lhd \varphi(b)$. □

At this point we can prove the first half of our main result.

Theorem 3.7. The class of generalized interval order structures $\mathcal{GI}$ and the class of generalized initially finite interval order structures $\mathcal{GI}_{IF}$ are extension complete.

Proof.(Sketch) Let $G = (X, \diamond, \sqsubset)$ be a generalized interval order structure, and let Ω be the set of interval orders that extend G. Let $\lhd$ be a relation from Theorem 3.6, $\lhd^+$ be its transitive closure and let $(T, \prec_t)$ be any total extension of $(T, \lhd^+)$. We define $o = (X, \xrightarrow{o})$ as $a \xrightarrow{o} b \iff \psi(a) \prec_t \varphi(b)$. By Theorem 3.3, o is an interval order. Using Theorem 3.6 we can prove that o is an extension of G, i.e. $o \in \Omega \neq \emptyset$, so it suffices to show that the conditions (a) and (b) of Lemma 3.1. are satisfied. To prove this we need to use Theorem 3.6, Theorem 3.3 and Lemma 3.2. The reasoning for initially finite case is very similar. □

The second half of our main result requires a separate proof and the following lemma.

Lemma 3.3. Let $(Q, \prec)$ be an upper semi-lattice[7], and $\{S_r \mid r \in Q\}$, where $S_r = (X, <_r, \sqsubset_r)$, be a class of partial order structures such that $r_1 \preceq r_2 \Rightarrow (<_{r_1} \subseteq <_{r_2} \wedge \sqsubset_{r_1} \subseteq \sqsubset_{r_2})$. Then $S = (X, <, \sqsubset)$, where $< = \bigcup_{r \in Q} <_r$ and $\sqsubset = \bigcup_{r \in Q} \sqsubset_r$, is also a partial order structure.

Proof. (Sketch) We show that the axioms P1-P4 are satisfied. □

Theorem 3.8. The class of generalized partial order structures $\mathcal{GP}$ and the class of generalized initially finite partial order structures $\mathcal{GP}_{IF}$ are extension complete.

[7] A partial order $(X, <)$ is an upper semi-lattice if for any $x, y \in X$ there is $z \in X$ such that $x \leq z \wedge y \leq z$. In particular any total order is an upper semi-lattice.

Proof.(Sketch) Let $G = (X, \diamond, \sqsubset)$ be a generalized partial order structure, and let Ξ be the set of partial orders that extend G. Let R_t be a relation from the proof of Theorem 3.6, R_t^+ be its transitive closure. We can show that $o = (X, R_t^+)$ belongs to Ξ. By using Lemmas 3.2 and 3.3, we can prove that the conditions (a) and (b) of Lemma 3.1 are satisfied, which proves extension completeness of $\mathcal{GP}$. For $\mathcal{GP}_{IF}$ we proceed similarly. □

The proofs are generally much longer and more complex that it might appear from the sketches. Transfinite induction and the axiom of well-ordering are used.

By merging Theorem 3.1([7]) and Theorem 3.2 we obtain *generalisations of Szpilrajn Theorem for various types of partial orders, including the most general case.*

For the sake of putting all the results together we formulate them as a theorem.

Theorem 3.9. (Generalisations of Szpilrajn Theorem) The classes of generalized structures: $\mathcal{GT}$, $\mathcal{GS}$, $\mathcal{GI}$, $\mathcal{GP}$, $\mathcal{GT}_{IF}$, $\mathcal{GS}_{IF}$, $\mathcal{GI}_{IF}$, $\mathcal{GP}_{IF}$ are extension complete. □

It is intuitively obvious that in some cases both ordered structures and generalized ordered structures generate the same set of extensions. According to [10] such cases conform to paradigm π_3, but this property can also be expressed without introducing explicitly the concept of paradigm (in the sense of [10]).

Theorem 3.10. For every generalized structure $G = (X, \diamond, \sqsubset)$:
$\diamond = <_G \cup <_G^{-1} \iff ext(G) = ext(S_G)$.

Proof. (Sketch) Standard, by definition manipulation. □

4 Generalized Order Structures and Concurrent Histories

Below we discuss axiomatic representations of histories satisfying π_1, i.e. no restriction at all. Our main result is that *every (initially finite or not) total, stratified, interval, or partial generalized order structure corresponds in a natural way to a concurrent history.* It was argued in [10] that runs (observations) should be initially finite interval orders at most. The argument was based on the laws of Physics under the assumption that observers work alone, from purely mathematical viewpoint this assumption is not necessary. It was shown in [23] that teams of observers can see runs modelled by arbitrary posets, so below we do not impose the restrictions from [10].

Let $\mathcal{TO}$, $\mathcal{SO}$, $\mathcal{IO}$, $\mathcal{PO}$ denote the class of total, stratified, interval and partial orders respectively. Recall that $\mathcal{GT}$, $\mathcal{GS}$, $\mathcal{GI}$, $\mathcal{GP}$ denote the classes of generalized total, stratified, interval and partial order structures, respectively, while $\mathcal{T}$, $\mathcal{S}$, $\mathcal{I}$, $\mathcal{P}$ denote the classes of total, stratified, interval and partial order structures.

Let us devide the set $\{\mathcal{TO}, \mathcal{SO}, \mathcal{IO}, \mathcal{PO}, \mathcal{GT}, \mathcal{GS}, \mathcal{GI}, \mathcal{GP}, \mathcal{T}, \mathcal{S}, \mathcal{I}, \mathcal{P}\}$ into the following partitions: $\{\mathcal{TO}, \mathcal{GT}, \mathcal{T}\}$, $\{\mathcal{SO}, \mathcal{GS}, \mathcal{S}\}$, $\{\mathcal{IO}, \mathcal{GI}, \mathcal{I}\}$, $\{\mathcal{PO}, \mathcal{GP}, \mathcal{P}\}$, $\{\mathcal{TO}_{IF}, \mathcal{GT}_{IF}, \mathcal{T}_{IF}\}$, $\{\mathcal{SO}_{IF}, \mathcal{GS}_{IF}, \mathcal{S}_{IF}\}$, $\{\mathcal{IO}_{IF}, \mathcal{GI}_{IF}, \mathcal{I}_{IF}\}$, $\{\mathcal{PO}_{IF}, \mathcal{GP}_{IF}, \mathcal{P}_{IF}\}$.

We say that two classes are *related* if ther belong to the same partition.

For every set of partial orders Δ with a common domain X, define $G_\Delta = (X, \diamond_\Delta, \sqsubset_\Delta)$, and $S_\Delta = (X, <_\Delta, \sqsubset_\Delta)$ by

- $\diamond_\Delta = \bigcap_{o \in \Delta} (\overset{o}{\rightarrow} \cup \overset{o}{\leftarrow})$,
- $<_\Delta = \bigcap_{o \in \Delta} \overset{o}{\rightarrow}$,
- $\sqsubset_\Delta = \bigcap_{o \in \Delta} (\overset{o}{\rightarrow} \cup \overset{o}{\leftrightarrow})$.

The main result of this section can now be formulated as follows.

Theorem 4.1. Let Δ be a non-empty set of partial orders with common domain. Then:

$\Delta = \Delta^{cl} \subseteq \Sigma \iff (G_\Delta \in \Theta \wedge gext_\Theta(G_\Delta) = \Delta)$,

where Σ is a class of partial orders, Θ is a class of generalized order structures, and (Σ, Θ) are related.

Proof. (Sketch) Standard, using definitions and Theorem 3.9. □

Under the paradigm π_3 ordered structures and generalized ordered structures describe the same behaviour, which can be formally described as follows.

Theorem 4.2. Let Δ be a non-empty set of partial orders with common domain *conforming to paradigm* π_3, Σ be a class of partial orders, Ξ be a class of ordered structures, Θ be a class of generalized order structures, and (Σ, Ξ, Θ) are related. Then the following are equivalent:

(a) $\Delta = \Delta^{cl} \subseteq \Sigma$
(b) $S_\Delta \in \Xi \wedge ext_\Xi(S_\Delta) = \Delta$
(c) $G_\Delta \in \Theta \wedge gext_\Theta(G_\Delta) = \Delta$

Proof. (Sketch) It was proven in [10] that π_3 implies $\diamond_\Delta = <_\Delta \cup <_\Delta^{-1}$. Next we can use Theorem 3.10. □

5 Final Comment

In this paper, we refined the notion of generalized structures introduced in [7], and proved that various classes of generalized structures are extension complete. From purely mathematical point of view the results of this paper can be seen as a generalization of Szpilrajn Theorem [25], from total orders to stratified, interval and general partial orders.

An immediate application of the obtained results seems to be in the concurrent system synthesis problem area. We believe that the approach introduced in [22] could now, after employing the results of this paper, handle the cases like the program P_1 from the introduction.

The main result of this paper, Theorem 3.9, although highly motivated by concurrency theory, is entirely independent of any particular interpretation.

Acknowledgements

I would like to thanks especially one anonymous referee for detailed comments and very helpful suggestions.

References

1. U. Abraham, S. Ben-David, M. Magodor, On global-time and inter-process communication, in *Semantics for Concurrency*, Workshops in Computing, Springer 1990, 311-323.
2. P. Baldan, N. Busi, A. Corradini, M. Pinna, Functorial Concurrent Semantics for Petri Nets with Read and Inhibitor Arcs, *Theoretical Computer Science*, to appear.
3. E. Best, F. de Boer, C. Palamedissi, Partial Order and SOS Semantics for Linear Constraint Programs, *Lecture Notes in Computer Science* 1282, Springer 1997, 256-273.
4. E. Best, M. Koutny, Operational and Denotational Semantics for the Box Algebras, *Theoretical Computer Science* 211 (1999), 1-83.
5. P. Degano, U. Montanari, Concurrent histories; a basis for observing distributed systems, *J. Comput. System Sci.* 34 (1987), 422-467.
6. H. Gaifman, V. Pratt, Partial order models of concurrency and the computation of functions, *Proc. of LICS'87*, 72-85.
7. G. Guo, R. Janicki, Modelling Concurrent Behaviours by Commutativity and Weak Causality Relations, Proc. of AMAST'02, *Lecture Notes in Computer Science* 2422 (2002), 178-191.
8. P. C. Fishburn, Intransitive indifference with unequal indifference intervals, *J. Math. Psych.* 7 (1970) 144-149.
9. R. Janicki, M. Koutny, Invariants and Paradigms of Concurrency Theory. Proc. of PARLE'91, *Lecture Notes in Computer Science* 506 (1991), 59-74.
10. R. Janicki, M. Koutny, Structure of Concurrency, *Theoretical Computer Science* 112 (1993), 5-52.
11. R. Janicki, M. Koutny, Semantics of Inhibitor Nets, *Information and Computation*, 123, 1(1995), 1-16.
12. R. Janicki, M. Koutny, Fundamentals of modelling concurrency using discrete relational structures, *Acta Informatica*, 34 (1997), 367-388.
13. R. Janicki, M. Koutny, On Causality Semantics of Nets with Priorities, *Fundamenta Informaticae*, 38 (1999), 222-255.
14. G. Juhás, R. Lorentz, C. Neumair, Synthesis of Controlled Behaviour with Modules of Signal Nets, Proc. of ATPN'04, *Lecture Notes in Computer Science* 3099, Springer 2004, 233-257.
15. S. Katz, D. Peled, Defining conditional independence using collapses, in *Semantics for Concurrency*, Workshops in Computing, Springer 1990, 262-290.
16. H. Klaudel, F. Pommereau, A Class of Composable and Preemptible High-Level Petri Nets witn and Application to a Multi-Tasking System, *Fundamenta Informaticae*, 50 (2002), 33-55.

17. J. Kleijn, M. Koutny, Process Semantics of P/T-Nets with Inhibitor Arcs, *Lecture Notes in Computer Science* 1825, Springer 2000, 261-281.
18. L. Lamport, The mutual exclusion problem: Part I - a theory of interprocess communication; Part II - statements and solutions, *Journal of ACM* 33,2 (1986) 313-326.
19. L. Lamport, What It Means for a Concurrent Programm to Satisfy a Specification: Why No One Has Specified Priority, *Proc. 12th ACM Symp. on Programming Languages*, 1985, 78-83.
20. A. Mazurkiewicz, Trace Theory, *Lecture Notes in Computer Science* 225, Springer 1986, 297-324.
21. R. Milner, Operational and Algebraic Semantics of Concurrent Processes, in J. van Leuween (ed.), *Handbook of Theoretical Computer Science*,vol. 2, Elsevier 1993, 1201-1242.
22. M. Pietkiewicz-Koutny, The Synthesis Problem for Elementary Net Systems, *Fundamenta Informaticae* 40,2,3 (1999) 310-327.
23. G. Plotkin, V. Pratt, Teams can see pomsets, Unpublished Memo, Stanford University, 1990.
24. O. H. Roux, D. Lime, Time Petri Nets with Inhibitor Arcs. Formal Semantics and State Space Complexity, Proc. of ATPN'04, *Lecture Notes in Computer Science* 3099, Springer 2004, 370-390.
25. E. Szpilrajn, Sur l'extension de l'ordre partial, *Fundamenta Mathematicae* 16 (1930), 386-389.
26. W. Vogler, Timed Testing of Concurrent Systems, *Information and Computation* 121 (1995), 149-171.
27. W. Vogler, Partial Order Semantics and Inhibitor Arcs, *Lecture Notes in Computer Science* 1295, Springer 1997, 508-517.
28. R. Wollowski, J. Beister, Precise Petri Net Modelling of Critical Races in Asynchronous Arbiters and Synchronizers, *Proc.* 1^{st} *Workshop on Hardware Design and Petri Nets*, Lisbon 1998, 46-65.
29. R. Wollowski, J. Beister, Comprehensive Causal Specification of Asynchronous Controller and Arbiter Behaviour, in A. Yakovlev, L. Gomes, L. Lavagno (eds.) *Hardware Design and Petri Nets*, Kluwer 2000.

A Logical Characterization of Efficiency Preorders

Neelesh Korade[1] and S. Arun-Kumar[2,⋆]

[1] Persistent Systems Private Limited, "Bhageerath",
402, Senapati Bapat Road, Pune 411016, India
neelesh_korade@persistent.co.in
[2] Department of Computer Science and Engineering,
Indian Institute of Technology, Delhi,
Hauz Khas, New Delhi 110016, India
sak@cse.iitd.ernet.in

Abstract. In this paper we present logical characterizations of two preorders, within the framework of Hennessy-Milner Logics. The two preorders (loosely termed bisimulation-based efficiency preorders) are on processes represented as labelled transition systems. The characterizations are particularly interesting as they explore preorders lying between strong and weak bisimilarity, guided by a principle of containment which is explained in the Introduction. Even though the proofs of the characterizations use standard methods, there are various subtleties introduced by the nature of the preorders and the logical operators needed to characterize them. The authors have not previously encountered the use of such operators in such simple logics.

Keywords: Concurrency, transition systems, bisimulation, efficiency preorders, process efficiency, Hennessy-Milner Logic.

1 Introduction

In [5] a modal logic for reasoning about labelled transition systems was first defined which characterized the notions of simulations, strong and weak bisimulations. Subsequently the logic has been extended in various ways to include the modal μ-calculus and various behavioural equivalences and preordering relations on labelled transition systems. For a comprehensive account, the reader is referred to [12].

In all such formulations, a process is identified by the set of formulas of a logic that it satisfies. Given a behavioural equivalence relation on processes, a logical language L characterizes this equivalence relation precisely when two equivalent processes satisfy the same set of formulas of the logic.

Formally therefore, if $\mathbb{P}$ is a set of processes and L is a logic then we may identify each process $p \in \mathbb{P}$ with the set of formulas of L that it satisfies, i.e.

⋆ Corresponding author.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 99–112, 2005.
© Springer-Verlag Berlin Heidelberg 2005

$L(p) = \{\phi \in L \mid p \vDash \phi\}$ denotes the set of formulas of L that p satisfies. A behavioural equivalence $\cong$ on $\mathbb{P}$ is characterized by L whenever $p \cong q$ iff $L(p) = L(q)$. Similarly a behavioural preorder $\lessapprox$ on $\mathbb{P}$ is characterized by L whenever $p \lessapprox q$ iff $L(p) \subseteq L(q)$ or $p \lessapprox q$ iff $L(q) \subseteq L(p)$.

Van Glabeek [14] has used precisely such formulations to characterize the various behavioural relations on concrete processes. In several instances, it has been shown that one behavioural relation R is coarser than another S, by showing that a less expressive logic characterizes R or that a more expressive logic characterizes S.

Such a formulation may actually be traced back to [5]. In it the authors present two modal logics - the stronger or more expressive one (subsequently referred to in the literature as *Hennessy-Milner Logic* or *HML*) characterizing strong bisimulation equivalence and the weaker or less expressive one (referred to by Stirling [12] as *Observable Hennessy-Milner Logic* or *OHML*) characterizing observational equivalence. The paper [5] also showed the characterization of trace equivalence and simulation equivalence in terms of sub-logics of HML, thus establishing that both trace equivalence and the simulation equivalence are coarser than bisimulation. The authors actually showed that for image-finite processes, HML with finitary conjunctions captured strong bisimilarity whereas HML without negation characterized *simulation equivalence. Trace equivalence* was characterized by removing both negation and conjunction from the logic.

In this paper we provide such logical characterizations for behavioural relations that lie strictly between strong and weak bisimilarity by defining modified *Hennessy-Milner Logics* for the purpose. We use the following properties as guiding principles in the design of the logics.

1. For any behavioural preorder $\leq$ and logical language L characterizing $\leq$, $p \leq q$ iff $L(p) \subseteq L(q)$. It then follows that the kernel of the preorder is an equivalence relation and is characterized by equality on sets of satisfying formulae. That is, $p \leq q$ and $q \leq p$ if and only if $L(p) = L(q)$.
2. Given preorders $\leq_1$, and $\leq_2$ characterized respectively by logics L_1 and L_2, $\leq_1 \subset \leq_2$ iff $L_1 \prec L_2$ where $L_1 \prec L_2$ denotes that L_1 is more expressive than L_2 (and hence can allow for finer distinctions to be made).

While adhering to these principles, it may be pointed out that so far the spectrum of behavioural relations that lie strictly between strong and weak bisimilarity has not really been explored in the literature. This includes some preorders defined by Milner and others ([11], [10], [3], [2]). In this paper we provide a characterization of the preorders defined in [3] and [2]. As we will show, the characterization of these preorders requires reasoning about linear orders within logics whose expressive power lies strictly between HML and OHML.

Efficiency-based preorders have been of interest to various people since they were first introduced in [1]. Several other authors have worked on obtaining similar preorders within the framework of extensionality (see [13], [6], [7], [8], [4]). Some of these works are based on extent theories such as testing and bisimulation in process algebra and Petri nets.

The paper is organized as follows. In the next section we review HML and OHML and show the characterizations of strong and weak bisimilarity respectively. Section 3 presents the logical characterization of the elaboration preorder [3] using an appropriately modified OHML. In this section we also highlight the principal differences between the elaboration preorder and weak bisimilarity and hence the need for a more expressive logic than OHML to characterize elaboration. In section 4, we present the characterization of the efficiency preorder relation. Though the proofs in these sections are standard the operators introduced to characterize the two preorders are not standard. Section 5 is the conclusion and highlights further properties of the new modal operators that have been introduced to enrich OHML so as to characterize these preorders.

2 Modal Characterizations of Strong and Weak Bisimilarity

Here we review Hennessy-Milner Logic (HML) and Observable Hennessy-Milner Logic (OHML) and show how they characterize strong and weak bisimilarity respectively.

Let V be a set of *visible actions*, $\tau \notin V$ a distinguished *invisible action* and $Act = V \cup \{\tau\}$ the set of *actions*. A *labelled transition system (LTS)* is a 3-tuple $\langle \mathbb{P}, Act, \longrightarrow \rangle$, where $\mathbb{P}$ is a set of *process states* or *processes* and $\longrightarrow \subseteq \mathbb{P} \times Act \times \mathbb{P}$ is the *transition relation*. We use the notation $p \xrightarrow{a} q$ to denote $(p, a, q) \in \longrightarrow$ and refer to q as a *strong a-derivative* of p.

2.1 Strong Bisimulation

Definition 1. *A binary relation $R \subseteq \mathbb{P} \times \mathbb{P}$ is a* `strong simulation(SS)` *if for every $\langle p, q \rangle \in R$ and $a \in Act$:*

$$p \xrightarrow{a} p' \implies \exists q' : q \xrightarrow{a} q' \wedge p' R q'$$

It is a `strong bisimulation (SB)` *if both R and R^{-1} are strong simulations. We write $p \sim q$ if there exists a strong bisimulation R such that pRq. The relation $\sim$ is called* **strong bisimilarity**.

The relation $\sim$ is itself a strong bisimulation, and in fact, the largest one. In [9] the author gives a comprehensive account of the modal logic (HML) that characterizes strong bisimilarity. It is defined as follows.

Definition 2. *The class L_{SB} of strong bisimulation formulas over Act is given by the following grammar (I is an indexing set, not necessarily finite).*

$$\varphi ::= \ < a > \varphi \mid \bigwedge_{i \in I} \varphi_i \mid \neg \varphi$$

$tt \in L_{SB}$ where $tt \equiv \bigwedge_{i \in \emptyset} \varphi_i$. Similarly, $ff \in L_{SB}$ where $ff \equiv \neg tt$.

Definition 3. *The satisfaction relation* $\vDash \subseteq \mathbb{P} \times L_{SB}$ *is defined recursively as follows:*

- $p \vDash tt$ *for all* $p \in \mathbb{P}$.
- $p \vDash < a > \varphi$ *for* $a \in Act$ *if* $\exists p' \in \mathbb{P} : p \xrightarrow{a} p'$ *and* $p' \vDash \varphi$.
- $p \vDash \bigwedge_{i \in I} \varphi_i$ *if* $p \vDash \varphi_i$ *for all* $i \in I$.
- $p \vDash \neg\varphi$ *if* $p \nvDash \varphi$.

The set $SB(p)$ is defined as $SB(p) = \{\varphi \in L_{SB} \mid p \vDash \varphi\}$. We write $p \sqsubseteq_{SB} q$ iff $SB(p) \subseteq SB(q)$ and we write $p =_{SB} q$ iff $SB(p) = SB(q)$. The negation operator collapses the preorder $\sqsubseteq_{SB}$ to $=_{SB}$ as the following proposition shows.

Proposition 1 (Van Glabeek [14]). $p \sqsubseteq_{SB} q \iff p =_{SB} q$.

Proof. If $\varphi \in SB(q) - SB(p)$ then $\neg\varphi \in SB(p) - SB(q)$. Hence, $p \sqsubseteq_{SB} q \iff p =_{SB} q$. □

Theorem 1 (Van Glabeek [14]). $p \sim q \iff p =_{SB} q$.

Proof. ($\Longrightarrow$) By induction on the structure of φ. Since $p \sim q$, there exists a *strong bisimulation* R such that pRq.

- Let $p \vDash < a > \varphi$. Then there exists a $p' \in \mathbb{P}$ with $p \xrightarrow{a} p'$ and $p' \vDash \varphi$. Since pRq, there must be a $q' \in \mathbb{P}$ with $q \xrightarrow{a} q'$ and $p'Rq'$. So by induction $q' \vDash \varphi$ and hence $q \vDash < a > \varphi$.
  By symmetry, one also obtains $q \vDash < a > \varphi \Longrightarrow p \vDash < a > \varphi$.
- $p \vDash \bigwedge_{i \in I} \varphi_i \iff \forall i \in I\,(p \vDash \varphi_i) \overset{induction}{\iff} \forall i \in I\,(q \vDash \varphi_i) \iff q \vDash \bigwedge_{i \in I} \varphi_i$.
- $p \vDash \neg\varphi \iff p \nvDash \varphi \overset{induction}{\iff} q \nvDash \varphi \iff q \vDash \neg\varphi$.

($\Longleftarrow$) To prove that $p =_{SB} q \Longrightarrow p \sim q$, it suffices to establish that $=_{SB}$ is a strong bisimulation. If however, we show that $\sqsubseteq_{SB}$ is a strong simulation, then proposition 1 implies that $=_{SB} \;=\; \sqsubseteq_{SB} \;=\; \sqsubseteq_{SB}^{-1}$ is a strong bisimulation. We proceed to show that $\sqsubseteq_{SB}$ is a strong simulation.

Suppose, $p \sqsubseteq_{SB} q$ and $p \xrightarrow{a} p'$. Then $p \vDash < a > tt$ and $p \sqsubseteq_{SB} q$ implies q has at least one strong a-derivative. We have to show that $\exists q' \in \mathbb{P}$ with $q \xrightarrow{a} q'$ and $p' \sqsubseteq_{SB} q'$. Let

$$Q = \{q'' \in \mathbb{P} \mid q \xrightarrow{a} q'' \wedge p' \not\sqsubseteq_{SB} q''\}$$

For every $q'' \in Q$ there is a formula $\varphi_{q''} \in SB(p') - SB(q'')$. Now $< a > \bigwedge_{q'' \in Q} \varphi_{q''} \in SB(p) \subseteq SB(q)$. Therefore there must be a $q' \in \mathbb{P}$ with $q \xrightarrow{a} q'$ and $q' \vDash \bigwedge_{q'' \in Q} \varphi_{q''}$, which implies $q' \notin Q$. □

With negation in the logic, we may define the duals of the operators in L_{SB} as derived ones.

$$[a]\varphi \equiv \neg < a > \neg\varphi \qquad\qquad \bigvee_{i \in I} \varphi_i \equiv \neg \bigwedge_{i \in I} \neg\varphi_i$$

We may then define a negation-free language $L_{SB'}$.

Definition 4. *The class $L_{SB'}$ of negation-free strong bisimulation formulas over Act is given by the following grammar*

$$\varphi ::= <a> \varphi \mid \bigwedge_{i \in I} \varphi_i \mid [a]\varphi \mid \bigvee_{i \in I} \varphi_i$$

with $tt \equiv \bigwedge_{i \in \emptyset} \varphi_i$ *and* $ff \equiv \bigvee_{i \in \emptyset} \varphi_i$.

Definition 5. *The satisfaction relation $\models \subseteq \mathbb{P} \times L_{SB'}$ is defined recursively as in the case of L_{SB} with the clause for negation being replaced by the following clauses.*

- $p \models ff$ *for no* $p \in \mathbb{P}$.
- $p \models [a]\varphi$ *for* $a \in Act$ *if* $\forall\, p' \in \mathbb{P} : p \xrightarrow{a} p' \implies p' \models \varphi$.
- $p \models \bigvee_{i \in I} \varphi_i$ *if* $p \models \varphi_i$ *for some* $i \in I$.

It is clear that $\neg\neg\varphi$ is equivalent to φ. It may be shown (refer [12]) that L_{SB} and $L_{SB'}$ are expressively equivalent and characterize the same equivalence.

2.2 Weak Bisimulation

We may define several other derived operators. Some of the relevant ones are given below.

$$<a>^0 \varphi \equiv \varphi \qquad <a>^{m+1} \varphi \equiv <a><a>^m \varphi, m \in \mathbb{N}$$
$$\ll\gg \varphi \equiv \bigvee_{m \geq 0} <\tau>^m \varphi \qquad \ll a \gg \varphi \equiv \ll\gg<a>\ll\gg \varphi$$
$$[[\ ]]\varphi \equiv \neg \ll\gg \neg\varphi \qquad [[a]]\varphi \equiv [[\ ]][a][[\ ]]\varphi$$

The last four operators correspond to the weak transition relation $\Longrightarrow$ which is the smallest relation on processes such that

- $p \Longrightarrow p$ for all processes p, and
- $p \xrightarrow{\tau} q$ and $q \Longrightarrow r$ implies $p \Longrightarrow r$

Further for any $a \in Act$,

- $p \overset{a}{\Longrightarrow} q$ if $p \Longrightarrow \xrightarrow{a} \Longrightarrow q$, and q is called a *weak a-derivative* or simply an *a-derivative* of p.
- $p \overset{\hat{a}}{\Longrightarrow} q$ denotes $p \Longrightarrow q$ if $a = \tau$ and $p \overset{a}{\Longrightarrow} q$ otherwise.
- $\ll \hat{a} \gg$ denotes $\ll\gg$ if $a = \tau$ and $\ll a \gg$ otherwise.

Definition 6. *A binary relation $R \subseteq \mathbb{P} \times \mathbb{P}$ is a* `weak simulation` *if for every $\langle p, q \rangle \in R$ and $a \in Act$,*

$$p \xrightarrow{a} p' \implies \exists q' : q \overset{\hat{a}}{\Longrightarrow} q' \wedge p' R q'$$

R is a **weak bisimulation (WB)** *if both R and R^{-1} are weak simulations. We write $p \approx q$ if there exists a weak bisimulation R such that pRq.*

Definition 7. *The class L_{WB} of weak bisimulation formulae over Act is defined by the following BNF.*

$$\varphi ::= \ll\alpha\gg\varphi \mid \bigwedge_{i\in I}\varphi_i \mid \neg\varphi$$

where $\alpha \in V$.

Note that in the light of our discussion on derived operators, L_{WB} is entirely contained in L_{SB}. It is also less expressive in the sense that it cannot express properties relating to the number of invisible moves a process might be able to make.

Definition 8. *The satisfaction relation $\vDash \subseteq \mathbb{P} \times L_{WB}$ is defined recursively by:*

- $p \vDash tt$ *for all* $p \in \mathbb{P}$.
- $p \vDash \ll\alpha\gg\varphi$ *for* $\alpha \in V$ *if* $\exists p' : p \stackrel{\alpha}{\Longrightarrow} p' \wedge p' \vDash \varphi$
- $p \vDash \bigwedge_{i\in I}\varphi_i$ *if* $p \vDash \varphi_i$ *for all* $i \in I$.
- $p \vDash \neg\varphi$ *if* $p \nvDash \varphi$.

The set of all formulae that a process p satisfies is defined as $WB(p) = \{\varphi \in L_{WB} \mid p \vDash \varphi\}$. We write $p \sqsubseteq_{WB} q$ iff $WB(p) \subseteq WB(q)$ and we write $p =_{WB} q$ iff $WB(p) = WB(q)$. The proofs that $p \sqsubseteq_{WB} q$ implies $p =_{WB} q$ and that $p \approx q$ iff $p =_{WB} q$ (i.e. that L_{WB} characterizes weak bisimilarity) are similar to the corresponding proofs for strong bisimilarity and hence are omitted. It is also possible to define a negation-free logic $L_{WB'}$ (analogous to $L_{SB'}$) which characterizes weak bisimilarity.

3 Elaboration

In this section we give a logical characterization of the elaboration preorder defined in [3]. This logic builds upon L_{WB} and exhibits *controlled* use of counting.

Definition 9. *A binary relation $R \subseteq \mathbb{P} \times \mathbb{P}$ is an* `elaboration` *iff for every $\langle p, q\rangle \in R$ the following conditions hold for every action $a \in Act$.*

$$p \xrightarrow{a} p' \Longrightarrow \exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge p' R q' \tag{1}$$

$$q \xrightarrow{a} q' \Longrightarrow \exists p' : p \stackrel{a}{\Longrightarrow} p' \wedge p' R q' \tag{2}$$

We write $p \lessapprox q$ if there exists an elaboration R such that pRq.

Loosely speaking, if $p \lessapprox q$, then $p \approx q$ and for every execution sequence of p, there exists a possibly shorter execution sequence of q which exhibits the same visible behaviour. In [3] the authors illustrate this feature with a small example. They also show in the equational axiomatization of the corresponding precongruence (and congruence) for finite CCS processes, that it differs from that of observational congruence in just one equation. Whereas observational congruence satisfies the equation $a.\tau.p = a.p$, in the case of elaboration, $a.\tau.p \lessapprox a.p$ but

the converse $a.p \lessapprox a.\tau.p$ does not always hold. In that sense the precongruence is closer to observational congruence than any other relation defined so far in the literature.

It is easy to show from the above definition that every *elaboration* is a weak bisimulation. In particular, the only difference from that of weak bisimulation is in clause (2), where the $\stackrel{\hat{a}}{\Longrightarrow}$ is replaced by $\stackrel{a}{\Longrightarrow}$. We state without proof, a result (see [3]) that we use in the proof of lemma 4.

Lemma 1. *If $R \subseteq \mathbb{P} \times \mathbb{P}$ is an elaboration then for every $\langle p, q \rangle \in R$ the following conditions hold for every action $a \in Act$.*

$$p \stackrel{a}{\Longrightarrow} p' \implies \exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge p' R q' \tag{3}$$

$$q \stackrel{a}{\Longrightarrow} q' \implies \exists p' : p \stackrel{a}{\Longrightarrow} p' \wedge p' R q' \tag{4}$$

From lemma 1 it is clear that any HML-style characterization that is in accordance with the principles laid out in section 1 would enrich observable HML slightly in order to characterize the preorder. We define the following two-level grammar, whose lower level (defined by the meta-variable φ) is L_{WB}. The higher level (defined by the meta-variable π) expresses a certain weak form of counting. It introduces a new operator ϵ^k, (where k is a positive integer) but does not permit negation to precede any occurrence of ϵ^k. The operator $\ll \hat{a} \gg$ excludes $\ll \tau \gg$. Where counting of τ actions is not important, the lack of negation is compensated by the derived operators $[[\ldots]]$ and $\bigvee$.

Definition 10. *The class L_E of Elaboration formulae over Act is given by the following two-level grammar, where $\alpha \in V$ and $a \in A$.*

$$\varphi ::= \ \ll \alpha \gg \varphi \mid \textstyle\bigwedge_{i \in I} \varphi_i \mid \neg \varphi$$

$$\pi ::= \ \varphi \mid \ \ll \hat{a} \gg \pi \mid \epsilon^k \pi \mid [[a]]\pi \mid \textstyle\bigwedge_{i \in I} \pi_i \mid \bigvee_{i \in I} \pi_i$$

Definition 11. *The satisfaction relation $\vDash \subseteq \ \mathbb{P} \times L_E$ is defined recursively in a manner that should be obvious by now for all operators drawn from L_{WB}. So we restrict ourselves to the definitions of the operators $\ll \hat{a} \gg$, ϵ^k and $[[a]]$ respectively.*

- $p \vDash \ll \hat{a} \gg \pi$ *for* $a \in Act$, *if* $\exists p' \in \mathbb{P} : p \stackrel{\hat{a}}{\Longrightarrow} p' \wedge p' \vDash \pi$.
- $p \vDash \epsilon^k \pi$ *for* $k > 0$, *if* $\forall p' \in \mathbb{P} : p \stackrel{\tau^j}{\longrightarrow} p' \wedge p' \vDash \pi \implies j < k$
- $p \vDash [[a]]\pi$ *for* $a \in Act$ *if for all* $p' \in \mathbb{P} : p \stackrel{a}{\Longrightarrow} p' \implies p' \vDash \pi$.

$p \stackrel{\tau^j}{\longrightarrow} p'$ denotes that p may evolve to p' after performing j consecutive invisible actions. $E(p) = \{\pi \in L_E \mid p \vDash \pi\}$ and $E_{WB}(p) = \{\varphi \in L_{WB} \mid p \vDash \varphi\}$. We write $p \sqsubseteq_E q$ iff $E(p) \subseteq E(q)$ and $p =_E q$ iff $E(p) = E(q)$. Similarly, $p \sqsubseteq_{E_{WB}} q$ iff $E_{WB}(p) \subseteq E_{WB}(q)$ and $p =_{E_{WB}} q$ iff $E_{WB}(p) = E_{WB}(q)$.

To gain a deeper understanding, we give below some examples that illustrate the expressive power of L_E.

Example 1.

- The statement "*p is stable*" i.e. p cannot perform a τ action, may be expresed as $\epsilon^1 tt$. More generally, $p \vDash \epsilon^k tt$ if and only if p may perform *no more than* $(k-1)$ consecutive τ actions.
- The statement "*p converges*" i.e. p cannot perform an infinite sequence of τ actions, is expressed as $\bigvee_{k>0} \epsilon^k tt$.
- The statement "*p diverges*" is however not expressible in the language since p diverges iff $p \not\vDash \bigvee_{k>0} \epsilon^k tt$.
- The statement $p \vDash \epsilon^k ff$ means that "p can perform τ^m for all $m \geq k$". However, this statement does not necessarily imply that p diverges, unless p is also finitely branching.
- $\epsilon^j \pi$ logically implies $\epsilon^k \pi$ for all $0 < j < k$.
- For any $j > 0$, $\bigwedge_{k \geq j} \epsilon^k \pi$ is logically equivalent to $\epsilon^j \pi$.
- Statements such as "*p can do at least two consecutive τ actions*" are not expressible since there is no operator which can express lower bounds on the number of consecutive τ actions[1].

We now proceed to prove the characterization theorem. We begin with the following lemma which is clearly implied by the fact that every elaboration is a weak bisimulation and that $\sqsubseteq_{E_{WB}}$ and $=_{E_{WB}}$ coincide.

Lemma 2. *$p \lessapprox q \implies p =_{E_{WB}} q$ and hence $p \lessapprox q \implies p \sqsubseteq_{E_{WB}} q$.* □

Theorem 2. The characterization. *$p \lessapprox q$ iff $p \sqsubseteq_E q$.* □

We split the proof of theorem 2 into two parts.

Lemma 3. *$p \sqsubseteq_E q$ implies $p \lessapprox q$.*

Proof. We show that $\sqsubseteq_E$ is an elaboration. Let $E(p) \subseteq E(q)$. We need to prove both parts (1) and (2) of definition 9.

Part (1). Consider $p \xrightarrow{a} p'$ where $a \in Act$. Then $p \vDash \ll \hat{a} \gg \pi$ for any $\pi \in E(p')$ and $q \vDash \ll \hat{a} \gg \pi$ which implies $\exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge q' \vDash \pi$. We need to show that $\exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge E(p') \subseteq E(q')$.

Let

$$Q = \{q'' \mid q \stackrel{\hat{a}}{\Longrightarrow} q'' \wedge E(p') \not\subseteq E(q'')\}$$

Then for each $q'' \in Q$, there exists a formula $\pi_{q''} \in E(p') - E(q'')$. This implies $p' \vDash \bigwedge_{q'' \in Q} \pi_{q''}$. Hence $p \vDash \pi$ where $\pi = \ll \hat{a} \gg \bigwedge_{q'' \in Q} \pi_{q''}$. That is, $\pi \in E(p) \subseteq E(q)$. Hence $\exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge q' \vDash \bigwedge_{q'' \in Q} \pi_{q''}$ and $q' \notin Q$, which shows that $E(p') \subseteq E(q')$, which needed to be proved.

[1] To be able to express such statements, would require either the power of negation or the dual of ϵ^k. But allowing such operators would make the logic equivalent to L_{SB}, something we wish to avoid.

Part (2). Suppose $q \xrightarrow{a} q'$. We claim $\exists p' : p \stackrel{a}{\Longrightarrow} p'$. Suppose not. If $a \in V$ then for some $\varphi \in E_{WB}(q')$, $q \vDash \ll a \gg \varphi$ and $p \nvDash \ll a \gg \varphi$. This implies $p \vDash \neg \ll a \gg \varphi$ and $q \nvDash \neg \ll a \gg \varphi$ which contradicts $E_{WB}(p) \subseteq E_{WB}(q)$. On the other hand, if $a = \tau$, then $p \stackrel{\tau}{\Longrightarrow}$ implies $p \vDash \epsilon^1 tt$. But since $q \xrightarrow{\tau} q'$, $q \nvDash \epsilon^1 tt$ which contradicts $E(p) \subseteq E(q)$.

Now, we need to show that $\exists p' : p \stackrel{a}{\Longrightarrow} p' \wedge E(p') \subseteq E(q')$. Suppose there is no such p'. Then $\forall p' : p \stackrel{a}{\Longrightarrow} p' \implies E(p') \nsubseteq E(q')$. Then for each weak a-derivative p'' of p, there exists a $\pi_{p''} \in E(p'') - E(q')$. Choose one such for each weak a-derivative p'' of p and collect them in a set Π. Then we have $p \vDash [[a]](\bigvee \Pi)$, but $q \nvDash [[a]](\bigvee \Pi)$, which contradicts $E(p) \subseteq E(q)$. Hence $\exists p' : p \stackrel{a}{\Longrightarrow} p' \wedge E(p') \subseteq E(q')$. □

Lemma 4. *$p \lessapprox q$ implies $p \sqsubseteq_E q$ i.e. $E(p) \subseteq E(q)$.*

Proof. We need to show $p \lessapprox q \Longrightarrow (p \vDash \pi \Longrightarrow q \vDash \pi)$. We prove this by induction on the structure of π

- $p \vDash \varphi$. Then $q \vDash \varphi$ follows from lemma 2.
- $p \vDash \ll \hat{a} \gg \pi$, $a \in Act$. Then $\exists p' : p \stackrel{\hat{a}}{\Longrightarrow} p' \wedge p' \vDash \pi$ and since $p \lessapprox q$, $\exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge p' \lessapprox q'$. Therefore, by induction hypothesis, $q' \vDash \pi \Longrightarrow q \vDash \ll \hat{a} \gg \pi$.
- $p \vDash \epsilon^k \pi$. Then $\forall p' : p \xrightarrow{\tau^j} p' \wedge p' \vDash \pi \Longrightarrow j < k$. Since $p \lessapprox q$, we have, for every such p' and j there exist q' and m respectively, such that $q \xrightarrow{\tau^m} q' \wedge p' \lessapprox q'$. Therefore by the induction hypothesis, $q' \vDash \pi$. We claim that, $m \geq k$ is impossible. Suppose not; then $q \xrightarrow{\tau^m} q' \wedge p' \lessapprox q'$ where $m \geq k$, for some q' and m. Then $\exists p'' : p \xrightarrow{\tau^j} p'' \wedge p'' \lessapprox q'$ for some $j \geq m \geq k$. But then $p \nvDash \epsilon^k \pi$ which is a contradiction. Hence $q \vDash \epsilon^k \pi$.
- $p \vDash [[a]]\pi$. Then $\forall p' : p \stackrel{a}{\Longrightarrow} p' \Longrightarrow p' \vDash \pi$. Since $p \lessapprox q$, we have, for every such p', $\exists q' : q \stackrel{\hat{a}}{\Longrightarrow} q' \wedge p' \lessapprox q'$. By the induction hypothesis $E(p') \subseteq E(q')$ and $q' \vDash \pi$. We now claim that there does not exist any q' such that $q \stackrel{a}{\Longrightarrow} q'$ and $q' \nvDash \pi$. Suppose the claim is false. Then we have $\exists q' : q \stackrel{a}{\Longrightarrow} q' \wedge q' \nvDash \pi$. This implies (by lemma 1) $\exists p' : p \stackrel{a}{\Longrightarrow} p' \wedge p' \lessapprox q' \nvDash \pi$. Again by induction hypothesis $E(p') \subseteq E(q')$ and hence $p' \nvDash \pi$. But this contradicts the assumption $p \vDash [[a]]\pi$. Hence the claim is false.
- $p \vDash \bigwedge_{i \in I} \pi_i$. Then $p \vDash \pi_i$ for all $i \in I$ and by the induction hypothesis, $q \vDash \pi_i$ for all $i \in I$ which implies $q \vDash \bigwedge_{i \in I} \pi_i$
- $p \vDash \bigvee_{i \in I} \pi_i$. Then $p \vDash \pi_i$ for at least one $i \in I$ and by induction hypothesis it follows that $q \vDash \bigvee_{i \in I} \pi_i$. □

The above proof has been presented in detail to highlight the steps for the operators at the higher level, especially the operator ϵ^k. Note that the proof would not go through if we allowed $\ll \tau \gg$ in the logic.

4 Efficiency Prebisimulation

As we did for *elaboration* in the previous section, we now present a logical characterization for *efficiency prebisimulation* [2] using a logic which is more expressive than L_E but not as expressive as L_{SB}.

The following formulation [2] gives a simple definition of the efficiency preorder. Closely related preorders have been defined independently by Milner in [10] and [11], though neither their algebraic nor logical characterizations have been presented.

Definition 12. *A binary relation $R \subseteq P \times P$ is an* **efficiency prebisimulation (EP)** *iff for every $\langle p, q\rangle \in R$, $\alpha \in V$, $a \in Act$ the following conditions are satisfied.*

$$p \xrightarrow{\alpha} p' \Longrightarrow \exists q' : q \xrightarrow{\alpha} q' \wedge p' R q' \tag{5}$$

$$p \xrightarrow{\tau} p' \Longrightarrow p' R q \vee (\exists q' : q \xrightarrow{\tau} q' \wedge p' R q') \tag{6}$$

$$q \xrightarrow{a} q' \Longrightarrow \exists p' : p \stackrel{a}{\Longrightarrow} p' \wedge p' R q' \tag{7}$$

We write $p \lesssim q$ if there exists an efficiency prebisimulation R such that pRq.

Intuitively, $p \lesssim q$ means that for every execution sequence that p may perform, it is possible to find a possibly shorter sequence that q may perform with the same visible content, and conversely, for any sequence that q may perform it is possible to find a possibly longer sequence that p may perform with the same visible content. In general, both the preorders $\lesssim$ and $\lessapprox$ represent comparisons between observationally equivalent processes and order them differently on the amount of internal computation they may perform. We refer to both preorders as efficiency-based preorders, but $\lesssim$ having been christened "efficiency preorder" earlier we continue to refer to it by the same name. We refer the reader to [2] and [1] for intuitively appealing examples and for the axiomatization of finite CCS processes.

In [3] it has been shown that $\sim \subset \lesssim \subset \lessapprox \subset \approx$ (where all the containments are strict). This suggests that we require a logic that is more expressive than L_E, but less than L_{SB}. This logic is very similar in structure to the logic for *elaboration* and is again defined in two levels with the lower level of weak formulae being the same.

Definition 13. *The class L_{EP} of efficiency prebisimulation formulae over Act is given by the following grammar. As before, we use α to denote a visible action and a to denote any action.*

$$\varphi ::= \ll\alpha\gg\varphi \mid \bigwedge_{i\in I}\varphi_i \mid \neg\varphi$$

$$\pi ::= \varphi \mid <\alpha>\pi \mid (\tau)\pi \mid \epsilon^k\pi \mid [[a]]\pi \mid \bigwedge_{i\in I}\pi_i \mid \bigvee_{i\in I}\pi_i$$

Note that the index set I may be infinite. Also negation ($\neg$) as in the case of L_E, is available only for *weak formulae*. The language L_{EP} is the set of all formulae π and the language L_{WB} is the set of all weak formulae φ. Note that $L_{WB} \subseteq L_{EP}$.

Definition 14. *The satisfaction relation* $\models \subseteq \mathbb{P} \times L_{EP}$ *is defined recursively. As before, we omit those clauses which have been previously presented and restrict ourselves to defining the clauses for the new operators.*

- $p \models <\alpha> \pi$ *for* $\alpha \in V$ *if for some* $p' \in \mathbb{P} : p \xrightarrow{\alpha} p'$ *and* $p' \models \pi$
- $p \models (\tau)\pi$ *if* $p \models \pi$ *or for some* $p' \in \mathbb{P} : p \xrightarrow{\tau} p'$ *and* $p' \models \pi$

$EP(p) = \{\pi \in L_{EP} \mid p \models \pi\}$ and $EP_{WB}(p) = \{\varphi \in L_{WB} \mid p \models \varphi\}$. Further, $p \sqsubseteq_{EP} q$ iff $EP(p) \subseteq EP(q)$, and $p =_{EP} q$ iff $EP(p) = EP(q)$. As before, $p \sqsubseteq_{EP_{WB}} q$ iff $EP_{WB}(p) \subseteq EP_{WB}(q)$ and $p =_{EP_{WB}} q$ iff $EP_{WB}(p) = EP_{WB}(q)$.

As for the expressiveness of this language, notice that

- we allow the "strong possibility" modality ("$<\alpha>$") from L_{SB}, but neither negation nor "strong necessity" ("$[\alpha]$").
- The operator "$<\tau>$" is replaced by the weaker prefix operator "(τ)" .
- The operator ϵ^k in the formula $\epsilon^k\pi$ excludes the possibility of a process being able to perform more than $k-1$ initial τ actions and reaching a state satisfying π.
- On the other hand, p satisfying the formula $(\tau)^k\pi$, $k > 0$ (obtained by prefixing π by k occurrences of (τ)) asserts the existence of a τ^j derivative of p (for some j, $0 \leq j \leq k$) which satisfies π.

There are no formulae in L_{EP} equivalent to the HML formula $\neg <\tau><\tau> \phi$. Even a statement such as "*p has a strong τ^2-derivative that satisfies π*" can only be inferred if it is known that $p \not\models \pi$, $p \not\models (\tau)\pi$ and $p \models (\tau)^2\pi$.

We now proceed to give a proof outline of the characterization theorem. We begin with the following lemma which is clearly implied by the fact that every efficiency prebisimulation is a weak bisimulation and that $\sqsubseteq_{EP_{WB}}$ and $=_{EP_{WB}}$ coincide.

Lemma 5. $p \lesssim q \implies p =_{EP_{WB}} q$ *and hence* $p \lesssim q \implies p \sqsubseteq_{EP_{WB}} q$. □

Theorem 3. The characterization. $p \lesssim q$ *iff* $p \sqsubseteq_{EP} q$. □

We prove this theorem in two parts. We omit most of the routine details and concentrate on only the operators that have been newly introduced.

Lemma 6. $p \sqsubseteq_{EP} q$ *implies* $p \lesssim q$.

Proof. We show that $\sqsubseteq_{EP}$ is an efficiency prebisimulation. We prove the various parts viz. (5), (6) and (7) of definition (12), assuming $EP(p) \subseteq EP(q)$.

Part (5). Suppose $p \xrightarrow{\alpha} p'$, $\alpha \in V$. Then $p \models <\alpha> \pi$ for each $\pi \in EP(p')$. Clearly $q \models <\alpha> \pi$ and there exists q' such that $q \xrightarrow{\alpha} q'$ and $q' \models \pi$. We need

to show $\exists q' : q \xrightarrow{\alpha} q' \wedge EP(p') \subseteq EP(q')$. We use an argument that has now become routine viz. consider the set $Q = \{q'' | q \xrightarrow{\alpha} q'' \wedge EP(p') \not\subseteq EP(q'')\}$ and proceed as before.

Part (6). Suppose $p \xrightarrow{\tau} p'$, then for each $\pi \in EP(p')$ we have $p \vDash (\tau)\pi$ and $q \vDash (\tau)\pi$. Then if $q \nvDash \pi$ there must exist a q' such that $q \xrightarrow{\tau} q'$ and $q' \vDash \pi$. It then suffices to show that $\exists q' : q \xrightarrow{\tau} q' \wedge EP(p') \subseteq EP(q')$. We may show this in a manner similar to that of Part 5.

Part (7). The proof of this part is similar to Part (2) of the proof of lemma 3. □

Lemma 7. *$p \lesssim q$ implies $p \sqsubseteq_{EP} q$.*

Proof. Assume $p \lesssim q$. We again use induction on the structure of formulae. For any formula ϕ from EP_{WB}, lemma 5 assures us that $p \vDash \phi$ implies $q \vDash \phi$. Of the rest of the cases from the language $L_{EP} - L_{EP_{WB}}$ we consider only the following:

- $p \vDash <\alpha> \pi$, $\alpha \in V$. Then for some p', we have $p \xrightarrow{\alpha} p'$ and $p' \vDash \pi$. Hence $\exists q' : q \xrightarrow{\alpha} q' \wedge p' \lesssim q'$. By the induction hypothesis, $EP(p') \subseteq EP(q')$ and so we get $q \vDash <\alpha> \pi$.
- $p \vDash (\tau)\pi$. If $p \vDash \pi$ then by the induction hypothesis $q \vDash \pi$ and so it follows that $q \vDash (\tau)\pi$. On the other hand, if $p \nvDash \pi$, then $\exists p' : p \xrightarrow{\tau} p' \wedge p' \vDash \pi$. From $p \lesssim q$ we have, either $p' \lesssim q$ or $\exists q' : q \xrightarrow{\tau} q' \wedge p' \lesssim q'$. If $p' \lesssim q$ then $p' \vDash \pi$ implies $q \vDash \pi$ by the induction hypothesis. If $p' \not\lesssim q$ then $p' \vDash \pi$ implies $q' \vDash \pi$ by the induction hypothesis and it follows that $q \vDash (\tau)\pi$.
- $p \vDash \epsilon^k \pi$, $k > 0$. This is again proved in a manner analogous to the corresponding proof for elaboration. □

5 Conclusion

In the foregoing sections we have introduced two new operators into Hennessy-Milner Logic (HML) viz. ϵ^k and (τ). We have shown and characterized preorders lying strictly between strong and weak bisimilarity using versions of HML and OHML with these new operators.

At the outset, we would first of all like to be convinced that their introduction into L_{SB} does not in any way alter the expressive power of HML. It is easy to see from their semantics that these operators enjoy the following equivalences. For any formula π in L_E or L_{EP}, let $\tilde{\pi}$ denote the equivalent formula in L_{SB}. We then have

$$(\tau)\pi \equiv \tilde{\pi} \vee \ <\tau> \tilde{\pi} \tag{8}$$

$$\epsilon^k \pi \equiv \bigwedge_{j \geq k} [\tau]^j \neg \tilde{\pi} \tag{9}$$

While (8) is obvious from its semantics, it is fairly easy to derive (9) by noting that $p \nvDash \epsilon^k \pi$ precisely when $\exists p' : p \xrightarrow{\tau^j} p' \wedge j \geq k \wedge \ (p' \vDash \pi)$.

These operators also satisfy some other properties. For example, any formula $\pi \in L_{EP}$ logically implies $(\tau)\pi$. And for any $k > 0$ we have $(\tau)^k\pi \equiv \bigvee_{j \leq k}(\tau)^j\pi$. Further, $(\tau)^k\pi \implies (\tau)^{k+1}\pi$. Similarly, in both L_E and L_{EP}, for any $k > 0$, we have $\epsilon^k\pi \implies \epsilon^{k+1}\pi$. For any set S of positive integers, let $inf(S)$ and $sup(S)$ denote the minimum and maximum (provided it exists) elements of S respectively. We then have the following identities.

$$\bigwedge_{j\in S}(\tau)^j\pi \equiv (\tau)^{inf(S)}\pi\ , \qquad \bigwedge_{j\in S}\epsilon^j\pi \equiv \epsilon^{inf(S)}\pi \qquad \text{and}$$
$$\bigvee_{j\in S}(\tau)^j\pi \equiv (\tau)^{sup(S)}\pi\ , \qquad \bigvee_{j\in S}\epsilon^j\pi \equiv \epsilon^{sup(S)}\pi \quad \text{if } S \text{ is finite.}$$

These properties enable us to follow the guiding principle that a preorder should be characterized by containment on sets of formulae.

If we were to extend the language to allow ϵ^0 as an operator, we would have that $p \vDash \epsilon^0\pi$ implies $p \nvDash \pi$. Then we would also be able to express the statement "*p diverges*" by the formula $p \vDash \epsilon^0 \bigvee_{k>0} \epsilon^k tt$ and the statement "*p can perform at least two τ actions*" by $\epsilon^0\epsilon^2 tt$.

The results that we have presented in this paper offer interesting technical insights into the construction of logical characterizations of preorders guided by the principles enunciated in section 1. As far as we are aware, in the literature, there are no logical characterizations of behavioural relations that lie strictly between strong and weak bisimilarity.

In future work, we hope to study and understand more about these operators and explore their interactions with fixpoint operators within a modal μ-calculus setting.

Acknowledgements. We are grateful to Sanjiva Prasad and Astrid Kiehn for pointing out various simplifications and corrections in the design of our logics, and for their patient reading and other constructive suggestions. In particular Sanjiva Prasad also pointed out that ϵ^0 would allow the full power of negation. We also thank Jamshid Bahgerzadeh for his valuable inputs and suggestions.

This work was partly supported by a research grant from MHRD, Government of India, under the Scheme of Research and Development in Technical Education (File No. F.26-1/2002-TS.V dated 19 March 2002).

References

1. S. Arun-Kumar and M. Hennessy: An Efficiency Preorder for Processes. Proceedings *Theoretical Aspects of Computer Software*, Sendai 1991, Lecture Notes in Computer Science 526, 152–175, Springer-Verlag 1991.
2. S. Arun-Kumar and M.Hennessy: An Efficiency Preorder for Processes. Acta Informatica, **29**, 737–760, Springer-Verlag, 1992.
3. S. Arun-Kumar and V. Natarajan: Conformance: A Precongruence Close to Bisimilarity. Proceedings *Structures in Concurrency Theory*, Springer Workshops in Computer Science Series, 1995.
4. F. Corradini and R. Gorrieri and M. Rocetti: Performance Preorder and Competitive Equivalence. Acta Informatica, **34**, 805–835, 1997.

5. M. Hennessy and R. Milner: Algebraic Laws for Nondeterminism and Concurrency. Journal of ACM, **32**, 137–161, 1985.
6. Kamal Jain and S. Arun-Kumar: Testing Processes for Efficiency. Proceedings *Fondations of Software Technology & Theoretical Computer Science 16*, Lecture Notes in Computer Science 1180, 100–110, Springer-Verlag, 1996.
7. L. Jenner and W. Vogler: Comparing the Efficiency of Asynchronous Systems. Technical Report 1998-3, Universität Augsburg, December 1998.
8. G. Luettgen and W. Vogler: A Faster than relation for Asynchronous Processes. Proceedings *CONCUR 2001*, Lecture Notes in Computer Science 2154, 262–276,
9. R. Milner: **Communication and Concurrency**, Prentice-Hall International, 1989.
10. R. Milner: Contractions, Handwritten notes, 1990.
11. R. Milner: Expansions. Handwritten notes, 1990
12. C. Stirling: **Modal and Temporal Properties of Processes**, Springer-Verlag 2001.
13. V. Natarajan and R. Cleaveland: An Algebraic Theory of Process Efficiency. Proceedings *Logic in Computer Science '96*, IEEE Computer Society Press 1996.
14. R. J. van Glabeek: The Linear Time – Branching Time Spectrum I. The Semantics of Concrete, Sequential Processes. In **Handbook of Process Algebra**, (eds.) J. A. Bergstra, A. Ponse and S. A. Smolka, Elsevier Science B. V., Netherlands., 2001.

Inherent Causal Orderings of Partial Order Scenarios

Bill Mitchell

Department of Computing, University of Surrey,
Guildford, Surrey GU2 7XH, UK
w.mitchell@surrey.ac.uk

Abstract. Scenario based requirements specifications are the industry norm for defining communication protocols. Basic scenarios captured as UML sequence diagrams, Message Sequence Charts (MSCs) or Live Sequence Charts (LSC) have partial order semantics that characterize system traces by restricting the possible order of events within those traces. The semantic partial order of the scenario specification is called the causal ordering.

Semantic inconsistencies often occur in partial order scenarios between the specified causal ordering and the order that events can occur in practice. Such inconsistencies are known as race conditions. The paper proves that there is a unique race free partial order that is a minimal weakening of the causal ordering. In other words, there is a canonical generalization of the requirements that corrects all race conditions. Hence any race free generalization of the original scenario is in fact a generalization of the canonical scenario. The paper also proves the dual result, there is a unique race free partial order that is a minimal strengthening of the causal order. I.e. there is a canonical refinement of the requirements that corrects all race conditions.

1 Introduction

UML sequence diagrams [19], Message Sequence Charts (MSCs) [18], and Live Sequence Charts (LSCs) [7] are popular for defining wireless and mobile communication protocols. The semantics of a basic scenario diagram defined with any of these languages can be given in terms of a partial order on the events in the scenario. The partial order restricts the order in which events can occur in any system trace. This partial order is called the causal ordering. We refer to any basic scenario diagram with such a semantics as a partial order scenario.

Although scenario specification languages have become quite sophisticated and have expressive powers beyond partial order scenarios, such scenarios are still the mainstay of industrial specifications. Consequently the study of partial order scenarios is still an active topic of research [5, 13, 12, 16]. Synthesizing various types of system models directly from these partial order scenarios is also an active area [1, 3, 4, 11, 15, 17]. Research into automatic test generation from partial order scenarios is another active research area [2, 6, 14].

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 113–127, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Industrial requirements specifications often contain inconsistencies between the specified causal ordering and the order that events can occur in practice. Race conditions are amongst the most common of these inconsistencies. Essentially a race condition asserts a particular order of events will occur as a consequence of the causal ordering, when in practise this order can not be guaranteed to occur. See [9, 10] for the original formal description of the problem within the MSC context.

It is possible to directly analyze the causal ordering to automatically detect race conditions [10]. This still leaves the onerous task of actually correcting the race conditions. Case studies such as [20] have shown that around a third of significant defects in SDL specifications are caused by poor requirements specifications. Since many SDL specifications are refined from MSCs and UML sequence diagrams this suggests a significant number of errors arise because of poor quality in partial order scenarios. Hence the ability to automatically correct race conditions would be of practical value.

In the paper we prove that given a causal ordering there exists a unique minimal weakening of that order which does not contain any race conditions, and which is itself the causal ordering of some scenario (Theorem 10). We call this weakening the inherent causal ordering, and the scenario to which it corresponds the inherent causal scenario. We prove the inherent causal scenario is canonical up to simulation equivalence of system behaviour. Therefore any race free generalization of the original scenario must be a generalization of the inherent causal scenario. Hence there is an optimal generalization of a partial order scenario that corrects all race conditions. In section 7 we describe an example MSC scenario from an industrial case study that illustrates how the inherent causal order can be of value in practise.

The paper also proves that there is a unique minimal strengthening of a causal order that corrects all race conditions, and which is equivalent to the causal ordering of some scenario (Theorem 18). We call this the inherent refinement ordering. As might be expected we prove the inherent refinement scenario is canonical up to simulation equivalence. Hence there is an optimal refinement of a partial order scenario that corrects all race conditions. The results can be generalized to scenarios that extend the basic partial order semantics with iteration and branching, as is the case with HMSCs. However, we do not prove that here due to lack of space.

Although our results are perfectly general and apply to any basic scenario diagram language such as basic UML sequence diagrams, MSCs or LSCs, we will use MSC as the central language for the paper. The MSC standard [18] is stable and MSCs are common in industry. Also MSC 2000 is being adopted within UML 2.0 [19]. In addition MSCs allow the most general form of causal ordering since it is possible for an MSC causal order to be almost any irreflexive transitive partial order.

2 Basic Partial Order Specifications

In this section we define the causal ordering semantics for partial order scenarios (e.g. basic MSCs). We use the same message semantics as the MSC 2000 standard

[18]. Hence, a partial order scenario defines a set of message exchanges between processes with asynchronous communication channels. Also we do not assume any type of buffering with the channels. However, the results in the paper do hold for both synchronous and FIFO channels.

Let $\mathcal{P}$ be a set of processes. A message m between processes is a pair $(!m, ?m)$ where $!m$ is the send event for m, and $?m$ is the receive event for m. We regard $!m$ as belonging to the sending process, and $?m$ as belonging to the receiving process. Let E be the set of all send and receive events between all processes. Each event has a label, let $l : E \longrightarrow L$ be the labelling function. For a message m, $l(!m) = l(?m)$. Within the MSC standard there are many other kinds of events such as action boxes and condition symbols, but here we only consider message events to simplify proofs as much as possible. It is straightforward to generalize the results to include these other events.

Definition 1. *A partial order scenario on processes $\mathcal{P}$ is*

- *a collection of disjoint sets $E(P) \subseteq E$, for each $P \in \mathcal{P}$ that defines the message events belonging to P,*
- *and a set of irreflexive partial orders $<_P$, where $<_P$ is a partial order on $E(P)$ that defines the local ordering of events for process P.*

These local partial orders must be subject to the constraint that for each send event $!m$ in a set $E(P)$ the corresponding receive event $?m$ occurs in some set $E(Q)$. Note messages are allowed to be sent from a process to itself, so we allow $P = Q$. We treat a partial order as a binary relation that can be represented as the set of pairs that are ordered by the relation. Hence we can take the union of partial orders, which is just the set theoretic union of the sets that represent the relevant order relations. It is important to note the local orders are not necessarily total, but can be any irreflexive partial order. In the literature it is sometimes assumed basic scenario diagrams have total local orderings, so it is worth emphasizing this does not have to be the case.

Let Msg be the set of messages defined as the set of send and receive event pairs:

$$\{(!e,\ ?e) \mid !e \in E(P) \text{ and } ?e \in E(Q) \text{ for some } P, Q \in \mathcal{P}\}$$

Definition 2. *The causal ordering $<_{\mathcal{C}}$ on a partial order scenario is the transitive closure of the relation given by*

$$\left(\bigcup_{P \in \mathcal{P}} (<_P) \right) \cup \mathsf{Msg}$$

From now on we will assume that all partial orders are transitive and irreflexive without loss of generality. We will also assume that all causal orderings are irreflexive, so that messages must be received after they are sent. We also, as is the norm, rule out message overtaking as shown in figure 2. The MSC standard includes a general ordering construct, which is a simple graphical notation that

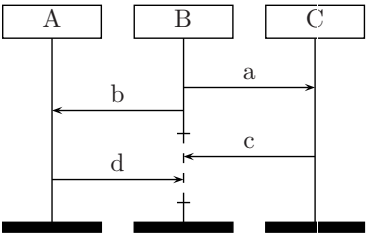


Fig. 1. Race hidden by coregion

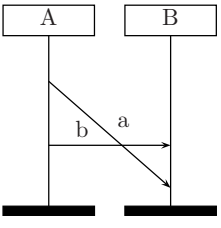


Fig. 2. Message Overtaking, which is prohibited

explicitly forces one event to occur before another event in the causal order. A general order construct is depicted as a dashed arrow between the events to be ordered, with arrow head placed in the middle of the arrow. In combination with the coregion construct that means a process order $<_P$ defined by an MSC can be any arbitrary irreflexive transitive partial order on the events $E(P)$.

The causal ordering defines the set of all possible system traces that are given by the partial order scenario. A system trace is any total order extension of $<_{\mathcal{C}}$. Recall a total order on a set S is a partial order $<$ on S where for any distinct elements $x, y \in S$, either $x < y$ or $y < x$.

Definition 3. *The set of system traces defined by a causal ordering $<_{\mathcal{C}}$ is the set of total order extensions of $<_{\mathcal{C}}$.*

Consider the MSC depicted graphically in figure 1. Each vertical line describes the time-line for a process, where time increases down the page. The distance between two events on a time-line does not represent any literal measurement of time, only that non-zero time has passed. Events on the same time-line are ordered linearly down the page, except where they occur within a coregion. Within a coregion events are not locally ordered unless that is directly imposed by a general order construct. Coregions are depicted with a dashed line. For process B events $?c$ and $?d$ are unordered as they occur within a coregion. The local partial orders defined by this MSC are given in figure 3 where we draw the ordering downwards, so that $!a <_B !b$ for example. In this case the causal ordering $<_{\mathcal{C}}$ is given in figure 4.

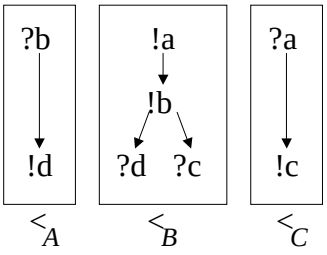


Fig. 3. Process Partial Orders

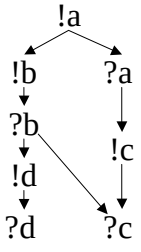


Fig. 4. Causal Ordering

Figure 1 illustrates a race condition. The causal ordering asserts that $!b <_C ?c$. If this MSC is taken as a specification it asserts that after C receives a it must send c so that it arrives after b is sent. It is not possible for C to know for sure when $!b$ occurs without querying B. Hence it is quite possible if this scenario is implemented naively that c will arrive before b is sent, contradicting the specification. This error can occur even though each of the processes A and C locally implements the specification correctly.

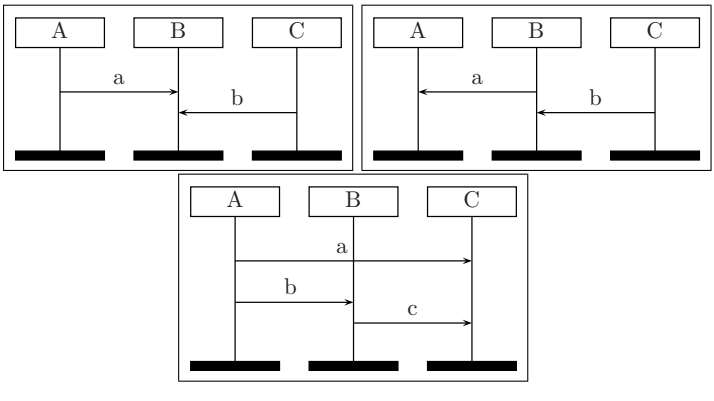


Fig. 5. Three basic types of race condition

Definition 4. *Define a partial order $<$ on E to be race free when for every event x and message e:*

$$x <?e \Rightarrow (x <!e \ \text{ or } \ x =!e)$$

We define an MSC to be race free when its causal ordering is race free.

That is $<$ is race free if the following holds. When $<$ orders an event x before the receive event of some message e, then it also orders x to be before the send event of e.

Note that figure 1 is not race free since $!b <_{\mathcal{C}} ?c$, but $!b \not<_{\mathcal{C}} !c$. The three basic types of race are illustrated in figure 5. In the first example we have $?a <_{\mathcal{C}} ?b$, but $\neg(?a <_{\mathcal{C}} !b)$. In the second example we have $!a <_{\mathcal{C}} ?b$, but $\neg(!a <_{\mathcal{C}} !b)$. In the last example we have $?a <_{\mathcal{C}} ?c$, but $\neg(?a <_{\mathcal{C}} !c)$. The third race example is the only one of the three that can be avoided by forcing messages to be synchronous. Hence the first two examples will cause semantic inconsistencies in synchronous and asynchronous scenarios.

3 Partial Order Processes

In this section we define a process algebra term that characterizes the traces that are defined by a causal ordering. This is a standard result for partial orders, but we present it in a slightly non-standard format for ease of use later in the paper. The process algebra term also characterizes the system behaviour up to strong bisimulation equivalence.

First we set up some notation for defining sets of events that are important in generating system and process traces. Let $<$ be a partial order on a set of events E. For a set $S \subseteq E$ define

$$\begin{aligned}
\mathsf{n}(S,<) &= \{x \in E \mid \exists\, y \in S : y < x, \\
&\qquad \text{and } \ \neg\exists\, z \in E : y < z < x\} \\
\mathsf{m}(S,<) &= \{x \in S \mid \neg\exists\, y \in S : y < x\} \\
\mathsf{af}(a,S,<) &= \mathsf{m}((S - \{a\}) \cup \mathsf{n}(\{a\},<),<)
\end{aligned}$$

The set $\mathsf{n}(S,<)$ are those events that are a least upper bound for some element in S. The set $\mathsf{m}(S,<)$ is just the set of minimal elements of S.

The set $\mathsf{af}(a,S,<)$ characterizes how events may follow a in an execution trace, where S describes the set of all events that are eligible to occur concurrently with a. Suppose we have an execution trace t that is a total extension of $<$. Let a be some event in t, so that t is of the form $t_0 \cdot a \cdot t_1$ (where $\cdot$ denotes concatenation). Let S be the set of minimal events from the set of all events in t_1. Then t_1 must be of the form $b \cdot t_2$ where $b \in \mathsf{af}(a,S,<)$. The first element that can occur in a trace that is a total extension of $<$ has to come from $\mathsf{m}(E,<)$. Hence we can define the system behaviour for a causal ordering as follows.

Definition 5. *For a set $S \subseteq E$ define a recursive process algebra term by*

$$P(S, <) = \sum_{\{a \in S\}} a \cdot P(\mathsf{af}(a, S, <), <)$$

and $P(\emptyset, <) = 0$.

Where $a \cdot P$ denotes the usual sequential composition of action and process, and the summation is nondeterministic choice (as standard in both CCS and CSP).

Definition 6. *Define the observable behaviour for causal ordering $<_{\mathcal{C}}$ to be the process:*

$$P(M) = P(\mathsf{m}(E, <_{\mathcal{C}}), <_{\mathcal{C}})$$

In [8] they define a process algebra term for M that defines the system traces for $<_{\mathcal{C}}$. Let $P^*(M)$ denote this process. Process $P(M)$ is strong bisimulation equivalent to $P^*(M)$. Hence $P(M)$ defines the system traces of the global behaviour for the processes defined by M.

Suppose that MSC M contains processes P_i for $1 \leq i \leq n$. Then a parallel composition of the $P(\mathsf{m}(E(P_i), <_{P_i}), <_{P_i})$ for $1 \leq i \leq n$ is strong bisimulation equivalent to $P(M)$ (which follows from an analogous result in [8]). However, we will not need to use that result here.

4 Inherent Causal Behaviour

A partial order $<$ on events preserves the message ordering when $!e <?e$ for every message e. Let $<_{\mathcal{C}}$ be the causal ordering for a partial order scenario.

Definition 7. *Define the inherent causal ordering $<_I$ of $<_{\mathcal{C}}$ to be the transitive closure of the following partial order relation $<$. For every event x and message e define:*

1. $x < !e \iff x <_{\mathcal{C}} !e$
2. $!e < ?e$

Note that when regarding a partial order as a set of pairs, we have

$$(<_I) \subseteq (<_{\mathcal{C}})$$

The inherent ordering is the causal order of some partial order scenario. This follows from the next lemma.

Lemma 8. *The inherent causal ordering $<_I$ of a partial order scenario with processes $\mathcal{P}$ is the transitive closure of the following partial order relation.*

1. $x < !e \iff \exists P \in \mathcal{P}$ *such that* $x <_P !e$
2. $!e < ?e$

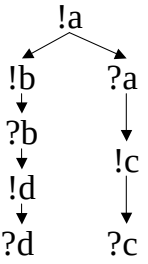


Fig. 6. Inherent Ordering

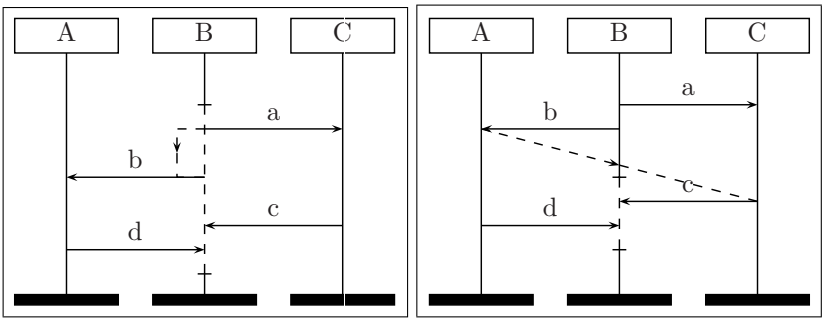


Fig. 7. Inherent Causal Ordering, and Inherent Refinement Ordering as MSCs

Figure 6 gives a graphical depiction of the inherent ordering for figure 1.

Since we are able to impose general orderings on events within MSC diagrams we can represent this inherent ordering as an MSC. That is we can define a second MSC who's causal ordering is in fact the inherent ordering of figure 1. This is the leftmost MSC in figure 7. Notice that the coregion for process B now covers all the events in $E(B)$. In order to assert that $!a$ must occur before $!b$ we have added a general ordering construct between these events. This is the dashed arrow, with arrow head placed at the mid point of the arrow. Wherever such a general ordering arrow occurs in an MSC from events x to y this explicitly defines $x <_{\mathcal{C}} y$. Thus definition 2 of $<_{\mathcal{C}}$ has to be extended so that it includes the set of pairs given by the general ordering construct.

5 Canonical Inherent Processes

Recall in section 3 we defined the observable process behaviour $P(M)$ of a partial order scenario M.

Definition 9. *Define the inherent process behaviour of a partial order scenario M to be $P_I(M) = P(\mathsf{m}(E, <_I), <_I)$*

Let $\sqsupset$ denote the standard simulation relation for process algebras. That is $P \sqsupset Q$ iff

for every transition $Q \xrightarrow{a} Q'$,
there exists a transition $P \xrightarrow{a} P'$ where $P' \sqsupset Q'$

Theorem 10

- $(<_I) \subseteq (<_{\mathcal{C}})$, *and* $P_I(M) \sqsupset P(M)$
- *For any race free partial order* $<$ *that preserves message ordering, let* $P_< = P(\mathsf{m}(E, <), <)$.
  Then $P_< \sqsupset P(M)$ *iff* $(<) \subseteq (<_I) \subseteq (<_{\mathcal{C}})$ *and* $P_< \sqsupset P_I(M)$

That is $P_I(M)$ *is the canonical process that simulates* $P(M)$ *and is race free. To say that* $(<_1) \subseteq (<_2)$ *means that for every* x *and* y *in* E, *when* $x <_1 y$ *then* $x <_2 y$.

This theorem proves that the order $<_I$ describes the maximal ordering with respect to simulation equivalence that is a race free weakening of $<_M$. Hence constructing an MSC that has partial order semantics given by $<_I$ defines a new MSC that corrects any race conditions in M, and weakens the causal ordering of M as little as possible. It is straightforward to construct such an MSC.

The theorem is a consequence of the following lemmas. For any partial order $<$ (which is not necessarily race free) let $T(<)$ be the set of total extensions of $<$. Lemma 11 follows immediately from our initial observations concerning the definition of $\mathsf{af}(a, S, <)$.

Lemma 11. *The set of traces of* $P_<$ *is exactly* $T(<)$, *the set of total order extensions of* $<$.

Lemma 12. *For partial orders* $<_1$ *and* $<_2$ *where*

$$P_{<_1} \sqsupset P_{<_2}$$

then $(<_1) \subseteq (<_2)$

Proof. Note that $x < y$ iff for every trace in $T(<)$, x occurs before y in the trace. When $P_{<_1} \sqsupset P_{<_2}$ then the set of traces for $P_{<_2}$ is contained in the set of traces for $P_{<_1}$, that is $T(<_2) \subseteq T(<_1)$.

$$\begin{aligned} x <_1 y &\Rightarrow x \text{ occurs before } y \text{ in every trace of } T(<_1) \\ &\Rightarrow x \text{ occurs before } y \text{ in every trace of } T(<_2) \\ &\Rightarrow x <_2 y \end{aligned}$$

Hence $(<_1) \subseteq (<_2)$, which concludes the proof.

Lemma 13. *Given two partial orders* $<_1$ *and* $<_2$,
$T(<_1) \subseteq T(<_2)$ *iff* $P_{<_1} \sqsubset P_{<_2}$

Proof. Note that $T(<_1) \subseteq T(<_2)$ iff $(<_2) \subseteq (<_1)$. Given $(<_2) \subseteq (<_1)$, to prove $P_{<_1} \sqsubset P_{<_2}$, it is enough to prove that for any $S \subseteq E$ and $a \in E$,

$$\mathsf{af}(a, S, <_1) \subseteq \mathsf{af}(a, S, <_2) \tag{1}$$

Let

$$m_1 = \mathsf{m}((S - \{a\}) \cup \mathsf{n}(\{a\}, <_1), <_1)$$
$$m_2 = \mathsf{m}((S - \{a\}) \cup \mathsf{n}(\{a\}, <_2), <_2)$$

We write $U \leq V$ for sets $U, V \subseteq E$, when for each $u \in U$, there is some $v \in V$ such that $u \leq v$. Note that since $(<_2) \subseteq (<_1)$ then $\mathsf{n}(\{a\}, <_2) \leq \mathsf{n}(\{a\}, <_1)$.

For a contradiction suppose that $x \in m_1$ and $x \notin m_2$. This implies there is some $y \in m_2$ such that $x <_2 y$. First consider if $y \in S - \{a\}$. Then $x <_1 y \in S - \{a\}$, hence $x \notin m_1$. This is a contradiction, hence we must have $y \in \mathsf{n}(\{a\}, <_2)$. Since $\mathsf{n}(\{a\}, <_2) \leq \mathsf{n}(\{a\}, <_1)$, there is some $y' \in \mathsf{n}(\{a\}, <_1)$ such that $x <_2 y <_1 y'$. Therefore $x <_1 y' \in \mathsf{n}(\{a\}, <_1)$, and so $x \notin m_1$. Again a contradiction as required to complete the proof of equation 1. The proof that $T(<_1) \subseteq T(<_2)$ implies $P_{<_1} \sqsubset P_{<_2}$, is completed once we note that $\mathsf{m}(E, <_1) \subseteq \mathsf{m}(E, <_2)$.

The converse implication is straightforward. It is true for any processes P and Q that if $P \sqsupset Q$ then the set of traces of Q is contained in the set of traces for P. Since the traces of $P_{<_i}$ are exactly $T(<_i)$, the result is then immediate. That completes the proof of the lemma. □

Lemma 14. *For a partial order $<$ that preserves message ordering and is race free,*

$$\Big((<) \subseteq (<_{\mathcal{C}})\Big) \Rightarrow \Big((<) \subseteq (<_I) \subseteq (<_{\mathcal{C}})\Big)$$

Proof. For this it is enough to prove that whenever $x < y$ then $x <_I y$. The proof splits into cases depending on whether y is a receive or send event. First suppose that $y = !e$ for some message e. Then $x <!e$ implies $x <_{\mathcal{C}} !e$ since $(<) \subseteq (<_{\mathcal{C}})$. By definition of $<_I$, $x <_{\mathcal{C}} !e$ implies $x <_I !e$.

The other case is where $y = ?e$ for some message e. Since $<$ is race free, $x <?e$ implies that $x <!e$. As above this implies $x <_I !e$. The ordering $<_I$ preserves message ordering, and hence $x <_I ?e$. This completes the proof of the lemma. □

6 Inherent Refinement Behaviour

In this section we prove the dual result of theorem 10, where instead of generalizing the causal ordering we refine it.

Definition 15. *Define the inherent refinement ordering $<_R$ of a causal ordering $<_{\mathcal{C}}$ to be the transitive closure of the following partial order $<$. For every event x and message e define:*

- $x <!e \iff x <_{\mathcal{C}} ?e$
- $!e <?e$

First note that $<_R$ is race free. Since it is clear from the definition that $x <_R ?e$ implies that $x <_R !e$ or $x = !e$. Also notice that the refinement order only extends $<_{\mathcal{C}}$ by forcing particular send events to be delayed so that other events may occur first, and hence is implementable.

If the partial order scenario is an MSC then the inherent refinement ordering can be constructed by adding suitable general orderings to the MSC, which cause appropriate send events to wait until the relevant receive events have occurred. For example the rightmost MSC of figure 7 gives the inherent refinement order for the partial order scenario in figure 1.

The use of general orderings is acceptable so long as they can be implemented in asynchronous distributed systems. We only use them to delay send events, and this effect can always be achieved by adding further messages to the partial order scenario. To force a general ordering $x <_{\mathcal{C}} !e$, where $x \in E(P)$ and $!e \in E(Q)$, we can add a new coordination message c with $!c \in E(P)$ and $?c \in E(Q)$. We then alter the local orderings by adding the pairs $x <_P !c$ and $?c <_Q !e$. Processes P and Q can enforce these orderings locally since they only have to delay send events to do so.

However, the choice of implementation is for the system designers who may use other mechanisms which are more appropriate for their particular circumstances. The goal of the solution presented here is to correct the semantics for the scenario in an optimal manner without altering the given message content of the scenario and without imposing any assumptions about communication channel semantics.

Lemma 16

$$(<_{\mathcal{C}}) \subseteq (<_R)$$

Proof. To prove this suppose $x <_{\mathcal{C}} y$. If $y = !e$ for some e, then $y <_{\mathcal{C}} ?e$. Hence from the definition $x <_{\mathcal{C}} ?e$ and hence $x <_R !e$. That is $x <_R y$.

When $y = ?e$, then $x <_R !e$. Also $!e <_R ?e$, hence by transitive closure, $x <_R ?e = y$. This completes the proof of the lemma. □

Lemma 17. *For any race free transitive partial order $<$ that preserves messages and where $(<_{\mathcal{C}}) \subseteq (<)$, then*

$$(<_{\mathcal{C}}) \subseteq (<_R) \subseteq (<)$$

Proof. To prove this first consider an event x and message e where $x \neq ?e$ and $x <_{\mathcal{C}} ?e$. That is $x <_R !e$. Since $(<_{\mathcal{C}}) \subseteq (<)$, we have $x < ?e$. Since $<$ is race free we have $x < !e$. Hence, if $x <_R !e$ then $x < !e$. Since $<$ preserves messages it trivially follows that $!e <_R ?e$ implies $!e < ?e$. Hence as $<$ is transitive we have proved that $(<_R) \subseteq (<)$. □

Given the lemmas already proved in section 5 we have thus proved the following theorem, which is the dual to theorem 10.

Theorem 18. *Let $P_R(M) = P(\mathsf{m}(E, <_R), <_R)$, then*

- *$(<_{\mathcal{C}}) \subseteq (<_R)$, and $P(M) \sqsupset P_R(M)$*
- *For any race free partial order $<$ that preserves message ordering, let $P_< = P(\mathsf{m}(E, <), <)$.*
  Then $P(M) \sqsupset P_<$ iff $(<_{\mathcal{C}}) \subseteq (<_R) \subseteq (<)$ and $P_R(M) \sqsupset P_<$

Hence $<_R$ is the canonical refinement of the causal order that corrects all race conditions in the specification.

7 Industrial Case Study Example

In collaboration with Motorola Research Labs, we have been conducting a number of case studies [6, 11] into automating pathology detection in MSC telecommunication specifications. Figure 8 is an anonymized example from a Motorola case study, which contains multiple race conditions. The original diagram is a UML 2.0 sequence diagram that describes traffic channel allocation and activation between various processes for a telecommunication protocol. Process A has delegated the task of determining what resource to allocate to process B.

A parallel construct in a MSC/UML sequence diagram, denoted by `PAR`, describes a set of concurrent threads that occur in the diagram. Dotted lines delineate the different threads. Hence, events from one thread are not causally ordered with respect to events from any other thread. Figure 8 contains two parallel constructs. The first parallel construct contains messages a, b and c in separate threads, which can therefore occur in any order. The bounding box of a parallel construct has no effect on the ordering of events, it solely delineates the scope of the concurrent threads. Note an MSC/UML sequence diagram containing solely messages, coregions and parallel constructs still defines a partial order scenario in the sense of definition 1.

An inline reference, denoted by `REF`, is a place holder for another sequence diagram. The reference can be replaced by the contents of the other sequence

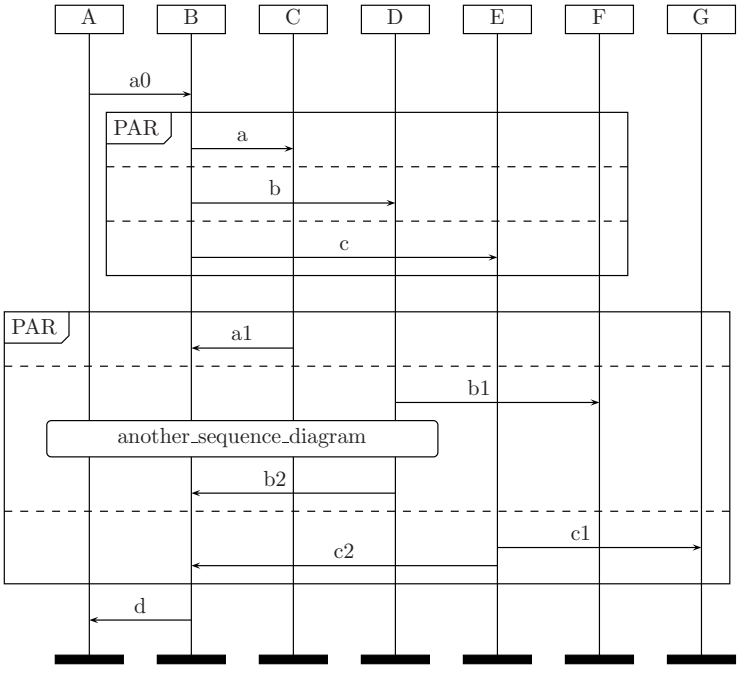


Fig. 8. UML 2.0 case study example with multiple race conditions

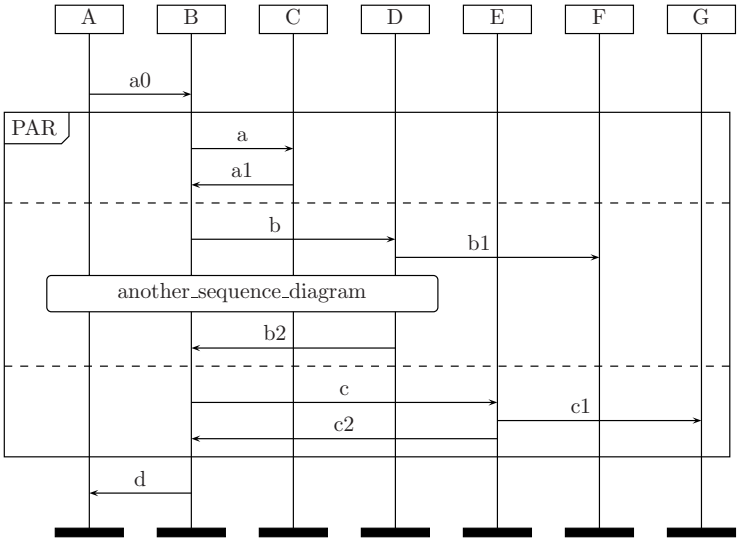


Fig. 9. Inherent Causal Scenario for Figure 8 as MSC

diagram if desired. The reference is weakly composed with the referring diagram when inlined. Figure 8 contains an inline reference spanning processes A through D. We will assume for this example that the inline reference is some linear ordering of events in order to simplify our calculations.

In total we have the following six race conditions in figure 8. Event $?a1$ is in a race with $!b$ and also with $!c$. Event $?c2$ is in a race with $!a$ and also with $!b$. Also event $?b2$ is in a race with $!a$ and also with $!c$. It may be that the authors implicitly assumed the downlink latency from B is much shorter than the uplink latency for the other processes. If this were true it may be possible in practise for the specification to be realizable. However it is far safer to rewrite the specification without these race conditions.

One way to remove these races would be to regroup the messages within a single parallel construct. Messages a and $a1$ could be grouped within the same thread of a parallel construct. Similarly b, $b1$, $b2$ and the inline reference could be grouped in a second thread. Finally c, $c1$ and $c2$ could be grouped in the third thread. Figure 9 depicts this solution. It seems reasonable to suppose this will not contradict what the authors originally intended.

Figure 9 is exactly the inherent causal scenario of figure 8. In this case the inherent causal order for figure 8 would seem to represent the specification intended by the authors, rather than the causal order of figure 8 itself.

The UML 2.0 case study also contained cases of sequence diagrams where a more intuitively 'correct' specification was given by the inherent refinement ordering, rather than the inherent causal ordering. Hence, we are not proposing either inherent ordering as some kind of panacea. However, providing practitioners

with both inherent orderings in the form of MSCs (or UML sequence diagrams) will give them a better understanding of what specifications are possible.

8 Conclusion

The paper has proved that there is a canonical solution for correcting all race conditions within a partial order scenario by weakening the causal relationship. Moreover the solution can be easily automated via lemma 8. The inherent causal ordering that defines the solution can also be presented in MSC format by use of the MSC coregion, parallel and general ordering constructs. Section 7 gave an example from an industrial case study where the inherent causal ordering captured the intended behaviour of a specification containing multiple race conditions.

The paper has also proved the dual result, that there is a canonical refinement of the specifications that corrects all race conditions. This is the inherent refinement ordering. This also can be presented in MSC format, which can be constructed automatically. Together these inherent orderings provide a useful insight into the semantically consistent specifications that are possible for a distributed system.

References

1. R. Alur, K. Etessami, M. Yannakakis, Inference of Message Sequence Charts, Proceedings 22nd International Conference on Software Engineering, pp 304-313, 2000.
2. M. Beyer, W. Dulz, F. Zhen, Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains, Proceedings of 12th Asian Test Symposium (ATS'03), IEEE 2003.
3. Yves Bontemps, Pierre-Yves Schobbens, Synthesis of Open Reactive Systems from Scenario-Based Specifications, (ACSD'03)
4. Yves Bontemps, Patrick Heymens, Turning high-level live sequence charts into automata, Proc of Scenarios and State Machines: Models Algorithms and tools, 24th International Conf. on Software Engineering, May 2002, ACM.
5. E. Gunter, A. Muscholl, D. Peled, Compositional Message Sequence Charts, TACAS 2001.
6. P. Baker, P. Bristow, C. Jervis, D. King, B. Mitchell, Automatic Generation of Conformance Tests From Message Sequence Charts, Proceedings of 3rd SAM Workshop 2002, The Broader Applicability of MSC and SDL, pp 170-198, LNCS 2599.
7. David Harel, Werner Damm LSCs: Breathing Life into Message Sequence Charts, Formal Methods in System Design, 19, 45-80, 2001
8. T. Gehrke, M. Hilhn, H. Wehrkeim, An algebraic semantics for message sequence chart documents. In FORTE/PSTV'98, pages 3-18. Kluwer Academic Publishers, 1998.
9. Gerard J. Holzmann, Doron A. Peled, Message Sequence Chart Analyzer, United States Patent, 5,812,145.
10. Gerard J. Holzmann, Doron A. Peled, and Margaret H. Redberg, An analyzer for message sequence charts, Software Concepts and Tools, 17(2), 1996.

11. B. Mitchell, R. Thomson, C. Jervis, Phase Automaton for Requirements Scenarios, Feature Interactions in Telecommunications and Software Systems VII, 77-84, 2003, IOS Press.
12. A. Muscholl, D. Peled: From Finite State Communication Protocols to High-Level Message Sequence Charts. ICALP 2001, 720-731.
13. D. Peled: Specification and Verification using Message Sequence Charts. Electronic Notes in Theoretical Computer Science 65, No 7, 2002.
14. E. Rudolph, I. Schieferdecker, J. Grabowski: Development of a MSC/UML Test Format. 153-164, Formale Beschreibungstechniken fur verteilte Systeme, 2000. Verlag Shaker 2000, ISBN 3-8265-7491-5.
15. J. Schumann, J. Whittle, Generating Statechart Designs From Scenarios, Proceedings 22nd international conference on on Software engineering, 2000.
16. A. Tsiolakis, Integrating Model Information in UML Sequence Diagrams, Electronic Notes in Theoretical Computer Science, June 2001.
17. S. Uchitel, J. Kramer, J. Magee, Synthesis of Behavioral Models from Scenarios, IEEE Transactions on Software Engineering, vol. 29, no. 2, February 2003
18. Z.120 (11/99)ITU-T Recommendation - Message Sequence Chart (MSC)
19. Object Management Group (OMG), *Unified Modeling Language (UML): Superstructure, Version 2.0*, 2003. Available from `http://www.omg.org`.
20. E. Wong, J. R. Horgan, W. Zage, D. Zage and M. Syring, Applying Design Metrics to a Large-Scale Software System, (Motorola), Proceedings of the 9th International Symposium on Software Engineering Reliability (ISSRE '98), Paderborn, Germany, November 4-7, 1998.

Atomic Components

Steve Reeves and David Streader

Department of Computer Science, University of Waikato, Hamilton, New Zealand
{dstr, stever}@cs.waikato.ac.nz

Abstract. The operational definition of observational congruence in CCS and ACP can be split into two parts: one, the definition of an observational semantics (*i.e.* abstraction); and two, the definition of a strong congruence. In both cases this "separation of concerns" has been applied with abstraction that is implicitly "fair". We define a novel (if obvious) observational semantics with no implicit "fairness". When combining this observational semantics with failure equality the resulting observational semantics is shown to be equal, other than for minor details, to NDFD semantics. We also combine our observational semantics with singleton failure semantics and we establish congruence results for this new observational equality.

1 Introduction

Industry is looking to create a market in reliable "plug-and-play" components. To do this the *interface* [1] of a component needs to be defined in a way that makes it safe to substitute components with the same interface.

Microsoft approaches this issue using Abstract State Machines (ASM) as a starting point and have noted [2] that it would be very useful to combine the event-based process algebras, which have modular reasoning built in, with the descriptive ability of the state-based ASM. To this end some process algebra features have been added to ASM [2] but many conceptual difficulties remain. An alternative approach is to start with a process algebra and enrich its descriptive ability to be more like ASM while retaining the desired modularity. The work of [3] can be interpreted as an example of this approach.

The field of Discrete Event Systems (DES, Ramadge and Wonham [4]) also lacks modularity in the style of process algebra and the work of Heymann and Meyer [5, 6] adds some process algebraic features to a DES formalism.

Here we work in the other direction and look for a process algebraic formalism that would be suitable basis for a state and event formalism.

We are going to consider only simple components with atomic states and atomic actions. Because of the simple semantics of our components they are easily recognisable as a small extension of processes. To take advantage of the well-known [7, 8, 9] isomorphism between state-based relational semantics and event-based operational semantics we focus our attention on the operational rather than denotational semantics of processes.

Out of all the many semantics, we focus on failure semantics for process and singleton failures for abstract data types (ADT) because they: one, have a very realistic

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 128–139, 2005.
© Springer-Verlag Berlin Heidelberg 2005

testing characterisation [10, 11, 12]; two, are congruent; and three, the removal of all τ actions is sound. Hoare and He say [13][p.198] "The main distinguishing feature of CSP is to define a hiding operator that succeeds in total concealment of internal actions."

There is, from an operational perspective, a natural "separation of concerns" that allows a decomposition of observational equivalences into two parts: one, abstraction (the construction of the observational semantics); and two, a definition of a strong, *i.e.* non-observational, equivalence (see Section 2.3). This approach is taken in [14, 15] where choice is implicitly assumed to be "fair".

We define a novel (if obvious) observational semantics modelling divergence where choice is not assumed to be fair. When combining this observational semantics with strong failure equivalence the resulting observational equivalence is, other than for minor details, NDFD equivalence [16].

Our definition of abstraction models divergence independently of any given strong equality. By applying our definition of abstraction and singleton failures equivalence [12] we construct an observational semantics for state-based definitions of components that is congruent with respect to composition and hiding.

Our model of components consists of two distinct parts, an interface that is public and an internal part that is private. Both states and events can be in either. We define an observational equality that "preserves" the interface and is **congruent with respect to composition and hiding (abstraction).**

When considering the start and end states to be in the interface and all other states to be internal then the *standard operational approach* to building congruences w.r.t. choice, as found in [14, 15], is to keep the internal actions that cross the internal - interface boundary. The *standard denotational approach* to building congruences w.r.t. choice, as found in [17, 18, 19], is to add stability. These two "standard" approaches result in slightly different equalities. Indeed the only difference between our operationally defined equality and the denotationally defined NDFD is that we adopt the standard operational approach to congruences (see Section 4.1).

It is possible to take previously defined models, one for states and one for events, and then glue them together [20, 8, 21]. This has the advantage of making tool reuse easy but requires accepting each model on an all-or-nothing basis. We are trying to define a model that treats states and events on an equal footing by taking what best fits our needs from a range of models.

We take an operational approach [14, 15] with failure [17, 18] or singleton failure [12] semantics and apply a novel operational definition of hiding that models divergence. To be consistent with our intention of distinguishing states in the interface and internal states we have used choice from [22] and not the more well-known process algebras [17, 18, 11, 14, 15]. We also reject CSP's external choice as hiding (and operationally $_\tau_S$) does not distribute through this definition of choice.

The discussion above has been rather general and conceptual, but our motivation has been to provide a semantics that permits modular reasoning *in practice*. So, in an attempt to give some assurance that our framework is of practical use, we use it (in Section 3) to model and reason about a simple example and briefly compare our model with well-known models from the literature.

2 Component Specifications

Components consist of atomic states and events. Both can either be a part of the component's *interface* or are internal to the component.

We will write $\mathsf{A}, \mathsf{B} \ldots$ for components and will assume a universe of observable action names $\mathsf{a}, \mathsf{b} \ldots \in Act$ and τ for internal actions and $Act^\tau \stackrel{\text{def}}{=} Act \cup \{\tau\}$ and we let $\mathsf{x}, \mathsf{y} \ldots \in Act^\tau$. We use a set of state propositions Π where $\{s, e\} \subseteq \Pi$ to define the state component of the interface (unlike [3]).

Definition 1. *Component transition system (CTS)*

$$\mathsf{A} \stackrel{\text{def}}{=} (N_\mathsf{A}, Os_\mathsf{A}, T_\mathsf{A}) \textit{ is a CTS where}$$

N_A *is a finite set of nodes - representing states,*
T_A *a set of transitions* $T_\mathsf{A} \subseteq \{(n, \mathsf{x}, m) \mid n, m \in N_\mathsf{A} \wedge \mathsf{x} \in Act^\tau\}$*- representing events,*
$Os_\mathsf{A} : N_\mathsf{A} \to 2^\Pi$ *(the observable states of* A*).* •

By referring to a state as "observable" all that we mean is that the state is a part of the *interface*. We use our two special state propositions $\{s, e\}$ to encode a set of start states $s_\mathsf{A} \stackrel{\text{def}}{=} \{n \mid s \in Os_\mathsf{A}(n)\}$ and a set of end states $e_\mathsf{A} \stackrel{\text{def}}{=} \{n \mid e \in Os_\mathsf{A}(n)\}$. In figures will use *s* for start nodes, *e* for end nodes and *se* for nodes that are both (see Fig. 1). The transitions of nonterminating components can be defined using a set of linear recursive equations, *e.g.* $\{X = \mathsf{a}X + \mathsf{b}Y, Y = \mathsf{c}Z\}$ (see [15]).

The above set of equalities can be interpreted as giving a mutually recursive definition of $\{X, Y, Z\}$. Alternatively $\{X, Y, Z\}$ can simply be interpreted as a set of states and an equality can be interpreted as defining the set of actions with the same pre-state. Each pair of the name and state on the right of the equality defines the name and post-state of a transition. For example B in Fig. 1 can be defined by $N_\mathsf{B} \stackrel{\text{def}}{=} \{X, Y, Z\}$ (seen as $\{s, \circ, se\}$), $s_\mathsf{B} \stackrel{\text{def}}{=} \{X, Z\}$, $e_\mathsf{B} \stackrel{\text{def}}{=} \{Z\}$ and $T_\mathsf{B} \stackrel{\text{def}}{=} \{X = \mathsf{a}Y, Y = \mathsf{c}Z, Z = \mathsf{b}Y\}$.

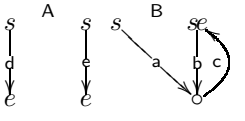


Fig. 1. A and B

If $Os_\mathsf{A}(n) = \varnothing$ then n is an *internal* state, else it is a part of the interface. We write $n \xrightarrow{\mathsf{x}} m$ for $(n, \mathsf{x}, m) \in T_\mathsf{A}$ and $n \xrightarrow{\mathsf{x}}$ for $\exists_m . (n, \mathsf{x}, m) \in T_\mathsf{A}$.

We define the ready set of a state $\pi(s) \stackrel{\text{def}}{=} \{\mathsf{x} \mid s \xrightarrow{\mathsf{x}}\}$ and alphabet of a process as $\alpha(\mathsf{A}) \stackrel{\text{def}}{=} \{\mathsf{x} \mid n \xrightarrow{\mathsf{x}} m \in T_\mathsf{A}\}$.

Processes are Special Components. A process can easily be seen as a component with state propositions restricted to propositions for the start and end of the process. Clearly process $(N_\mathsf{A}, Os_\mathsf{A}, T_\mathsf{A})$ defines a labelled transition system (LTS) $(N_\mathsf{A}, s_\mathsf{A}, e_\mathsf{A}, T_\mathsf{A})$. So, we can say that every CTS can be viewed as an LTS, which we shall do in the sequel, especially when being able to relate back to the standard world of LTS is desirable.

Although we regard our atomic components as a minor extension of processes these processes are slightly different from those of CSP, CCS and ACP in that they have a set of start states.

2.1 A Component Algebra

The congruence of process algebraic equivalences give them their much sought-after modularity. With this in mind we define component operators, taken from the process literature, in order that our observational equivalences are congruent. We adopt the operators from the process literature: $\sqcap$ internal choice; $\oplus$ choice; ; sequential composition; and $\|_\gamma$ parallel composition.

How we would wish operators to affect state propositions depends upon the meaning of the propositions. Thus operators parameterised by "proposition rules" would seem appropriate (as in [3]). Here we only model what is neededfor start and end states.

Choice has been defined ([15, 22]) on operational semantics with one start state by gluing the two start states together. Here we glue together two sets of start states. The reason for this generalisation is that we wish to model components that, like CSP processes, can immediately be nondeterministic; but unlike CSP we wish hiding (abstraction) to distribute through choice.

Let $S' = \{s'_1, s'_2, \ldots, s'_m\}$ and $S = \{s_1, s_2, \ldots, s_n\}$ then define $\{\{S/S \times S'\}\}$ to be the n substitutions $\{s_i/\{\text{“}(s_i, s'_1)\text{”}, \ldots, \text{“}(s_i, s'_m)\text{”}\}\}$ for $s_i \in S$ and define $\{\{S'/S \times S'\}\}$ to be the m substitutions $\{s'_j/\{\text{“}(s_1, s'_j)\text{”}, \ldots, \text{“}(s_n, s'_j)\text{”}\}\}$ for $s'_j \in S'$. We use quotes to indicate that "(s_1, s'_j)" is a name and any subsequent substitution must match on the whole name not just part of it such as "s_1".

We define $\{\{SS'/S \times S'\}\}$ to be the $n+m$ substitutions $\{\{S/S \times S'\}\} \cup \{\{S'/S \times S'\}\}$. The first n substitutions replaces each element of $\{s_1, s_2, \ldots, s_n\}$ with m nodes and the last m substitutions replaces each element of $\{s'_1, s'_2, \ldots, s'_m\}$ with n nodes. Consequently $\{\{s_\mathsf{A} s_\mathsf{B}/s_\mathsf{A} \times s_\mathsf{B}\}\}$ will identify the two sets of nodes s_A and s_B as $s_\mathsf{A}\{\{s_\mathsf{A} s_\mathsf{B}/s_\mathsf{A} \times s_\mathsf{B}\}\}$ and $s_\mathsf{B}\{\{s_\mathsf{A} s_\mathsf{B}/s_\mathsf{A} \times s_\mathsf{B}\}\}$ are both the $n \times m$ set of nodes $s_\mathsf{A} \times s_\mathsf{B}$.

First we define what it means to have a set of nodes in a transition:

$$n \xrightarrow{\mathsf{x}} \{s_1, \ldots, s_n\} \stackrel{\text{def}}{=} \{n \xrightarrow{\mathsf{x}} s_1, \ldots, n \xrightarrow{\mathsf{x}} s_n\}$$

Using this we can apply our node substitutions, *e.g.* $\{\{s_\mathsf{A} s_\mathsf{B}/s_\mathsf{A} \times s_\mathsf{B}\}\}$, to sets of transitions by applying the substitutions to the two nodes in each transition in the set. Consquently, applying $\{\{s_\mathsf{A} s_\mathsf{B}/s_\mathsf{A} \times s_\mathsf{B}\}\}$ to all the components of A and B will glue together the two sets of states.

Definition 2. *Operations $\oplus$, ; τ_S δ_S and $\|_\gamma$ on CTSs A and B, where γ is a partial function from Act $\times$ Act to Act, $S \subseteq Act$ and $\mathsf{x} \in Act^\tau$.*

$N_{\mathsf{A}\|_\gamma\mathsf{B}} \stackrel{\text{def}}{=} N_\mathsf{A} \times N_\mathsf{B}$, $Os_{\mathsf{A}\|_\gamma\mathsf{B}}((n,m)) \stackrel{\text{def}}{=} Os_\mathsf{A}(n) \cap Os_\mathsf{B}(m)$, $T_{\mathsf{A}\|_\gamma\mathsf{B}}$ *is defined by:*

$$\frac{n \xrightarrow{\mathsf{x}}_\mathsf{A} n',\ o \in N_\mathsf{B}}{(n,o) \xrightarrow{\mathsf{x}} (n',o)} \quad \frac{m \xrightarrow{\mathsf{x}}_\mathsf{B} m',\ o \in N_\mathsf{A}}{(o,m) \xrightarrow{\mathsf{x}} (o,m')} \quad \frac{n \xrightarrow{\mathsf{a}}_\mathsf{A} n',\ m \xrightarrow{\mathsf{b}}_\mathsf{B} m',\ \gamma(\mathsf{a},\mathsf{b})=\mathsf{c}}{(n,m) \xrightarrow{\mathsf{c}} (n',m')}$$

$N_{\mathsf{A}\tau_S} = N_\mathsf{A}$, $Os_{\mathsf{A}\tau_S}(n) = Os_\mathsf{A}(n)$, $\dfrac{n \xrightarrow{\mathsf{x}}_\mathsf{A} n',\ \mathsf{x} \notin S}{n \xrightarrow{\mathsf{x}}_{\mathsf{A}\tau_S} n'}$ $\quad \dfrac{n \xrightarrow{\mathsf{a}}_\mathsf{A} n',\ \mathsf{a} \in S}{n \xrightarrow{\tau}_{\mathsf{A}\tau_S} n'}$ $\quad \dfrac{n \xrightarrow{\mathsf{x}}_\mathsf{A} n',\ \mathsf{x} \notin S}{n \xrightarrow{\mathsf{x}}_{\mathsf{A}\delta_S} n'}$

$N_{\mathsf{A}\delta_S} = N_\mathsf{A}$, $Os_{\mathsf{A}\delta_S}(n) = Os_\mathsf{A}(n)$,

$\mathsf{A} \sqcap \mathsf{B} \stackrel{\text{def}}{=} (N_\mathsf{A} \cup N_\mathsf{B}, Os_{\mathsf{A}\sqcap\mathsf{B}}(n) \stackrel{\text{def}}{=}$ *if* $n \in N_\mathsf{A}$ *then* $Os_\mathsf{A}(n)$ *else* $Os_\mathsf{B}(n), T_\mathsf{A} \cup T_\mathsf{B})$

$\mathsf{A} \oplus \mathsf{B} \stackrel{\text{def}}{=} (\mathsf{A} \sqcap \mathsf{B})\{\{s_\mathsf{A} s_\mathsf{B}/s_\mathsf{A} \times s_\mathsf{B}\}\}$

Let $s^*_\mathsf{B} = \{s^*_i \mid s_i \in s_\mathsf{B}\}$ *and* $s^* e^*_\mathsf{B} = \{s^*_i \mid s_i \in s_\mathsf{B} \cap e_\mathsf{B}\}$ *and* $Os_{\mathsf{A}\mathsf{B}}(n) \stackrel{\text{def}}{=}$ *if* $n \in N_\mathsf{A}$ *then* $Os_\mathsf{A}(n)$ *elseif* $n \in N_\mathsf{B}$ *then* $Os_\mathsf{B}(n)$ *else if* $n = s^*_i$ *then* $Os_\mathsf{B}(s_i)$

$$\mathsf{A};\mathsf{B} \stackrel{\text{def}}{=} (N_\mathsf{A} \cup N_\mathsf{B} \cup s^*_\mathsf{B}, Os_\mathsf{AB}, T_\mathsf{A} \cup T_\mathsf{B} \cup \{s^*_i \xrightarrow{\mathsf{x}} n \mid s_i \in s_\mathsf{B} \wedge s_i \xrightarrow{\mathsf{x}} n\})\{\{e_\mathsf{A} s^*_\mathsf{B}/e_\mathsf{A} \times s^*_\mathsf{B}\}\}$$

Hiding actions in the set S can be defined in terms of renaming the actions in S to τ and then abstracting them. The details are obviously dependent upon how the actions are abstracted. We return to this in Section 2.3.

Our definition of choice, see $\mathsf{A} \oplus \mathsf{B}$ Fig. 2, is based on that from [22] where the start states of both processes are simply glued together. This is unlike that of ACP where the processes are first root unwound. We amend the definition of [22] in the obvious way to cope with a set of start states.

In the example $\mathsf{A};\mathsf{B}$(Fig. 2) the transitions with dotted arrows have been added by our definition of sequential composition.

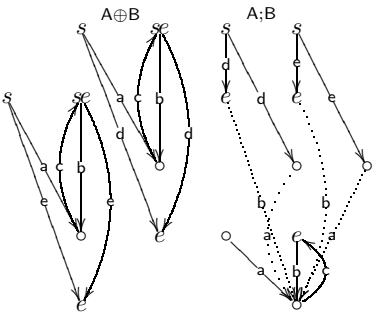


Fig. 2. $\mathsf{A} \oplus \mathsf{B}$, $\mathsf{A};\mathsf{B}$ (A,B in Fig. 1)

The details of sequential composition are dependent upon the details of successful termination but CSP and ACP treat successful termination differently. Our definitions are based on ACP not CSP (adding the actions $\{n \xrightarrow{\mathsf{x}} s \mid e \in e_\mathsf{A} \wedge s \in s_\mathsf{B} \wedge n \xrightarrow{\mathsf{x}} e\}$ would result in a more CSP-like definition). For further discussion see Section 4.2 later.

2.2 Strong Equivalences

A common denotational approach is to define processes without any τ actions and to define hiding as removing some actions. Here we adopt a more operational approach and rename some actions as τ actions and then define an abstraction function that removes the τ actions.

Failures are usually defined with traces that are sequences of action names. But we wish to take account of state observation and the fact that we have a set of start states.

We write $n_1 \xrightarrow{Os(n_1)\mathsf{x}_1 Os(n_2)\mathsf{x}_2 \ldots \mathsf{x}_k Os(n_{k+1})} n_{k+1}$ when $\exists_{n_2,\ldots n_k} .(n_1, \mathsf{x}_1, n_2) \in T_\mathsf{A}$, $\ldots (n_k, \mathsf{x}_k, n_{k+1}) \in T_\mathsf{A}$ and let θ range over alternating sequences of state observations $Os(n_i)$ and actions $\mathsf{x}_i \in Act^\tau$.

$$Tr(\mathsf{A}) \stackrel{\text{def}}{=} \{\theta \mid m \xrightarrow{\theta} n \wedge s \in Os_\mathsf{A}(m)\}$$
$$F(\mathsf{A}) \stackrel{\text{def}}{=} \{\langle \theta, X\rangle \mid m \xrightarrow{\theta} n \wedge s \in Os_\mathsf{A}(m) \wedge \forall_{\mathsf{x} \in X} .n \stackrel{\mathsf{x}}{\not\longrightarrow}\}.$$
$$SF(\mathsf{A}) \stackrel{\text{def}}{=} \{\langle \theta, \{x\}\rangle \mid m \xrightarrow{\theta} n \wedge s \in Os_\mathsf{A}(m) \wedge n \stackrel{\mathsf{x}}{\not\longrightarrow}\}.$$
$$\mathsf{A} =_F \mathsf{B} \Leftrightarrow F(\mathsf{A}) = F(\mathsf{B}) \qquad \mathsf{A} =_{SF} \mathsf{B} \Leftrightarrow SF(\mathsf{A}) = SF(\mathsf{B})$$

If we restrict our components to having only start and end predicates and further restrict them so that no transition enters a start node nor leaves an end node then our definition of failures on processes without τ actions is the same as that in CSP.

2.3 Component Abstraction

The CTSs in Definition 1 take no account of τ actions being unobservable, so we would call it a **strong semantics** ($\rightarrow$) and an equivalence based on it a **strong equivalence** ($=_X$, where X is the sort of equivalence, *e.g.* failure).

Definition 3. $\stackrel{\mathsf{a}}{\Longrightarrow}_o$ *is a predicate where:*

$$s_1 \stackrel{\tau}{\Longrightarrow} s_n \stackrel{\text{def}}{=} s_0 \stackrel{\tau}{\longrightarrow} s_1, s_1 \stackrel{\tau}{\longrightarrow} s_2, \ldots s_{n-1} \stackrel{\tau}{\longrightarrow} s_n \wedge \forall_{i \leq n} . Os(s_0) = Os(s_i)$$

$$n \stackrel{\mathsf{a}}{\Longrightarrow}_o m \stackrel{\text{def}}{=} n \stackrel{\tau}{\Longrightarrow} n', n' \stackrel{\mathsf{a}}{\longrightarrow} m', m' \stackrel{\tau}{\Longrightarrow} m \wedge (\mathsf{a} \in Act \vee Os(n') \neq Os(m'))$$

We define a parameterised abstraction function from a strong semantics:

$$Abs_z(\mathsf{A}) \stackrel{\text{def}}{=} \langle N_{\mathsf{A}}, Os_{\mathsf{A}}, \{(n,x,m) \in T_{\mathsf{A}} \mid n \stackrel{x}{\Longrightarrow}_z m\} \rangle$$

Here the parameter z tells us, as we will see, what sort of (observational) semantics we are dealing with, i.e. *whether it disregards divergent behaviour* etc. *If* A *is a component then* $Abs_z(\mathsf{A})$ *is its* **interface**, i.e.*defines how it can interact with any context.*

A method of abstraction Abs_z *is* **well defined** *w.r.t. a strong equivalence* $=_X$ *when:*

$$\textit{if } \mathsf{A} =_X \mathsf{C} \textit{ then } Abs_z(\mathsf{A}) =_X Abs_z(\mathsf{C})$$

We define **observational equivalence**s *(*$=_{zX}$*) as:*

$$\mathsf{A} =_{zX} \mathsf{C} \stackrel{\text{def}}{=} Abs_z(\mathsf{A}) =_X Abs_z(\mathsf{C}). \quad \bullet$$

We will use a lower case prefix to depict the abstraction function and an upper case suffix to depict the strong equivalence (*e.g.* F for failure), so for example we have:

$$\mathsf{A} =_{oF} \mathsf{C} \stackrel{\text{def}}{=} Abs_o(\mathsf{A}) =_F Abs_o(\mathsf{C})$$

The effect of our definition of abstraction acts on τ actions that are not connected to any state in the interface can be seen in Fig. 3 and is normal [18, 19] for failure style semantics. Keeping τ labelled transitions that are connected to the start and end states is the usual operational technique [14, 15] to define observational congruences with respect to choice and sequential composition.

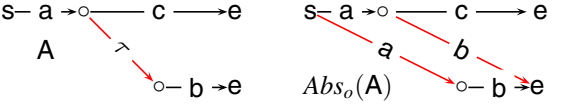


Fig. 3. Action abstraction

Different versions of hiding can be derived from different definitions of abstraction since $\mathsf{A}/_z H \stackrel{\text{def}}{=} Abs_z(\mathsf{A}\tau_H)$. CSP's denotational semantics has no τ actions and hiding, for terminating processes, [18–Ch.3] can be formalised by $\mathsf{A}/_o H \stackrel{\text{def}}{=} Abs_o(\mathsf{A}\tau_H)$, whereas with CSP's operational semantics [18–Ch.7] and NDFD semantics [23] hiding is defined by $_\tau_H$ and the observational semantics is defined by $\stackrel{\mathsf{a}}{\Longrightarrow}_o$.

Is Divergence Observable? Although we believe divergence not to be directly observable we will show that the distinction between livelock and deadlock it is a consequence of the interleaving assumption and the unfairness of choice.

The interleaving assumption equates concurrent processes to a sequential process. We write $\mathsf{P} \parallel_\varnothing \mathsf{Q}$ for P in parallel with but not synchronising with Q. We assume that choice can behave unfairly in that a process P (see Fig. 4) which can diverge, *i.e.* $\mathsf{P} \stackrel{\tau}{\Longrightarrow} \mathsf{P}$, can prevent all other actions of P from being performed.

When interleaving and unfair choice are combined the divergence of P in $\mathsf{P} \parallel_\varnothing \mathsf{Q}$ is able to stop Q from performing any action. Had P deadlocked Q would continue. We will refer to this feature as **divergence leakage**. Divergence leaking can be regarded as

counter intuitive as my computer performing an unlimited internal chatter should not, we believe, affect an unrelated computer on your desk.

We extend Abs_o in order to define congruences with nonterminating components by replacing τ loops with nondeterminism. Because of " divergence leakage" we need to distinguish livelock from deadlock. This we do by including $x^* \xrightarrow{\tau} x^*$ in Definition 4.

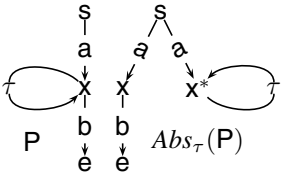


Fig. 4. Divergence

Process P in Fig. 4 is an example of what we call *optional divergence*. It could either act fairly and always eventually perform b or alternatively it could stay forever in the x state. What is more which it does is not determined by any outside agent. Hence we model this divergence as the nondeterministic choice between performing b and being trapped in the **divergent state** x^* (see Fig. 4).

Definition 4. *Let* $\mathsf{A} \stackrel{\text{def}}{=} (N_\mathsf{A}, Os_\mathsf{A}, T_\mathsf{A})$, $N_{\tau\mathsf{A}} = N_\mathsf{A} \cup \{x^* \mid x \stackrel{\tau}{\Longrightarrow} x\}$,
$\mathsf{A}^+ \stackrel{\text{def}}{=} (N_{\tau\mathsf{A}}, Os_\mathsf{A}, T_\mathsf{A} \cup \{n \xrightarrow{\mathsf{a}} x^* \mid \mathsf{a} \in Act_\tau \wedge x \stackrel{\tau}{\Longrightarrow} x \wedge n \xrightarrow{\mathsf{a}} x\})$,
$T_{\tau\mathsf{A}} \stackrel{\text{def}}{=} T_{Abs_o(\mathsf{A}^+)} \cup \{s^* \xrightarrow{\tau} s^* \mid s \stackrel{\tau}{\Longrightarrow} s\}$ *and*
$Os_{\tau\mathsf{A}}(s) \stackrel{\text{def}}{=}$ *if* $s \in N_\mathsf{A}$ *then* $Os_\mathsf{A}(s)$ *else* $Os_{\tau\mathsf{A}}(s^{(*^{-1})})$.
Then $Abs_\tau(\mathsf{A}) \stackrel{\text{def}}{=} (N_{\tau\mathsf{A}}, Os_{\tau\mathsf{A}}, T_{\tau\mathsf{A}})$ •

Theorem 1. $=_{\tau F}$ *and* $=_{\tau SF}$ *are congruent w.r.t.* $\{\sqcap, ;, \oplus, \|_\gamma, \delta_S, /_\tau S\}$.

3 Z Components

We will use Z schemas to define both operations and state. Unfortunately Z leaves as informal any attempt to localise state or action, so here we informally follow the convention of simply allowing schemas to be grouped together. This is the approach taken in Object-Z [7, 8].

We define a Z-Component to be $(State, SS, OP)$ where *State* is a state schema, *SS* a set of state schemas such that $init \in SS$ and $final \in SS$ and *OP* a set of named operation schemas (see Fig. 5 for an example).

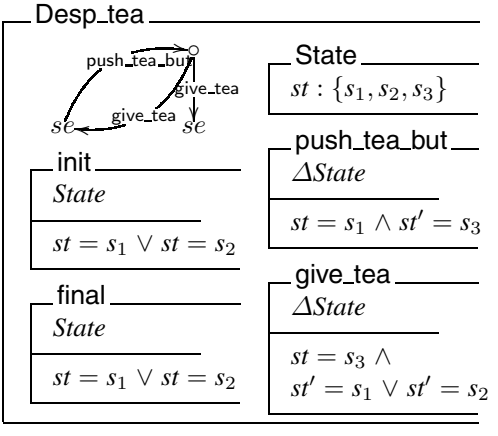


Fig. 5. Desp_tea

By restricting *State* to be a finite set of observations of *enumerated* type the normal evaluation function will map Z-Components to CTS. This is a small generalisation of LTS semantics of Z found in [7, 8, 24, 9].

Example. A vending machine accepts an electronic money card, then allows the user to request cups of tea if the card has sufficient funds, and finally to remove the card.

We formalise the vending machine using components: one - Insert_card; two - Desp_tea; and three - Remove_card. We then define the vending machine as the sequential composition of the three components.

We define Desp_tea Fig. 5 with two initial states, *i.e.* sufficient funds or insufficient funds. Our definition of sequential composition (Definition 2) then introduces the nondeterministic branching of the previous insert_card action.

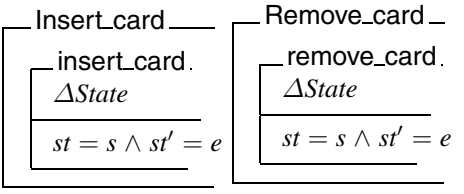


Fig. 6

This definition of Desp_tea would not be possible had we used the CCS, ACP or CSP semantics because, as we explain in Section 4.1, they equate processes with their root unwinding.

For brevity, in Fig. 6 and Fig. 7, we assume the existance of both *State*, defined as the obvious enumerated data type and init $\stackrel{\text{def}}{=}$ $[State \mid st = s]$ and final $\stackrel{\text{def}}{=}$ $[State \mid st = e]$.

The operational semantics of Insert_card; Desp_tea; Remove_card can be constructed by *evaluation* and then simplified by pruning unreachable operations and identifying bisimilar nodes, resulting in the CTS VM in Fig. 7. The Z text is constructed from this CTS by using as state a single observation of enumerated type that ranges over the set of nodes N_{VM}.

To verify that two cards cannot be inserted without an intervening removal of a card we can simplify VM (as all we are interested in are the actions insert_card and remove_card) by hiding the other actions.

Analysis by hiding is conceptually different from the hiding of "private communication". In analysis actions can be hidden that need not be observed, but can still be controlled. In CSP [18–p296] the hiding of "private communication" is modelled by *eager* abstraction and hiding in analysis is modelled by *lazy* abstraction. Because we do not model divergence as chaotic behaviour we can use the same definition of hiding in analysis as in the hiding of "private communication".

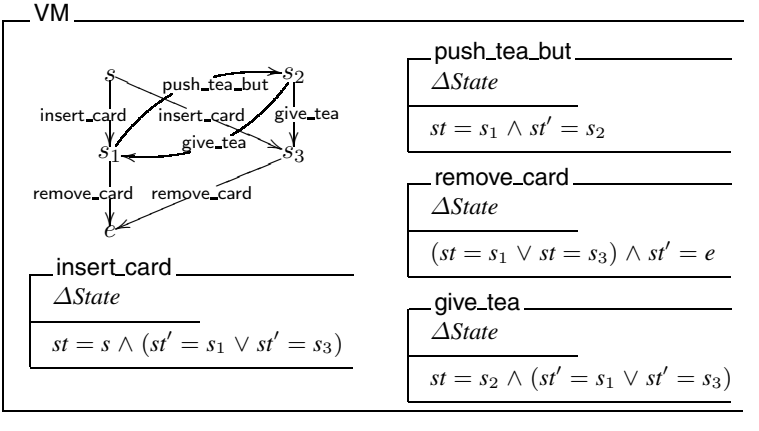


Fig. 7. VM

4 Comparison

Action-based approaches frequently use a single syntactic class (of actions) and use recursion to define nonterminating components, which are given a fixed point semantics. The need to have a unique fixed point semantics has had a strong influence on the semantics of CSP [18–p215] and unifying theories of programming [13–2.6, 2.7]. Alternatively, a very powerful argument, for state-based systems, has been made [25] in support of using refinement semantics rather than fixed point semantics.

When modelling State-and-Action systems it is natural to use two syntactic classes, one for states and one for actions. Using such a formalism, recursion and fixed points are not needed to define nonterminating components that have finite state and alphabet.

4.1 Choice

In CSP (but not CCS/ACP) τ actions $s \xrightarrow{\tau} n_2$ and $s \xrightarrow{\tau} n_1$ can model "the process could be in state n_1 or n_2 but we cannot know which"(see $\mathsf{a} \sqcap \mathsf{b}$ in Fig. 8). We interpret CSP external choice $\Box$ and ACP choice $+$ to be the same and use the different role of τ actions in CSP and ACP/CCS to explain why $\mathsf{A} \Box \mathsf{c} \neq \mathsf{A} + \mathsf{c}$ in Fig. 8.

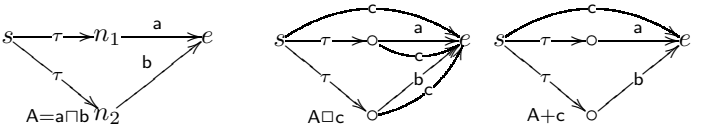


Fig. 8. CSP $\Box$ and ACP $+$

CSP, CFFD and NDFD all use the standard denotational approach to define congruence w.r.t. choice, *i.e. stability*, whereas CCS, ACP and this paper use the standard operational approach to define congruence, *i.e.* keep $s_{\mathsf{A}} \xrightarrow{\tau} \circ$. Although we believe this to be of little conceptual importance it does introduce small discrepancies in what would otherwise be the same equivalences.

The renaming of observable actions as τ actions, $\tau_{\{\mathsf{a}\}}$, does not distribute through CSP choice, whereas $\tau_{\{\mathsf{a}\}}$ does distribute through CCS/ACP choice and our $\oplus$.

Root Unwinding or Not. We have motivated our congruence by using distinguishing states that are in the interface from those that are not. This introduces a question: what happens if a process returns to one of the start states that is in the interface?

Choice as defined in [23, 15] is $+$ (see Fig. 9) and first *root unwinds* the LTS then identifies start states. Root unwinding allows us to view loops as mere "sugar" for their true meaning as an acyclic LTS.

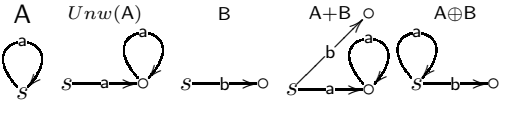


Fig. 9. Choice + or $\oplus$

Here choice is modelled by gluing together the root nodes of two CTSs without performing root unwinding. This is not new: it appears in [22] where such a definition of choice is given as limits in categories of labelled transition systems and Petri nets.

By changing the definition of choice, what is required of a congruence is changed. With the semantics in [22] $\mathsf{A} \neq \mathit{Unw}(\mathsf{A})$, as we would expect from our desire to distinguish states that are in the interface from those that are not.

4.2 Sequential Composition

Sequential composition is defined using an explicit representation of the successful *termination* of a process. In CSP termination *SKIP* "can always be chosen" when offered, *i.e.* $\mathit{SKIP}\Box\mathsf{a} \rightarrow \mathit{STOP} = (\mathit{SKIP}\Box\mathsf{a} \rightarrow \mathit{STOP}) \sqcap \mathit{SKIP}$. This is quite different from ACP termination ϵ which cannot always be chosen. This can be seen in the construction of $((\mathsf{a};\epsilon) + \epsilon) \parallel (\mathsf{a};\epsilon)$ (see [15] [p. 76]) where the ability of one of the components to initially terminate is simply lost. Because of these differences we will avoid comparing congruence w.r.t. sequential composition from CSP and our definition which follows that of ACP.

4.3 NDFD Divergence Without Chaos

In [19, 23][1] they construct a denotational semantics without interpreting divergence as chaos. Stability *sta* and divergence *div* are defined on the strong operational semantics. Failure semantics is defined on the observational semantics ($\Rightarrow_o$) and finally stability, divergence and failures are all used in the definition of *NDFD*.

Let A and B be CTSs.

$\mathit{sta}(\mathsf{A}) \stackrel{\text{def}}{=} \forall_{s \in s_\mathsf{A}}\, s \stackrel{\tau}{\not\longrightarrow}$ and

$\mathit{div}(\mathsf{A}) \stackrel{\text{def}}{=} \{\theta \mid s \stackrel{\theta}{\Longrightarrow}_o n \wedge s \in s_\mathsf{A} \wedge n \stackrel{\tau\tau^*}{\longrightarrow} n\}$

$\mathit{fail}(\mathsf{A}) \stackrel{\text{def}}{=} \{(\theta, X) \mid \exists_n\, s \stackrel{\theta}{\Longrightarrow}_o n \wedge s \in s_\mathsf{A} \wedge \forall_{\mathsf{x} \in X}\, n \stackrel{\mathsf{x}}{\not\Longrightarrow}_o\}$

$\mathit{dfail}(\mathsf{A}) \stackrel{\text{def}}{=} \{(\theta, X) \mid (\theta, X) \in \mathit{fail}(\mathsf{A}) \vee \theta \in \mathit{div}(\mathsf{A})\}$

$\mathsf{A} =_{\mathit{NDFD}} \mathsf{B} \stackrel{\text{def}}{=} \mathit{sta}(\mathsf{A}) = \mathit{sta}(\mathsf{B}) \wedge \mathit{dfail}(\mathsf{A}) = \mathit{dfail}(\mathsf{B}) \wedge \mathit{div}(\mathsf{A}) = \mathit{div}(\mathsf{B})$

A CTS is *well-terminating* if $n \stackrel{a}{\longrightarrow}$ implies $n \notin e_\mathsf{A}$.

Lemma 1. *For stable, unwound and well-terminating processes.*

$$\mathsf{A} =_{\tau F} \mathsf{B} \Leftrightarrow \mathsf{A} =_{\mathit{NDFD}} \mathsf{B}$$

We have provided an operational interpretation of action abstraction that transforms divergence into nondeterminism. The above result tells us that computing failure equivalence on the observational semantics gives the same result as computing NDFD equivalence on the strong semantics. The restriction to stable, unwound processes is explained in Section 4.1 and the restriction to well-terminating processes is explained in Section 4.2.

5 Conclusion

To model components we give an equal status to states and events. We require that our components have an interface and that components with the same interface are

[1] We do not need traces as we consider only finite state processes.

"observationally" equivalent. In particular we allow both states and events to be a part of the interface. To this end we use two syntactic classes, one for states and one for events, and consequently we do not need recursion or unique fixed points to define the semantics of nonterminating components that have finite state and alphabet.

In order to preserve a component's interface we reject the root unwinding built into the semantics of many process algebras. In this regard our approach is based on that of Winskel and Nielson [22]. We demonstrate some practical advantages of this approach in Section 3.

To take advantage of the well-known [7, 8, 9] isomorphism between state-based relational semantics and event-based operational semantics we use operational rather than denotational semantics. There is a natural way of defining observational equivalences in two steps: first, apply abstraction to build an observational semantics from the strong semantics; then, apply a strong equivalence to the newly built observational semantics Section 2.3. We take advantage of this and define abstraction that models what can be observed of divergent processes.

In [12] they define a singleton failures semantics for ADTs but hiding has to be restricted to exclude the possibility of considering divergent ADTs. We can extend this work to consider nonterminating processes by first applying our definition of abstraction and then applying their definitions to the resulting observational semantics. This results in $=_{\tau SF}$, a singleton version of NDFD equivalence.

The work in [9] gives testing characterisations that "explain" the difference between several known refinements including LOTOS's extension [26], conformance [27], may and must testing [11], failure refinement and singleton failure refinement [12]. But all these refinements ignore divergence and hence, if we require congruence with respect to our operators, can only by applied to terminating processes. We can construct the observational semantics defined here and subsequently apply the work in [9] to the observational semantics. This extends the original work to cover nonterminating processes where divergence is not ignored.

References

1. Barnett, M., Schulte, W.: The ABCs of Specification: AsmL, Behavior, and Components. Informatica **25** (2001)
2. Bolognesi, T., Börger, E.: Abstract State Processes. In Börger, E., Gargantini, A., Riccobene, E., eds.: Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, Proceedings. Volume 2589 of Lecture Notes in Computer Science. (2003) 218–228
3. Hansen, H., Virtanen, H., Valmari, A.: Merging state-based and action-based verification. In: Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03), Guimarães, Portugal, IEEE Computer Society (2003)
4. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proceedings of IEEE **77(1)** (1989) 81–98
5. Heymann, M., Meyer, G.: Algebra of discrete event processes (1991)
6. Heymann, M.: Concurrency and Discrete Event Control. IEEE Control Systems Magazine **10** (1990) 103–112

7. Smith, G.: A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In Fitzgerald, J., Jones, C.B., Lucas, P., eds.: FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997). Volume 1313., Springer-Verlag (1997) 62–81
8. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Formal Approaches to Computing and Information Technology. Springer (2001)
9. Reeves, S., Streader, D.: Comparison of Data and Process Refinement. In Dong, J.S., Woodcock, J.C.P., eds.: ICFEM 2003. LNCS 2885. Springer-Verlag (2003) 266–285
10. de Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science **34** (84)
11. Hennessy, M.: Algebraic Theory of Processes. The MIT Press (1988)
12. Bolton, C., Davies, J.: A singleton failures semantics for Communicating Sequential Processes. Research Report PRG-RR-01-11, Oxford University Computing Laboratory (2001)
13. Hoare, C., Jifeng, H.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
14. Milner, R.: Communication and Concurrency. Prentice-Hall International (1989)
15. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge Tracts in Theoretical Computer Science 18 (1990)
16. Kaivola, R., Valmari, A.: The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear temporal logic. In: International Conference on Concurrency Theory. (1992) 207–221
17. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
18. Roscoe, A.: The Theory and Practice of Concurrency. Prentice Hall International Series in Computer Science (1997)
19. Valmari, A., Tienari, M.: An improved failure equivalence for finite-state systems with a reduction algorithm. In: Protocol Specification, Testing and Verification. IFIP XI, North-Holland (1991)
20. Smith, G.: A Fully Abstract Semantics of Classes for Object-Z. Formal Aspects of Computing **7** (1995) 289–313
21. Woodcock, J.C.P., Cavalcanti, A.L.C.: The Semantics of Circus. In Didier Ber, Jonathan P. Bowen, M.C.H., Robinson, K., eds.: ZB 2002 Formal Specification and Development in Z and B LNCS 2272. Springer-Verlag (2002) 184–203
22. Winskel, G., Nielsen, M.: Models for concurrency. Technical Report DAIMI PB 429, Computer Science Dept. Aarhus University (1992)
23. Valmari, A., Tienari, M.: Compositional Failure-based Semantics Models for Basic LOTOS. Formal Aspects of Computing **7** (1995) 440–468
24. Reeves, S., Streader, D.: State-based and process-based value passing. In: Proceedings of St.Eve @ FM'03. (2003) Available at www.cs.waikato.ac.nz/~stever/06-Reeves-Streader.pdf.
25. Hehner, E.C.R., Gravell, A.M.: Refinement semantics and loop rules. In: World Congress on Formal Methods (2). (1999) 1497–1510
26. Brinksma, E., Scollo, G.: Formal notions of implementation and conformance in LOTOS. Technical Report INF-86-13, Twente University of Technology, Department of Informatics, Enschede, The Netherlands (1986)
27. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS specifications, their implementation and their tests. In Sarikaya, B., Bochmann, G.V., eds.: Protocol Specification, Testing and Verification. Volume VI., North-Holland (1986) 349–360

Towards an Optimization-Based Method for Consolidating Domain Variabilities in Domain-Specific Web Services Composition

Jun-Feng Zhao, Lu Zhang, Ya-Sha Wang, Ying Jiang, and Bing Xie

Software Institute, School of Electronics Engineering and Computer Science,
Peking University, Beijing, 100871, P. R. China
{zhaojf, zhanglu, wangys, jiangy, xiebing}@sei.pku.edu.cn

Abstract. In this paper, we put forward an automatic method of acquiring the specific system composition model from a domain composition model and requirements for the specific system in domain-specific Web services composition. This is referred to as the variability consolidation problem in this paper. To achieve this goal, we designed a language to describe domain properties for Web services composition. The basis of our approach is to transform the domain composition model and the requirements for the specific system into a mathematical optimization problem, which can be solved by existing algorithms. Thus, this method is fully automatic and not prone to human errors. Our preliminary experimental results show that our method is quite feasible for solving problems with real world sizes.

1 Introduction

In recent years, using Web services to construct new applications has become an emerging paradigm of integrating Web applications across the Internet [2][15][19]. A Web service is a software application identified by a URI, whose interface and bindings are capable of being identified and discovered by XML artifacts and support direct interactions with other software applications using XML-based messages via Internet-based protocols [3]. From the perspective of software reuse, Web services composition can be viewed as an Internet version of component-based software development [18].

As Web services are usually prone to interruptions on the Internet, the non-functional properties of Web services (which are often referred to as quality of service in the literature on Web services [12]) have to be considered very seriously in the composition. A common way of dealing with this problem is to employ several services fulfilling the same functionality but with different quality of service (QoS) as candidates competing for one place in the composed application [22]. Therefore, Web services composition also includes the selection among the candidate Web services. Actually, researchers outside the software engineering community (e.g. the authors of [22]) may even view Web services composition merely as candidate Web services selection.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 140–153, 2005.
© Springer-Verlag Berlin Heidelberg 2005

According to the literature, component reuse is typically beneficial when it is confined to a special domain [13][16]. As Web services can be viewed as a kind of reusable components, it should also be preferable to reuse Web services within a particular domain. In this paper, our interest focuses on domain-specific Web services composition, in which the composition can be guided by a domain composition model containing domain variabilities. Thus, an important problem in this kind of composition is to consolidate the variabilities and/or select the proper candidate services in the domain composition model according to the functional and non-functional requirements for constructing a particular system in the domain.

Traditionally, the consolidation of variabilities in domain-specific composition is performed by developers according to the requirements with the help of domain specialists and system analysts. Although this does not seem to be a challenging task, the manual nature may make it tedious and prone to errors, especially when there are many candidate services and many constraints to be considered. For example, if the system consists of ten types of services to be integrated, and each service has 10 candidates, then there should be 10^{10} different ways to compose the system. Obviously, it is impossible for developers to achieve the optimal or sub-optimal composition via considering all these possibilities. Therefore, it is in need of introducing an automatic or automated method to help developers to decide which Web services is the proper services.

In this paper, we propose a method to solve the above problem using mathematical optimization (please refer to [8][14] for information on mathematical optimization). In our approach, we use a way similar to the domain engineering approach to describe domain variabilities in the domain composition model. In the model, Web services selection can be represented as a special case of variability consolidation. Based on the domain composition model, we can formalize the above problem as an optimization problem, which can be solved using existing algorithms.

The remainder of this paper is organized as follows. Section 2 presents some preliminary knowledge used in this paper. Section 3 presents a language for describing domain variabilities in the domain composition model. In section 4, we propose our method to consolidate domain variabilities for constructing a particular system in the domain. In section 5, we present some empirical results on the feasibility of our method. Section 6 discusses some related research, and section 7 concludes this paper.

2 Preliminaries

2.1 QoS of Web Services

The nature of Web services determines the importance of quality of service (QoS) for both constituent Web services and the application composed of Web services. In the literature, there have been quite a few papers discussing quality-related issues of Web services (see e.g. [12][17]). In the following, we just list some

frequently discussed quality attributes for Web services: reliability, availability, execution time, and cost.

• **Reliability**

Reliability is an overall measure of a Web service to maintain its service quality. In this paper, we adopt the definition that the reliability of a Web service is the probability that the service responds correctly. In practice, we can use Time-to-Fails (TTF) to measure the reliability of the service. TTF means the running Web service's mean time to fail.

• **Availability**

Availability defines the extent to which a Web service is ready for immediate consumption. In this paper, we adopt the definition that the availability is the probability that the service is accessible. Associated with availability is Time-to-Repair (TTR). TTR represents the time it takes to repair the Web Service.

• **Execution Time**

The execution time is the time taken by a Web Service to finish its task. It can be measured by the average response time of a certain Web service.

• **Cost**

The cost of a Web service is the amount of money that a service requests or has to be paid for executing the service.

The QoS affects Web services composition in the following way. When using Web services to compose a new application, we may have some non-functional or quality requirements for the overall application. Therefore, we should select the Web services with proper quality attributes and determine the proper interconnections of the selected services to achieve a system that satisfies the target quality requirements.

2.2 Domain Engineering

Domain engineering (see e.g. [13][16]) is aiming at systematically managing variability within a domain. In a typical domain engineering process, several existing systems in the domain are analyzed to acquire a domain composition model containing some variabilities, which is usually termed as the domain-specific software architecture (DSSA) in the domain engineering community. When a new task of constructing a new system in the domain arises, the domain composition model is customized into a composition model for the particular system. As the main task in the customization is to consolidate the variabilities contained in the domain composition model, we refer to this customization process as domain variability consolidation in this paper.

In a domain composition model, domain variability is usually described as the characteristics of the constituents in the model and the relationships between the constituents [6][9][11][20][24]. There are mainly three characteristics identified in the literature:

• **Mandatory**

Mandatory constituents embody the essence and/or common requirements within the domain. When consolidating the variabilities, all the mandatory constituents should be included in the composition model for the specific system.

• **Optional**

Optional constituents are those that appear in some systems in the domain, but are not required in all the systems. During variability consolidation, we should determine which optional constituents would be included in the specific system.

• **Alternative**

Alternative constituents are a set of optional constituents that satisfy the following condition during variability consolidation. Any system in the domain should include one and only one of these constituents. Actually, the relationship between a set of alternative constituents is the mutually exclusive relationship discussed below.

The relationships between constituents identified in the literature are listed below:

• **Dependent**

Constituent p is said to be dependent on constituent q, if and only if any system in the domain containing p should also contain q. Therefore, we should avoid including p without including q in the specific system when consolidating the variabilities.

• **Mutually Exclusive**

Two constituents are said to be mutually exclusive, if and only if the two constituents cannot both exist in any system in the domain. Obviously, any two constituents among a set of alternative constituents are mutually exclusive.

Given a domain composition model and the requirements for a particular system in the domain, variability consolidation is to produce the system's composition model satisfying both the variability constraints described in the domain composition model and the system requirements. When the domain composition model is small and/or the variability constraints are simple, this variability consolidation can be fulfilled by developers manually. However, this manual consolidation can be quite time-consuming and prone to errors in a complex domain. The situation could be even worse when we are consolidating variabilities for composition of Web services, because the services selection problem discussed above is intertwined with the variability consolidation problem. Below, we will present a method to automatically select Web services and consolidate domain variability for domain-specific Web services composition.

3 Describing Domain Properties for Web Services Composition

In a commercial Web services composition language (such as [1][10]), the primitive composition unit is usually an operation within a Web service. For the simplicity of presentation, we treat Web services as primitive units in this paper. Therefore, a Web service in this paper is corresponding to an operation in a commercial language.

In order to describe the domain properties in a domain composition model, we define the Domain Specific Web Services Composition Language (DSWSCL)

⟨DomainProperties⟩::=⟨ServiceProperties⟩⟨RelationProperties⟩

⟨ServicePropeties⟩::={⟨SingleServiceProperties⟩}{⟨AlternativeServicesProperties⟩}
⟨SingleServiceProperties⟩::=SERVICE ⟨ServiceID⟩⟨DomainProperty⟩
⟨DomainProperty⟩::=MANDATORY|OPTIONAL_THROUGH|OPTIONAL_IGNORANCE

⟨AlternativeServicesProperties⟩::=ALTERNATIVE ⟨ServiceID⟩, ⟨ServiceID⟩ {,⟨ServiceID⟩}
⟨RelationProperties⟩::={⟨DependentRelation⟩|⟨MutexRelation⟩}
⟨DependentRelation⟩::=⟨ServiceID⟩ DEPENDS_ON ⟨ServiceID⟩
⟨MutexRelation⟩::=MUTEX ⟨ServiceID⟩, ⟨ServiceID⟩

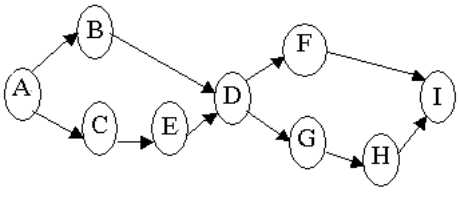
Fig. 1. Definition of DSWSCL

to work with a Web services composition language. Therefore, a domain composition model includes two parts: the domain part and the composition part. The domain part is written in DSWSCL, containing all the domain commonalities and variabilities. The composition part is written in a Web services composition language, containing all the possible Web services and all the possible links. Here a link refers to the order in which the linked two services will have to be performed. Please note that the composition part here is usually not a valid composition model, and some consolidation according to the domain part is needed to obtain a valid composition model from the composition part. The definition of the DSWSCL is depicted in Fig.1.

The description of the domain properties includes that of the Web services and that of the relations between Web services. The description of Web services indicates whether a Web service is mandatory, optional, or alternative. We distinguish two types of optional Web services. The first type includes those Web services whose deselection means that the inputs will be directly connected to the outputs. This situation is identified by the reserved word OPTIONAL_THROUGH. The second type includes those Web services whose deselection means that all the inputs and outputs will be ignored. This situation is identified by the reserved word OPTIONAL_IGNORANCE.

SERVICE *A* MANDATORY
SERVICE *D* MANDATORY
SERVICE *I* MANDATORY
SERVICE *B* OPTIONAL_IGNORANCE
SERVICE *C* OPTIONAL_ THROUGH
SERVICE *E* OPTIONAL_IGNORANCE
SERVICE *F* OPTIONAL_IGNORANCE
SERVICE *G* OPTIONAL_IGNORANCE
SERVICE *H* OPTIONAL_IGNORANCE
H DEPENDS_ON *G*
MUTEX *B, C*
MUTEX *F, G*

(a)



(b)

Fig. 2. An example of the domain properties in DSWSCL

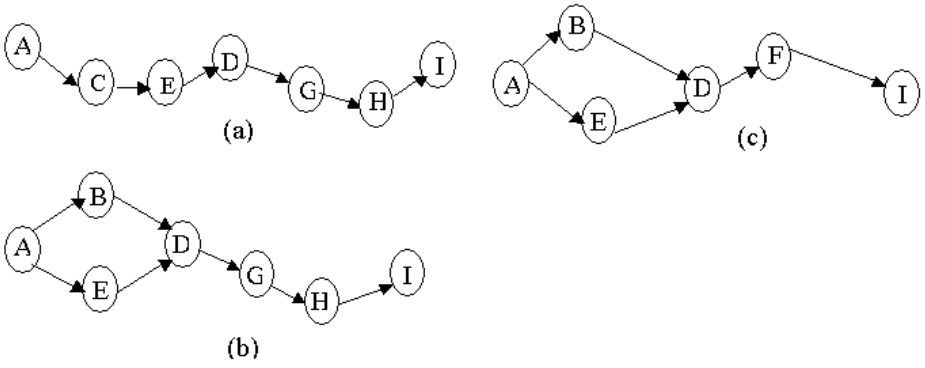


Fig. 3. An example of domain variability consolidation

For example, we have a domain model that includes nine services: A, B, C, D, E, F, G, H and I. Services A, D and I are mandatory, while services B, C, E, F, G and H are optional. The relationship between B and C and that between F and G are mutually exclusive. The properties of B, E, F G and H are OPTIONAL_IGNORANCE and the property of C is OPTIONAL_THROUGH. Service H is dependent on service G. We can describe the domain properties in Fig. 2(a).

Supposing the possible Web services and their links (we use a circle to represent the service and a directed edge to show the order between two services) are depicted in Fig. 2(b), three possible specific composition models are depicted in (a), (b) and (c) of Fig. 3. From the above constraints, B and C cannot be both selected, F and G cannot be both selected, and the selection of H implies the selection of G. Fig. 3(a) represents the selection of C, E, G and H; Fig. 3(b) represents the selection of B, E, G and H; and Fig. 3(c) represents the selection of B, E and F. All the three satisfy the domain constraints.

4 The Method

The basic idea of our approach is to formalize the variability consolidation and/or the Web services selection as a mathematical optimization problem, which has been studied for quite a long time and can be solved using existing algorithms. To treat variability consolidation and Web services selection in a uniform way, we transform the latter into the former before formalization. Thus, the central part of our method focuses on how to turn the variability consolidation problem into a form of optimizing a subjective function under certain constraints.

4.1 Transforming Web Services Selection to Variability Consolidation

To simplify the formalization, we treat Web services selection as a special case of variability consolidation. For the Web services selection problem, we have a

set of candidate services competing for one place in the composed system. This situation satisfies the condition for a set of alternative Web services. Therefore, we can treat a set of competing candidate Web services as a set of alternative Web services, and thus we can solve the Web services selection problem in the same way of solving the variability consolidation problem.

4.2 Formalizing Variability Consolidation as Optimization

Given a set of variables (denoted as x_1, x_2, $\ldots x_n$), a set of constraints on the values of a set function of the variables and a subjective function of the variables, an optimization problem is to find a set of values for the variables that satisfy the constraints and maximize or minimize the subjective function. In our method, we use one variable to represent the selection of one corresponding Web service. Therefore, if there are n Web services in the domain composition model, there will be n selection variables in the formalized optimization problem. Each variable can be of the value 0 (representing deselection) or 1 (representing selection).

The variabilities described in the domain composition model will be formalized as a set of constraints (which are referred to as domain constraints). The functional and non-functional requirements for the specific system will also be formalized as a set of constraints (which are referred to as requirement constraints). To obtain the subject function, we can transform one requirement constraint into the subject function. For example, we can transform the constraint on the cost of the composed system into the form of minimizing the cost. Thus, a solution of the optimization problem (denoted as a set of values for the variables) represents the consolidation of the domain variabilities.

In the following, we assume that there are totally n Web services in the domain composition model. We use $WS_{total} = \{1, 2, \ldots n\}$ to denote the set of all the n Web services, and we use i to denote the ith Web services in WS_{total} and x_i denote its corresponding selection variable ($i \in WS_{total}$). A variable in the formalized optimization problem is denoted as a lower case letter with possible subscripts and a constant is denoted as an upper case letter with possible subscripts.

4.2.1 Formalizing Domain Constraints

For each mechanism of describing domain variability discussed above, the corresponding formalization is as follows.

- **Mandatory Web Services**

Supposing the set of mandatory Web services in the domain composition model is WS_{man}, for each element in the set, the equation in (1) will be added as a constraint. This equation can ensure that the Web service appear in the composition model for the target system.

$$x_i = 1(i \in WS_{man}) \tag{1}$$

• **Optional Web Services**

As the value of the corresponding selection variable for a Web service can deter-mine whether the Web service will appear in the composition model for the target system, simply no constraint will ensure the Web service to be optional.

• **Alternative Web Services**

For a set of alternative Web services WS_{alt}, the equation in (2) can ensure one and only one of them will be selected into the composition model for the target system.

$$\sum_{i \in WS_{alt}} x_i = 1 \tag{2}$$

• **Dependent Relationships**

For two Web services p and q, supposing p is dependent on q, and the corresponding variable of p is x_p and that of q is x_q, the inequity in (3) can ensure that if p is selected, q will also be selected. In inequity (3), if the value of x_p is 1, then the value of x_q have to be 1 to satisfy the inequity.

$$x_p - x_q \leq 0 \tag{3}$$

• **Mutually Exclusive Relationships**

For two mutually exclusive Web services p and q, supposing the corresponding variable of p is x_p and that of q is x_q, the inequity in (4) can ensure that not both services be selected. As this relationship allows the situation in which neither of the two services is selected, we should use an inequity rather than an equation similar to that in (2).

$$x_p + x_q \leq 1 \tag{4}$$

4.2.2 Formalizing Requirement Constraints

When constructing a specific system based on the domain composition model, the construction will also follow some functional and non-functional requirements. In our method, theses requirements will be formalized as constraints in the optimization problem.

• **Functional Requirements**

Firstly, functional requirements in this domain-specific Web services composition may be represented as requiring some optional and/or alternative services to be included in the target system composition model. This means that those services will be treated as mandatory in the composition. Therefore, some more constraints like those in (1) will also be added.

Secondly, when there is some freedom of choosing services, we may want the selected services to fulfill as much functionality as possible. In such a case, we can assign a value of functionality to each service, and require the total value of functionality to be higher than a threshold. The value we assign to each Web Service is depended on a subjective way. We make a questionnaire and let the user give the score of functionality satisfaction of each service. The mean value

of these scores is the value of functionality of each service. Supposing F_i is the corresponding value of functionality for Web service i ($i \in WS_{total}$), and F_{total} is the threshold value of total functionality, the requirements of total functionality can be represented as the constraint in (5).

$$\sum_{i \in WS_{total}} F_i X_i \geq F_{total} \tag{5}$$

• **Non-functional Requirements**

There may also be non-functional requirements on the QoS of the composed system. In the following, we demonstrate how these requirements on the above-discussed QoS factors can be transformed into constraints. Similar ways of transforming non-functional requirements into constraints can be found in [22]. We believe that this kind of transformation can be extended to support requirements on other QoS factors discussed in the literature.

Firstly, the requirements on the total cost can be transformed into a constraint as follows. Supposing the cost for Web service i is C_i ($i \in WS_{total}$), and the requirement on the total cost is no greater than C_{total}, the constraint on the total cost can be formalized in (6).

$$\sum_{i \in WS_{total}} C_i X_i \leq C_{total} \tag{6}$$

Secondly, the transformation for the requirement on the maximum execution time is as follows. In this transformation, we introduce two new variables (s_i and e_i) for Web service i representing the starting time and the ending time of the service ($i \in WS_{total}$). Note that the values of variables s_i and e_i are not confined to 0 and 1, but any positive real numbers. The formalization is according to the type of the Web service.

For a set of alternative Web services WS_{alt}, these services will share the same proceeding services (denoted as WS_{pro}) and succeeding services (denoted as WS_{suc}). Sup-posing the execution time of service i is Ti ($i \in WS_{alt}$), the constraints can be represented in (7), (8) and (9). Constraints in (7) ensure that any service in WS_{alt} will start after the ending of any proceeding services. Similarly, constraints in (8) ensure that any succeeding services will start after the ending of any service in WS_{alt}. In (9), T is a sufficient large number, which ensures that each constraint in (9) is effective only when the corresponding Web service i is selected. In such a case, the effective constraint ensures that there is enough time for the selected Web service to be executed.

$$s_i \geq e_j (i \in WS_{alt}, j \in WS_{pro}) \tag{7}$$

$$e_i \leq s_k (i \in WS_{alt}, k \in WS_{suc}) \tag{8}$$

$$T_i x_i \leq (e_i - s_i) + T(1 - x_i)(i \in WS_{alt}) \tag{9}$$

For an optional Web service o, the proceeding services and the succeeding services of o are denoted as WS_{pro} and WS_{suc} respectively.

If the deselection of o represents directly connecting its input flows to its output flows, Web service o is actually an alternative service competing with a service with no execution time. Thus, this situation can be formalized the same as the formalization for alternative Web services.

If the deselection of o represents the ignorance of its input flows and output flows, the formalization is denoted in (10), (11) and (12). Constraints in (10) and (11) ensure the starting time of o is no earlier than the ending time of any its proceeding services, and its ending time no later than the starting time of any its succeeding services. In (12), To is the execution time of o, and T is a sufficient large number. Therefore, if Web service o is selected (i.e. $x_o = 1$), the constraint in (12) ensures that there is enough time for o to be executed, and if o is deselected (i.e. $x_o = 0$), the constraint in (12) actually represents no constraint at all.

$$s_o \geq e_j (j \in WS_{pro}) \tag{10}$$

$$e_o \leq s_k (k \in WS_{suc}) \tag{11}$$

$$e_o - s_o \geq T_o - T(1 - x_o) \tag{12}$$

Supposing the Web services connecting to the starting point are WS_{start}, and the Web services connecting to the ending point are WS_{end}, we also need the constraints in (13), (14) and (15), where t_{actual} is the actual execution time of the composed Web service, and T_{total} is the maximum allowed execution time for the composed Web service. In fact, the introduction of tactual is for the ease of transforming the constraint on execution time to the subjective function.

$$s_i \geq 0 (i \in WS_{start}) \tag{13}$$

$$e_j \leq t_{actual} (j \in WS_{end}) \tag{14}$$

$$t_{actual} \leq T_{total} \tag{15}$$

Thirdly, as the reliability is modelled as the probability of correct responding, the incorrectness of any selected Web services will result in the incorrectness of the composed Web service. Therefore, supposing the reliability of Web service i is R_i, the reliability of the composed Web service is denoted in (16), and the constraint on reliability is denoted in (17), where R_{total} is the minimum reliability.

$$\prod_{i \in WS_{total}} {R_i}^{x_i} \tag{16}$$

$$\sum_{i \in WS_{total}} ln(R_i) x_i \geq ln(R_{total}) \tag{17}$$

Fourthly, the transformation of the requirements on the total availability is similar to that of reliability. Supposing the availability of Web service i is Ai and the minimum availability is A_{total}, the constraint is in (18).

$$\sum_{i \in WS_{total}} ln(A_i)x_i \geq ln(A_{total}) \tag{18}$$

4.2.3 Determining the Subjective Function

We can transform one of the constraints in (5), (6), (15), (17) and (18) into the subjective function. Therefore, the candidate subjective functions are in (19), (20), (21), (22) and (23) respectively.

$$\max(\sum_{i \in WS_{total}} F_i x_i) \tag{19}$$

$$\min(\sum_{i \in WS_{total}} C_i x_i) \tag{20}$$

$$\min(t_{actual}) \tag{21}$$

$$\max(\sum_{i \in WS_{total}} ln(R_i)x_i) \tag{22}$$

$$\max(\sum_{i \in WS_{total}} ln(A_i)x_i) \tag{23}$$

4.3 Solving the Optimization Problem

The problem formalized above is actually a special case of a mixed integer linear programming problem. Therefore, we can adopt existing algorithms to solve the problem. In this paper, we adopt the branch and bound method to solve this problem (see [8][14] for details on this issue).

4.4 Generating the System Composition Model

After we acquire a solution of the formalized optimization problem, we can use the result to generate the system composition model from the domain composition model. This generation is actually to eliminate those deselected Web services and corresponding links from the composition part of the domain composition model. As there are tags in a Web services composition language, it is not difficult to find the targets for elimination.

5 Preliminary Empirical Results on Complexity

As algorithms for mix integer linear programming problems could be exponential in the worst case, we conduct an experiment to see whether our method is feasible for problems of the size the same with real problems.

For a given number of services contained in the system and a given number of average candidates for each service, we generate a domain composition model with sparse domain constraints. Then we generate the mix integer linear programming problem with some random domain requirements as input. Finally, we use the branch and bound method to solve this problem. The entire experiment is conducted on a PC with a 2GHz Pentium 4 processor and 512M RAM.

The results of our experiment are depicted in Fig. 4, in which, (a) depicts the number of iterations and (b) depicts the execution time for solving each optimization problem. In both (a) and (b), the x-coordinate represents the number of services that a target system contains, and the y-coordinate is the average number of iterations or the average execution time in milliseconds. Different lines represent different average numbers of candidates for each service. From Fig. 4, when the number of services is 40 and each service has 12 candidates, the execution time is still less than 25 seconds. Actually this size is larger than many current Web services composition problems. Although our experiment is still preliminary, the advantage of our approach is obvious compared to the tedious manual variability consolidation.

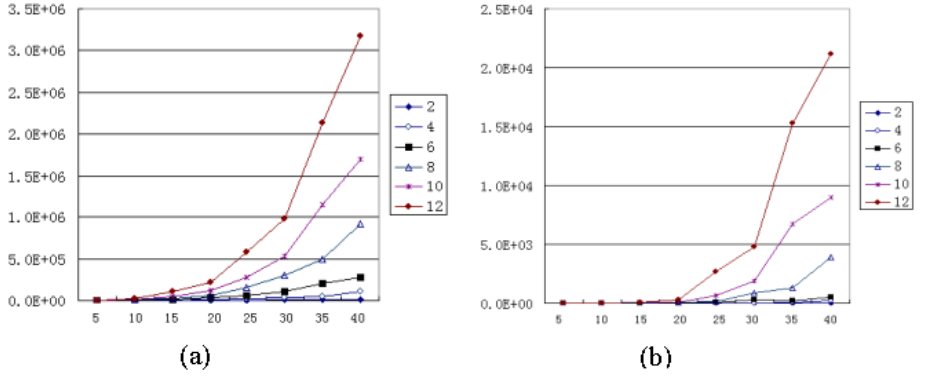


Fig. 4. Performance of our approach

6 Related Work

Web services composition has become a very active research area. There are several languages for describing composition models for Web services, including WSFL [10], BPEL4WS [1]. These works are actually foundations for our approach and other related works. The works most related to ours are those on

addressing service selection in Web services composition, such as eFlow [5], METEOR [4] and SELF-SERV [22][23]. The eFlow approach is put forward by HP lab, which focuses on optimizing service selection at the task level. In this approach, the selection is mainly based on execution time and budget without considering other QoS factors. METEOR has more considerations for QoS factors, but it concentrates on analyzing, predicting, and monitoring QoS of workflow processes. The SELF-SERV approach concentrates on service selection based on QoS factors. While eFlow and METEOR use local optimization strategies for service selection, SELF-SERV adopts the global optimization strategy. In this paper, we extend the idea in SELF-SERV to tackle the problem of variability consolidation in domain engineering. As a result, our approach can automatically calculate system models rather than mere selecting among functionally identical services.

7 Conclusions and Future Work

Composing Web services to form new applications on the Internet provides challenges for existing component-based approaches, as the QoS issues can play an important role in the composition. In this paper, we propose an approach to consolidating variabilities within the domain-specific Web service composition model. The central idea of our method is to formalize the variability consolidation problem as a mix integer linear programming problem and apply existing algorithms to solve it. Our preliminary results show that our approach is obviously superior over manual variability consolidation.

The work reported in this paper is still an ongoing one. In the future, we will conduct more experiments, especially those with real world background, to further evaluate our approach. Furthermore, we plan to extend our method to tackle traditional component composition and/or composition with less domain knowledge.

Acknowledgements. The work is supported by the National High-Tech Research and Development Plan (No. 2001AA113070) of China and the National Grand Fundamental Research 973 Program of China (No. 2002CB312003).

References

1. Andrews T., Curbera F., Dholakia H., Goland Y.: Specification: Business Process Execution Language for Web Services Version 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/. (2003)
2. Aoyama, M., Weerawarana, S., Maruyama, H., Szyperski, C., Sullivan, K., and Lea, D.:Web Services Engineering: Promises and Challenges.Proceedings of 24th International Conference on Software Engineering. (2002) 647-648
3. Austin D., Barbir A., Garg S.: Web Services Architecture Requirements. http://www.w3.org/TR/2002/WD-wsa-reqs-20020429. (2002)
4. Cardoso J.: Quality of Service and Semantic Composition of Workflows.Ph.D. Thesis,University of Georgia.(2002)

5. Casati F.Ilnicki S., Jin L.-J., Krishnamoorthy V., Shan M.C.: eFlow: a Platform for Developing and Managing Composite e-Services.Technical Report HPL-2000-36, HP Laboratories, Palo Alto. (2000)
6. Chen Z.L.: Research on Domain Application Variability Control Mechanism and Technology(in Chinese).Ph.D. Thesis, Peking University. (2003)
7. Frakes W., Prieto-Diaz R., and Fox E.: Domain Analysis and Reuse Environment. Annals of Software Engineering. **5** (1998)125-141
8. Harvey M. Salkin and Kamlesh Mathur.: Foundations of Integer Programming. North-Holland.(1989)
9. Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study (CMU/SEI-90-TR-021). Software Engineering Institute, CMU. (1990)
10. Leymann F.: Web services Flow Language (WSFL 1.0). http://www-106.ibm.com/developerworks/webservices. (2001)
11. Li K.Q.: Research on Object Oriented Domain Engineering Method. Ph.D. Thesis, Peking University. (2001)
12. Mani A., and Nagarajan A.: Understanding Quality of Service for web services. http://www-106.ibm.com/developerworks/webservices/library/ws-quality.html. (2002)
13. Mili H., Mili F. and Mili A.: Reusing Software: Issues and Research Directions. IEEE Transactions on Software Engineering. **21** (1995) 528-562
14. Nemhauser G. L. and Wolsey L. A.: Integer and Combinatorial Optimization. John Wiley and Sons, New York. (1988)
15. Pahl, C. and Casey, M.: Ontology Support for Web Service Processes. Proceeding of 11th ACM Symposium on Foundations of Software Engineering, Finland.(2003)208-216
16. Prieto-Diaz R: DDomain Analysis for Reusability. Proceedings of COMPSAC'87, Tokyo, Japan. (1987)23-29
17. Rajesh S., Arulazi D.: Quality of Service for Web services-Demystification, Limitations, and Best Practices. http://www.developer.com/java/web/article.php/2248251 (2002)
18. Stal, M.: Web Services: Beyond Component-Based Computing. Communications of the ACM. **45** (2002)71-76
19. Van Den Heuvel, W. and MaamarMoving, Z.: Toward a Framework to Compose Intelligent Web Services. Communications of the ACM. **46** (2003)103-109
20. Van Gurp, J., Bosch, J. and Svahnberg, M.: On the Notion of Variability in Software Product lines. Working IEEE/IFIP Conference on Software Architecture (WICSA'01)Amsterdam, The Netherlands. (2001)45-54
21. Weiss, D. and Lai, R.: Software Product Line Engineering.Addison-Wesley. (1999)
22. Zeng L.Z., Benatallah B. and Dumas M.: Quality Driven Web Services Composition. WWW2003, Budapest, Hungary. (2003)411-421
23. Zeng LZ., Benatallah B., Lei H., Ngu AHH., Flaxer D. and Chang H.: Flexible Composition of Enterprise Web Services. International Journal of Electronic Commerce and Business Media. **13** (2003)
24. Zhang W.J.: Research on Software Component Model and Corresponding Techniques to Support Variability(in Chinese). Ph.D. Thesis, Peking University. (2002)

A Formal Framework for Ontology Integration Based on a Default Extension to DDL

Yinglong Ma, Jun Wei, Beihong Jin, and Shaohua Liu

Technology Center of Software Engineering, Institute of Software,
Chinese Academy of Science, P.O.Box.8718, Beijing 100080, P.R. China
{m_y_long, wj, jbh, ham_liu}@otcaix.iscas.ac.cn

Abstract. The information society demands complete information from multiple sources, where available information is often heterogeneous and distributed. Because of semantic heterogeneities among ontologies of different information sources, it is rather difficult to integrate these local ontologies and get completely available information. In this paper, we propose a formal framework for integration of multiple ontologies from distributed information sources. To achieve this goal, implicit default information is extensively considered. We make a default extension to distributed description logics (DDL) for ontology integration and complete information query. A complete information query based on the integrated ontologies can boil down to checking default satisfiability of complex concept in accord with the query. Default satisfiability can be detected through an adapted tableau algorithm. Based on the proposed formal framework, a prototype system is developed, which can integrate strict as well as default information from multiple distributed information sources and global semantic information query can be performed.

Keywords: Ontology integration, distributed description logics, default extension, Tableau algorithm, semantic query.

1 Introduction

The use of ontology for the explication of implicit and hidden knowledge is one of approaches to overcome the semantics heterogeneity of multiple information sources [1, 2]. More importantly, in the last few years, there has been a lot of effort put in the development of techniques that aim at the Semantic Web [3], which will enable computers to partly "understand" the information on the Internet. A lot of those newly developed techniques require and enable the specification of ontologies on the Web [4]. With the increased availability of large and specialized online ontologies, the questions about the integration of independently deployed ontologies have become even more important.

Description Logic (DL) [5] is formalism for knowledge representation and reasoning. It's very useful for defining, integrating, and maintaining ontology, which provide the Semantic Web with a common understanding of the basic semantic concepts used to annotate Web pages. It's also ideal candidates for ontology languages [6]. RDF [14], RDFS [15], DAML+OIL [7] are clear examples of Description Logics.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 154–169, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Distinguished from with DL, Distributed Description Logics (DDL) [8] can better present heterogeneous distributed systems by modeling relations between objects and relations between concepts contained in different heterogeneous information sources. A DDL are composed of a collection of "distributed" DLs, each of which represents a subsystem of the whole system. All of DLs in DDL are not completely independent from one another as the same piece of knowledge might be presented from different points of view in different DLs. Each DL autonomously represents and reasons about a certain subset of the whole knowledge. A DDL might be regarded as a global integrated ontology, which is connected with a global DL, which encodes the information available in local DLs.

For integration of heterogeneous information sources, there are some problems that require to be solved. In some situations, only incomplete information can be got. These happen sometime as unavailability of pieces of information, sometime as semantic heterogeneities of information sources. Another problem is that there always exist some exceptional facts, which conflict with commonsense information. For example, commonly bird can fly, penguin belongs to bird, but penguin couldn't fly. In these situations, information reasoning should be based on default rules. This form of reasoning is called default reasoning, which is non-monotonic. Little attention, however, has been paid to the problem of endowing these logics above with default reasoning capabilities. It is a solution to model distributed information systems using DDL and further make a default extension to DDL for default reasoning. Then, a complete information query based on the integrated ontologies with default information can boil down to checking default satisfiability of the complex concept in accord with the query.

For these reasons above, we propose a formal framework for distributed ontologies integration based on a default extension of DDL. The framework provides a mathematic basis for querying complete information from integrated ontologies. To get complete information from multiple sources, we add default information into a distributed knowledge base derived from integrated ontologies.

In section 2 we introduce Description Logics and Distributed Description Logics briefly. The approach for default extension to DDL is presented in section 3. We make default extension to DDL using default rules, and introduce the definition of extended default distributed knowledge base (*EDDK*) by adding default rules into original distributed knowledge base. In Section 4, to check *default satisfiability* of a concept and perform default reasoning based on a distributed knowledge base with default rules, we adapt classical Tableau algorithm. Some examples are provided to explain our formal definition and default distributed reasoning. In Section 5, based on the formal framework, we develop a prototype system called OISDI for integrating strict ontology information as well as default information from distributed information sources. The system architecture and its main components are depicted, and the semantic query performed by the system is introduced. At the end, we discuss the related work in Section 6 and make a conclusion in Section 7.

2 Formalism Related to DL

Formalisms related to Description Logics have been used in a wide range of applications, which are usually given a declarative semantics. Unlike other

formalisms, one of the characteristics of formalisms related to Description Logics is that they are equipped with a formal, logic-based semantics. Another distinguished feature is the emphasis on reasoning as a central service.

2.1 Description Logics

Description Logics view the world as being populated by individuals. The basic notations in a DL are the notation of concept embracing some individuals on a domain of individuals, and roles representing binary relations on the domain of individuals. A specific DL provides a specific set of "constructors" for building more complex concept and role. The syntax and semantics of ALC DL are listed as Figure 1. Then one can make several kinds of assertions using these descriptions. There exist two kinds of assertions: subsumption assertions and assertions about individuals. The collection of subsumption assertions is called Tbox, which specifies the terminology used to describe some application domain. The collection of assertions about individuals forms Abox, which describes some states of world. A DL knowledge base $K=(T, A)$, where T and A are Tbox and Abox in DL respectively.

Constructor	Syntax	Semantic
primitive concepts	C	$A^I \subseteq \triangle^I$
primitive roles	R	$R \subseteq \triangle^I \times \triangle^I$
top	$\top$	$\triangle^I$
bottom	$\bot$	$\varnothing$
conjunction	$C \sqcap D$	$C^I \cap D^I$
disjunction	$C \sqcup D$	$C^I \cup D^I$
negation	$\neg C$	$\triangle^I \backslash C^I$
existential restriction	$\exists R.C$	$\{x \mid \exists y.\ (x, y) \in R^I \wedge y \in C^I\}$
value restriction	$\forall R.C$	$\{x \mid \forall y.\ (x, y) \in R^I \rightarrow y \in C^I\}$

Fig. 1. Syntax and semantics of the ALC DL

An interpretation for DL $I=<\triangle^I, \bullet^I>$, where $\triangle^I$ is a domain of objects and $\bullet^I$ the interpretation function. The interpretation function maps roles into subsets of $\triangle^I \times \triangle^I$, concepts into subsets of $\triangle^I$ and individuals into elements of $\triangle^I$.

Satisfaction and entailment in DL Tbox will be described using following notations:

1. $I \models C \sqsubseteq D$ iff $C^I \subseteq D^I$
2. $I \models T$, iff for all $C \sqsubseteq D$ in T, $I \models C \sqsubseteq D$
3. $C \sqsubseteq D$, iff for all possible interpretations I, $I \models C \sqsubseteq D$
4. $T \models C \sqsubseteq D$, iff for all interpretations I, $I \models C \sqsubseteq D$, such that $I \models T$
5. $K \models C \sqsubseteq D$, iff for all interpretations I, $I \models C \sqsubseteq D$ such that $I \models K$

These definitions are extended to Aboxes according to the following rules:

1. $I \models C(a)$, iff $a^I \in C^I$
2. $I \models p(a, b)$, iff $(a^I, b^I) \in p^I$

3. $I \models A$, iff for every assertion in form of $\alpha = C(a)$ or $\alpha = p(a, b)$ in A, $I \models \alpha$.
4. $I \models K$ iff $I \models T$ and $I \models A$
5. $K \models C(a)$, iff for all interpretations I, $I \models C(a)$ such that $I \models K$

   $K \models p(a, b)$, iff for all interpretations I, $I \models p(a, b)$ such that $I \models K$

2.2 Distributed Description Logics

A DDL consists of a collection of DLs, which is written $\{DL_i\}_{i \in I}$, every local DL in DDL is distinguished by different subscripts. The constraint relations between different DLs are described by using so-called "bridge rules" in an implicit manner, while the constraints between the corresponding domains of different DLs are described by introducing the so-called "semantics binary relations". In order to support directionality, the bridge rules from DL_i to DL_j will be viewed as describing "flow of information" from DL_i to DL_j from the point of view of DL_j. In DDL, i:C denotes the concept C in DL_i, i:C⊑D denotes subsumption assertion C⊑D in DL_i, and i:x denotes that x is individual in DL_i, j:{y,y_1,y_2,..} denotes that y, y_1, y_2, …are individuals in DL_j.

A bridge rule from i to j is described according to following two forms: $i:C \xrightarrow{\subseteq} j:D$ and $i:C \xrightarrow{\supseteq} j:D$. The former is called into-bridge rule, and the latter called onto-bridge rule. An individual correspondence from i to j is expressed as following two forms: $i:x \rightarrow j:y$ and $i:x \xrightarrow{=} j:\{y_1,y_2,...\}$. The first is called a partial individual correspondence, which shows the binary semantics relation r_{ij} (x, y) holds. The second is called a complete individual correspondence, which shows that for every element z in j:{y_1,y_2,..}, the relation r_{ij} (x, z) holds.

Similar to DL, DDL also embraces a set of subsumption assertions and a set of assertions about individuals, which are called DTB and DAB, respectively. A distributed Tbox (DTB) is defined based on Tboxes in all of local DLs and bridge rules between these Tboxes. A DTB $DT=<\{T_i\}_{i \in I}, B>$, where T_i is Tbox in DL_i, and for every $i \neq j \in I$, $B=\{B_{ij}\}$, where B_{ij} is a set of bridge rules from DL_i to DL_j. The definition of a distributed Abox (DAB) is based on Aboxes in all of local DLs and their partial and complete correspondence between these Aboxes. A DAB $DA=<\{A_i\}_{i \in I}, C>$, where A_i is Abox in DL_i, and for every $i \neq j \in I$, $C=\{C_{ij}\}$, where C_{ij} is a set of partial and complete individual correspondences from DL_i to DL_j.

The semantics for distributed description logics are provided by using local interpretation for individual DL and connecting their domains using semantics binary relations r_{ij} .A distributed interpretation $\Im=<\{I_i\}_{i \in I}, \boldsymbol{r}>$ of DT consists of interpretations I_i for DL_i over domain Δ^{I_i}, and a function $\boldsymbol{r}$ associating to each i, j∈I a binary relation $r_{ij} \subseteq \Delta^{I_i} \times \Delta^{I_j}$. $r_{ij}(d)=\{d' \in \Delta^{I_j} \mid <d, d'> \in r_{ij}\}$, and for any $D \subseteq \Delta^{I_i}$, $r_{ij}(D)=\bigcup_{d \in D} r_{ij}(d)$.

A distributed interpretation $\Im$ d-satisfies (written $\Im \models_d$) the elements of DTB $DT=<\{T_i\}_{i \in I}, B>$ according to following clauses: For every i, j∈I

1. $\Im \models_d i:C \xrightarrow{\subseteq} j:D$, if $r_{ij}(C^{I_i}) \subseteq D^{I_j}$
2. $\Im \models_d i:C \xrightarrow{\supseteq} j:D$, if $r_{ij}(C^{I_i}) \supseteq D^{I_j}$
3. $\Im \models_d i: C \sqsubseteq D$, if $I_i \models C \sqsubseteq D$
4. $\Im \models_d T_i$, if for all $C \sqsubseteq D$ in T_i, such that $I_i \models C \sqsubseteq D$
5. $\Im \models_d DT$, if for every i, $j \in I$, $\Im \models_d T_i$, and $\Im \models_d b$, for every $b \in \cup B_{ij}$
6. $DT \models_d i:C \sqsubseteq D$, if for every distributed interpretation $\Im$, $\Im \models_d DT \Rightarrow \Im \models_d i:C \sqsubseteq D$

A distributed interpretation $\Im$d-satisfies the elements of DAB $DA=<\{A_i\}_{i \in I}, C>$ according to following clauses: For every i, $j \in I$

1. $\Im \models_d i: x \rightarrow j: y$, if $y^{I_j} \in r_{ij}(x^{I_i})$
2. $\Im \models_d i: x \xrightarrow{=} j: \{y_1, y_2, ...\}$, if $r_{ij}(x^{I_i}) = \{y_1^{I_j}, y_2^{I_j}, ...\}$
3. $\Im \models_d i:C(a)$, if $I_i \models C(a)$

   $\Im \models_d i:p(a, b)$, if $I_i \models p(a, b)$
4. $\Im \models_d A_i$, if for all $\alpha = C(a)$ or $\alpha = p(a, b)$ in A_i, $\Im \models_d \alpha$
5. $\Im \models_d DA$, if for every $i \in I$, $\Im \models_d A_i$, and $\Im \models_d c$, for every $c \in \cup C_{ij}$
6. $DA \models_d i:C(a)$, if for every distributed interpretation $\Im$, $\Im \models_d DA \Rightarrow \Im \models_d i:C(a)$

   $DA \models_d i:p(a, b)$, if for every distributed interpretation $\Im$, $\Im \models_d DA \Rightarrow \Im \models_d i:p(a, b)$

A distributed knowledge base for distributed description logics $DK=(DT, DA)$, where DT is a DTB, DA a DAB.

3 Default Extension to DDL

Our default extension approach is operated on a distributed knowledge base. A distributed knowledge base originally embraces only some strict information. So there also exists the satisfiability problem of elements in distributed knowledge base. Based on semantic interpretation of DDL, we define the satisfiability of elements in a distributed knowledge base.

Definition 1. A distributed interpretation $\Im$ d-satisfies (written $\models_d$) the elements of $DK=(DT, DA)$ according to following clauses: For every i, $j \in I$

1. $\Im \models_d DK$, if $\Im \models_d DT$ and $\Im \models_d DA$
2. $DK \models_d i:C \sqsubseteq D$, if for every distributed interpretation $\Im$, $\Im \models_d DK \Rightarrow \Im \models_d i:C \sqsubseteq D$
3. $DK \models_d i:C(a)$, if for every distributed interpretation $\Im$, $\Im \models_d DK \Rightarrow \Im \models_d i:C(a)$

   $DK \models_d i:p(a, b)$, if for every distributed interpretation $\Im$, $\Im \models_d DK \Rightarrow \Im \models_d i:p(a, b)$
4. $DK \models_d DT$, if for all distributed interpretation $\Im$, $\Im \models_d DK \Rightarrow \Im \models_d DT$

   $DK \models_d DA$, if for all distributed interpretation $\Im$, $\Im \models_d DK \Rightarrow \Im \models_d DA$

Default information is useful for getting complete information from multiple distributed information sources. To be able to include default information in a distributed knowledge base, we firstly introduce the notation description of a default rule.

Definition 2. A default rule is an expression of the form $P(x):J_1(x),J_2(x),\ldots,J_n(x)/C(x)$, where P, C and J_i are concept names ($1\leq i\leq n$), and x is a variable. P(x) is called the prerequisite of the default, all of $J_i(x)$ are called the justifications of the default, and C(x) is called the consequent of the default. The meaning of default rule $P(x):J_1(x),J_2(x),\ldots,J_n(x)/C(x)$ can be expressed as follows: If there exists an interpretation *I* such that *I* satisfies P(x) and doesn't satisfy every $J_i(x)$ ($1\leq i\leq n$), then *I* satisfies C(x). Otherwise, if *I* satisfies every $J_i(x)$ ($1\leq i\leq n$), then *I* satisfies $\neg C(x)$.

For example, to state that a person can speak except if s/he is a dummy, we can use the default rule Person(x):Dummy(x)/CanSpeak(x). If there is a individual named John in a domain of individuals, then the closed default rule is Person(John):Dummy(John)/CanSpeak(John).

Then, to deal with strict taxonomies information as well as default information in distributed knowledge base, the definition of distributed knowledge base should be extended for including a set of default rules.

Definition 3. A default distributed knowledge base *DDK*=(*DT*, *DA*, **D**), where *DT* and *DA* are DTB and DAB respectively, **D** is a set of default rules.

An example of a *DDK* is shown in figure 2. The *DDK* is based on two local DLs, named DL1 and DL2 respectively. The DTB, DAB and **D** of the *DDK* are shown in Figure 2(a). Figure 2(b) provides a distributed interpretation of the *DDK*.

DT={{T_1={PARROT⊑BIRD, SPARROW⊑BIRD}, T_2={PARROT⊑FLYING_ANIMAL, GOAT⊑¬SPEAKING_ANIMAL}}, B={1:PARROT⊑2:PARROT}}
DA={{A_1={PARROT(parrot1), PARROT(parrot2)}, A_2= {PARROT(parrot)}}, C=∅}
D={BIRD(x):PARROT(x)/¬SPEAKING_ANIMAL(x)}

(a) DTB, DTA and **D** of the *DDK*

Δ^{I_1} ={parrot1, parrot2, sparrow, swan} $PARROT^{I_1}$ ={parrot1, parrot2} $BIRD^{I_1}$ ={parrot1, parrot2, sparrow, swan} $SPARROW^{I_1}$ ={sparrow}	Δ^{I_2} ={parrot, goat, butterfly} $PARROT^{I_2}$ ={parrot} $GOAT^{I_2}$ ={goat} $FLYING_ANIMAL^{I_2}$ ={parrot, butterfly} $\neg SPEAKING_ANIMAL^{I_2}$ ={goat}
r_{12}={(parrot1, parrot), (parrot2, parrot)}	

(b) Distributed interpretation of the *DDK*

Fig. 2. A *DDK* and its distributed interpretation

Similar to *DK*, There exists satisfaction (written $\models_{dd}$) problem of elements in *DDK*. So we call satisfiability of elements in *DDK default satifiability*. *Default satisfiability* serves as a complement of satisfiability definition in a distributed knowledge base with default rules.

Definition 4. A distributed interpretation $\Im$ dd-satisfies (written $\models_{dd}$) the elements of *DDK*=(*DT*, *DA*, **D**), according to following clauses: For every default rule δ in **D**, δ=P(x):$J_1(x)$, $J_2(x)$, …, $J_n(x)/C(x)$, and every i, j∈I

1. $\Im\models_{dd} DDK$, if $\Im\models_{d} DK$ and $\Im\models_{d} \delta$
2. $\Im\models_{dd} DT$, if $\Im\models_{d} DT$ and $\Im\models_{d} \delta$
3. $\Im\models_{dd} DA$, if $\Im\models_{d} Da$ and $\Im\models_{d} \delta$

4. $\mathfrak{I} \models_d \delta$, if $\mathfrak{I} \models_d P \sqsubseteq C$ =>there doesn't exist $\mathfrak{I}$, such that $\mathfrak{I} \models J_k \sqsubseteq \neg C$, $(1 \leq k \leq n)$
5. $\mathfrak{I} \models_d P \sqsubseteq C$, if $i \neq j$, such that $\mathfrak{I} \models_d i: P \sqsubseteq C$ or $\mathfrak{I} \models_d i: C \sqsupseteq P$ or $\mathfrak{I} \models_d i: P \xrightarrow{\sqsubseteq} j: C$ or $\mathfrak{I} \models_d i: C \xrightarrow{\sqsupseteq} j: P$
6. $DDK \models_{dd} DT$, if for all distributed interpretation $\mathfrak{I}$, $\mathfrak{I} \models_{dd} DDK \Rightarrow \mathfrak{I} \models_{dd} DT$
   $DDK \models_{dd} DA$, if for all distributed interpretation $\mathfrak{I}$, $\mathfrak{I} \models_{dd} DDK \Rightarrow \mathfrak{I} \models_{dd} DA$

In a distributed knowledge base, default information may have been used during reasoning, but a *DDK* is difficult to operate and not really helpful for reasoning with default information in a distributed knowledge base. Some additional information with respect to default rules should be included explicitly into *DT* and *DA* respectively. A closed default rule $P(x):J_1(x),J_2(x), \ldots, J_n(x)/C(x)$ can be divided into two parts: $P(x) \rightarrow C(x)$ and $J_i(x) \rightarrow \neg C(x)$, $(1 \leq i \leq n)$. We call the first part fulfilled rule, and the second exceptional rules. A rule of the form $A(x) \rightarrow B(x)$ means for every (distributed) interpretation I, $x \in A^I$, then $x \in B^I$, i.e. $A \sqsubseteq B$, where A and B are concept names, and x denotes an individual.

Definition 5. An extended distributed knowledge base *EDDK* is constructed based on a $DDK=(DT, DA, \mathbf{D})$, according to the following clauses: For every default rule δ in $\mathbf{D}$, $\delta = P(x): J_1(x), J_2(x), \ldots, J_n(x)/C(x)$,

1) Dividing δ into two parts which embrace fulfilled rule and exceptional rules, respectively. The fulfilled rule denotes that it holds in most cases until the exception facts appear, while the exceptional rules denote some exceptional facts.
2) Adding $P \sqsubseteq C$ and $J_i \sqsubseteq \neg C$ into *DT* $(1 \leq i \leq n)$, which are the assertions corresponding to fulfilled rule and exceptional rules, respectively.
3) Setting the priorities of different rules for selecting appropriate rules during reasoning. The assertions corresponding to exceptional rules have the highest priority, while original strict information has normal priority. The assertions corresponding to fulfilled rules are given the lowest priority.

In the course of constructing an *EDDK*, default information has been added into distributed knowledge base for default reasoning, because these default information may have been used during reasoning. Exceptional information has been assigned the highest priority to avoid conflicting with some strict information, while fulfilled rules would be used only in the situation that no other strict information can be used, its priority is least. A simplified view of the *EDDK* based on the *DDK* (shown in Figure 2) can be found in Figure 3.

In Figure 3, The default rule BIRD(x):PARROT(x)/¬SPEAKING_ANIMAL(x) is divided into one fulfilled rule and one exceptional rule. The fulfilled rule BIRD⊑¬SPEAKING_ANIMAL and the exceptional rule PARROT⊑SPEAKING_ANIMAL has been added into *EDDK*. In fact, an *EDDK* can be recognized as a collection of integrated ontologies with explicitly expressed default information. Default reasoning can be performed based on an *EDDK*.

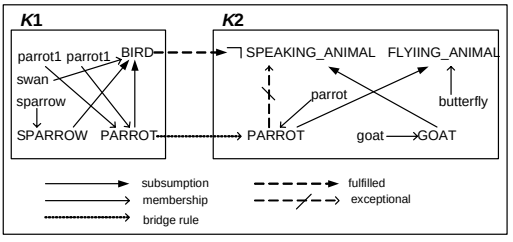


Fig. 3. An *EDDK*

4 Reasoning with Default Information

Reasoning with default information provides integrated ontologies with stronger query capability. A query based on integrated ontologies can boil down to checking default satisfiability of complex concept in accord with the query. Based on description logics, satisfiability of a complex concept is decided in polynomial time according to Tableau algorithm for ALC [5, 9]. To a certain extent, a DDL is connected with a global DL, which encodes the information available in local DLs. This would allow us to transfer theoretical results and reasoning techniques from the extensive current DL literatures. So in my opinion, detecting default satisfiability of a DDL is just detecting the default satisfiability of the global DL in accord with the DDL. A default extension to Tableau algorithm for ACL DL can be used for detecting *default satisfiability* of ACL concepts based on an *EDDK*.

Definition 6. A constraint set S consists of constraints of the form C(x), p(x, y), where C and p are primitive concept and primitive role, respectively. Both x and y are variables.

An I-assignment maps a variable x into a element of $\triangle^I$. If $x^I \in C^I$, the I-assignment satisfies C(x). If $(x^I, y^I) \in p^I$, the I-assignment satisfies p(x, y). If the I-assignment satisfies every element in constraint set S, it satisfies S. If there exist an interpretation I and an I-assignment such that the I-assignment satisfies the constraint set S, S is satisfiable. S is satisfiable iff all the constraints in S are satisfiable.

It will be convenient to assume that all concept descriptions in *EDDK* are in *negation normal form* (NNF). Using de-Morgan's rules and the usual rules for quantifiers, any ALC concept description can be transformed into an equivalent description in NNF. For example, the assertion description SPARROW$\sqsubseteq$BIRD can be transformed the form $\neg$SPARROW$\sqcup$BIRD.

To check satisfiability of concept C, our extended algorithm starts with constraint set S={$\neg$C(x)}, and applies transformation rules in an extended distributed knowledge base. The concept C is satisfiable iff the constraint set S is unsatisfiable. In applying transformation rules, if there exist all obvious conflicts (clashes) in S, S is unsatisfiable, which means the concept C is satisfiable. Otherwise, S is unsatisfiable. The transformation rules are derived from concepts and assertions in *EDDK*. If the constraint set S before the action is satisfiable, S after the action is also satisfiable. The adapted extension algorithm are shown as follows:

Begin
Step 1: Exceptional rules:
Condition: there exists a default rule of the form $P(x):J_1(x), J_2(x),\ldots,J_n(x)/C(x)$, and $\{J_i(x)\}\subseteq S$, $(1\le i\le n)$ in set of default rules.
Action: $S=S\{\neg C(x)\}$
Step 2: Strict rules
$\sqcap$ rule: Condition: $\{(C\sqcap D)(x)\}\subseteq S$, but S doesn't contain both C(x) and D(x).
Action: $S=S\cup\{C(x), D(x)\}$
$\sqcup$ rule: Condition: $\{(C\sqcup D)(x)\}\subseteq S$ and $\{C(x),D(x)\}\cap S=\varnothing$.
Action: $S=S\cup\{C(x)\}$ or $S=S\cup\{D(x)\}$
$\exists$ rule: Condition: $\{(\exists R.C)(x)\}\subseteq S$, and there doesn't exist y such that S contains C(y) and R(x, y).
Action: $S=S\cup\{C(y), R(x, y)\}$
$\forall$ rule: Condition: $\{(\forall R.C)(x), R(x, y)\}\subseteq S$, and S doesn't contain C(y).
Action: $S=S\cup\{C(y)\}$
Step 3: Fulfilled rule□
Condition: no other transformation rules can be applied, there exists a default rule of the form $P(x):J_1(x), J_2(x), \ldots, J_n(x)/C(x)$, and $\{P(x)\}\subseteq S$, S doesn't contain $\{J_i(x)\}$ $(1\le i\le n)$.
Action: $S=S\cup\{C(x)\}$
End

When the adapted algorithm is used for detecting default satisfiability of ALC concepts, every action must preserve satisfiability. Because if an action don't preserve satifiability, we cannot ensure the condition that if the constraint set before the action is satisfiable then the set after the action is satisfiabile. In the extension algorithm, we must prove the actions preserve satisfiability.

Theorem 1. Every action in the extension algorithm preserves satisfiability.

Proof. In the extension algorithm, every step probably embraces some actions, so we must prove that all of actions in these steps preserve satisfiability. Because the actions in the second step are originally derived from the classical Tableau algorithm, we have known they preserve satisfiability [5]. The remainder of the proof will only consider the actions in the first step and the third step.

1) In the first step, the action condition is that there exists a default rule of the form $P(x):J_1(x),J_2(x),\ldots,J_n(x)/C(x)$, and $\{J_i(x)\}\subseteq S$, $(1\le i\le n)$ in set of default rules. If there exists an interpretation *I* makes all $J_i(x)$ $(1\le i\le n)$ hold, then according to the Definition 2, we know *I* satisfies $\neg C(x)$. If the constraint set S before the action is satisfiable, then there exists an interpretation *I* such that *I* satisfies all of elements of S. Because $\{J_i(x)\}\subseteq S$, then *I* satisfies $J_i(x)$, $(1\le i\le n)$. Then because *I* satisfies both $\neg C(x)$ and S, we get *I* satisfies $S\cup\{\neg C(x)\}$ after the action.
2) In the third step, the action condition is that $\{P(x)\}\subseteq S$, S doesn't contain $\{J_i(x)\}$ $(1\le i\le n)$ and no other transformation rules can be applied. If there exists a

interpretation *I* makes all $J_i(x)$ $(1 \le i \le n)$ doesn't hold but P(x) hold, then according to the Definition 2, we know *I* satisfies C(x). If the constraint set S before the action is satisfiable, then there exists an interpretation *I* such that *I* satisfies all of elements of S. Because $\{P(x)\} \subseteq S$, then *I* satisfies P(x). Furthermore, we know *I* doesn't satisfy $J_i(x)$ $(1 \le i \le n)$, otherwise, there would exist some exceptional rules which can be applied.So from Definition 2, we get *I* satisfies C(x). Because *I* satisfies both S and C(x), we get *I* satisfies $S \cup \{C(x)\}$.

From above proofs, we can conclude that every action in the extension algorithm preserves satisfiability.

The default extension algorithm is divided into three steps. In the first step, we apply exceptional rules to constraint set because they have the highest priority. If exceptional rules can be used for the detected concept, strict rules will not be used. Otherwise, if no exceptional rules can be used, the strict rules can be applied to constraint set (step 2). The reason why we do like this is to avoid conflicting with some strict information. Another reason is to save reasoning time. In step three, only in the situation that no other strict information can be used, could fulfilled rules be used. The default extension algorithm either stops because all actions fail with obvious conflicts, or it stops without further used rules.

The algorithm is also applied to detect subsumption assertions. As usual, a subsumption assertion $A \sqsubseteq B$ is satisfiable iff the concept $A \sqcap \neg B$ is not satisfiable. The following example shown in Figure 4 demonstrates the algorithm with a tree-like diagram.

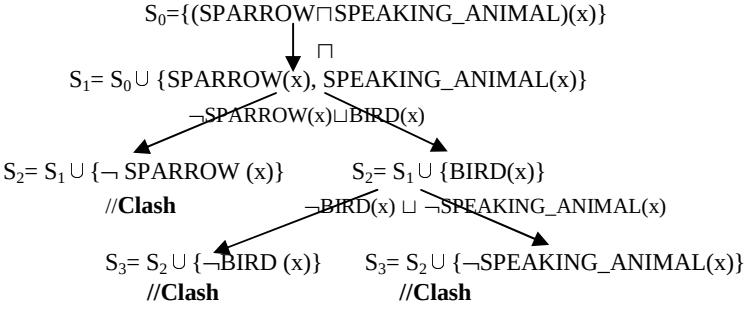


Fig. 4. Detecting default satisfiability of complex concept

In figure 4, we want to know whether the subsumption assertion $SPARROW \sqsubseteq \neg SPEAKING_ANIMAL$ is satisfiable in the *EDDK* shown in figure 3. That is to say, we should detect that the concept $SPARROW \sqcap SPEAKING_ANIMAL$ is unsatisfiable. The concept is firstly transformed into constrain set S_0. Considering the default rule BIRD(x):PARROT(x)/¬SPEAKING_ANIMAL(x) , we know that PARROT(x) isn't contained in S_0, Then, in the first step, the exceptional rule $\neg PARROT(x) \sqcup SPEAKING_ANIMAL(x)$ can not be applied to S_0. In the following steps, we apply strict rules, the reasoning continues until it stops with obvious

conflicts. Finally, the leaf node of every branch in this tree-like diagram is notated using “Clash” tag. So we know the constraint SPARROW⊓SPEAKING_ANIMAL is not satisfiable. That is to say, the subsumption assertion SPARROW⊑¬SPEAKING_ANIMAL is satisfiable.

Please note that the extension algorithm can tackle both general subsumption assertions and assertions about exceptional facts. In another example, We want to check whether the subsumption assertion PARROT(x)⊑SPEAKING_ANIMAL(x) is satisfiable, that is to say, we check the satisfiability of the concept PARROT(x)⊓¬SPEAKING_ANIMAL(x), which transformed into a constrain set. In the first step, when the exceptional rule ¬PARROT(x)⊔SPEAKING_ANIMAL(x) is applied to constraint set, the complete conflicts occur. So we know the concept PARROT(x)⊓¬SPEAKING_ANIMAL(x) is not satisfiable, which means that the subsumption assertion PARROT(x)⊑SPEAKING_ANIMAL(x) is satisfiable. Then reasoning process stops without applying other transformation rules. This can be served as an example of reasoning for an exceptional fact.

5 Prototype System for Semantic Information Integration

Based on the formal framework, we develop a prototype system using Hewlett-Packard Company’s Jena Semantic Web Toolkits [16] for integrating strict ontology information as well as default information from multiple distributed information sources. In the ontology integration system with default information (OISDI), We use specific ontology language such as RDF and RDFS for describing and organizing ontology information and knowledge. The component and logical architecture of the system OISDI and its semantic query are introduced in this section.

5.1 System Architecture

The architectures of the prototype system OISDI are described from two perspectives: component architecture and logical architecture. The component architecture of the case system is shown in figure 5. The whole distributed system is constructed by multiple local information sources, each of which describes and classifies local knowledge objects. In order to meet the need of ontology information sharing and integration, the ontology information and knowledge objects of each local information source are formally described and classified by RDF and RDFS ontology languages. As a result, it forms different local personalized ontologies and local knowledge bases. RDF is a semi-structured data model. Data is encoded using so called resource-property-value triples, which are also called statements. RDFS introduces classes and a subsumption hierarchy on classes for specifying metadata information such as classes, properties and hierarchy of knowledge objects. The metadata support provides the ability to describe, organize and associate knowledge objects, and promotes their interoperability.

The *EDDK* is implemented by the global ontology and the global knowledge base, which are created on a server module for describing and classifying the integrated

knowledge objects from different local information sources. The global ontology can integrate different terminology of ontologies through a common vocabulary, which provides global semantic query capability based on integrated ontologies.

We use RDF documents or RDFDB database [19] to store ontology data and metadata information. Using RDF query language, RDF files can be used for ontology information query. This way is simple and easily operated. But if the case system embraces a large amount of information data, it is quite difficult to use RDF files for storage of ontology information and semantic query, which causes very low query efficiency. RDFDB can be used to tackle this problem, and allows users to load RDF files from URIs into the database without any adaptation to them. RDFDB is a lightweight RDF database server, which supports RDF query language for performing semantic query.

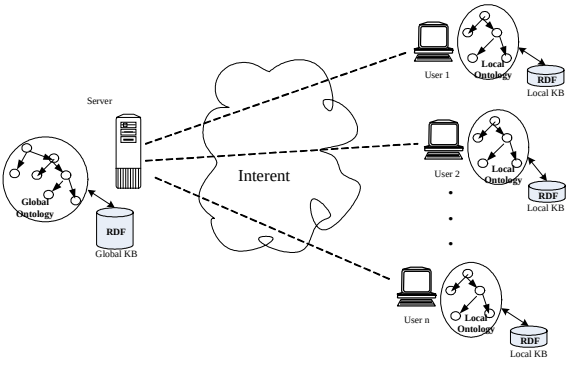


Fig. 5. Component architecture of the prototype system

The logical architecture of the prototype system OISDI is mainly considered for providing better capability of information interchange and interoperability. There are several kinds of information data such as individual knowledge objects, metadata and binary relations between different metadata, etc. The ontology representation combining RDF with RDFS can specify and organize these different kinds of information data. Based on these ontology representations, we can construct ontology inference model for semantic queries of application level. The logical architecture is considered into four levels: data description, metadata description, semantic description and application. Because a local knowledge base doesn't need to integrate local ontology information from other local knowledge bases, its logical architecture is simpler and easier than that of the global knowledge base on the server module. Here, we mainly pay attention to the logical architecture of the global ontology and the global knowledge base, which is described in Figure 6.

The global ontology and a global knowledge base are maintained on the server module. Each local user publishes their some personalized information that needs to share to the server module, and maps their personal metadata to certain categories of the global ontology. The information is described using RDF and RDFS languages. *Model Integrator* component is constructed for integrating local

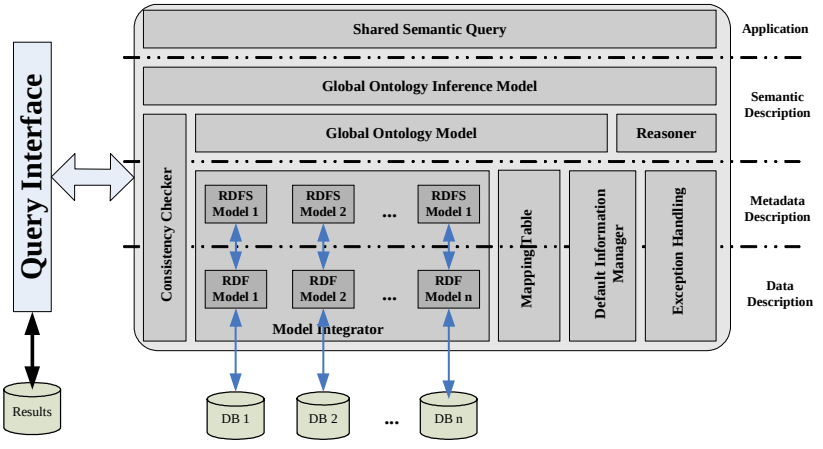


Fig. 6. Logical architecture of the server

ontologies information and forming the *Global Ontology Model*. *Mapping Table* component is constructed for terminology (e.g. concepts, properties) mapping between local sources and server module, where maintains a mapping table. The concepts of classes and properties from local information sources are either sub-concepts and sub-properties or equivalent concepts and properties of the global concepts and properties, respectively. *Consistency Checker* is used to detect some kinds of consistencies such as type, domains or range of properties. The default information of the whole system is organized and managed by *Default Information Manager* component and further is added into the global ontology model. *Exception Handling* component may tackle some exceptions such as terminology inconsistencies and query exceptions, etc. Using *Reasoner* component (Jena can provide different types of reasoners), the implicit information hidden in global ontology model can be explicitly expressed and added into *Global Ontology Inference Model*, which provides the capability for global semantic query of application level. Using RDF query language, user can perform global semantic query of application level based on *Shared Semantic Query* Component. The *Query Interface* component provides a semantic query interface for users.

5.2 Performing Semantic Query of Application Level

Semantic queries of application level are performed based on the global ontology inference. According to RDF semantic specification [18], the so-called semantics based on ontology involves primarily computation of transitive closure of the classes and property hierarchies, computation of all implicit members of classes and properties. The same is done for domains and range of properties. For example, an instance of subclass of a class (property) is still instance of the class (property), etc. The system allows local users to perform semantic query based on local information sources. If users want to get complete information from multiple information sources, they can execute global semantic query based on global ontology.

In order to execute semantic query of application level, we use RDQL query language [17] as query language for ontology information. RDQL is an SQL-like syntax, and regards RDF model as a set of triple data. RDQL is programmable for semantic query in Jena toolkit.

6 Related Work

In the description logics community, a number of approaches to extend description logics with default reasoning have been proposed. Baader and Hollunder [10] investigated the problems about open default in detail and defined a preference relation. The approach is not restricted to simple normal default. Two kinds of default rules were introduced by Straccia [11]. The first kind is similar to the fulfilled rules in our approach. The second kind of rules allows for expressing default information of fillers of roles. Lambrix [12] presented a default extension to description logics for use in an intelligent search engine, Dwebic. Besides the standard inferences, Lambrix added a new kind of inference to description logic framework to describe whether an individual belongs to a concept from a knowledge base. Calvanese [13] proposed a formal framework to specify the mapping between the global and the local ontologies. Maedche [21] also proposed a framework for managing and integrating multiple distributed ontologies. McGuinness [20] proposed a new merging and diagnostic ontology environment. Davies [22], Huynh [23] and Nejdl [24] develop some platforms built on metadata and/or ontologies for describing Web information and achieving information sharing and interoperability. However, default information was not considered in these different frameworks and systems.

An important feature of our formal framework distinguished from other work is that our default extension approach is based on DDL. To our best knowledge, little work has been done to pay attention to default extension to DDL.

7 Conclusion

In this paper, a formal framework is presented for integrating distributed ontologies with default information. The framework provides a mathematic foundation for querying complete information from integrated ontologies. To get complete information from multiple information sources, in which available information is often heterogeneous and distributed, we add default information into distributed knowledge base derived from integrated ontologies. The framework is based on default extension to DDL. The distributed knowledge base is originally used to present strict information. To perform default reasoning based on DDL, strict as well as default information is taken into account. Then, all of default information above is added into an *EDDK*, which is constructed from a distributed knowledge base with default rules. The default Tableau algorithm is used on *EDDK* where different rules have different priority: exceptional rules have the highest priority, and fulfilled rules the least. Based on the formal framework, we develop a prototype system called OISDI using Hewlett-Packard's Jena semantic web toolkits for integrating both strict ontology information and default information from multiple distributed information

sources. The component and logical architectures of the system and its semantic query are introduced. Based on the system, users can perform global shared semantic query for getting complete information from multiple information sources.

Acknowledgements

This research is partially supported by the National Grand Fundamental Research Program of China under Grant No.TG1999035805, 2002CB312005, the Chinese National "863" High-Tech Program under Grant No.2001AA113010.

References

1. Wache, H., et al.: Ontology-Based Integration of Information—A Survey of Existing Approaches. In proceedings of IJCAI'01 Workshop: Ontologies and Information Sharing, USA, (2001) 108-117
2. Visser, P. R. S., et al.: An Analysis of Ontology Mismatches; Heterogeneity Versus Interoperability. In AAAI 1997 Spring Symposium on Ontological Engineering, Stanford, USA, (1997) 164-172
3. Berners-Lee, T., et al.: The Semantic Web. Scientific American, Vol. 284, (2001) 34-43
4. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition, Vol.5, (1993) 199-220
5. Baader, F., et al.: The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, Cambridge, (2003)
6. Baader,F., Horrocks, I. and Sattler, U.: Description logics as ontology languages for the semantic web. Lecture Notes in Artificial Intelligence, Springer, 2003
7. Horrocks, I., et al.: Reviewing the design of DAML+OIL: language for the Semantic Web. In proceedings of AAAI'02, Cannada, (2002) 792-797
8. Borgida, A., et al.: Distributed Description Logics: Directed Domain Correspondences in Federated Information Sources. In Journal of Data Semantics, Vol.1, Springer Verlag, (2003) 153-184
9. Schimidt-schauβ, M., Smolka, G.: Attributive concept descriptions with complements. In Artificial Intelligence, Vol.48, (1991) 1-26
10. Baader, F., Hollunder, B.: Embedding Defaults into Terminological Representation Systems. In Journal of Automated Reasoning, Vol.14, (1995) 149-180
11. Straccia, U.: Default Inheritance Reasoning in Hybrid KL-ONE-style Logics. In proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93), (1993) 676-681
12. Lambrix, P., et al.: A Default Extension to Description Logics for use in an Intelligent Search Engine. In proceedings of 31st Annual Hawaii International Conference on System Sciences, Vol.5, USA, (1998) 28-35
13. Calvanese, D., et al.: A Framework for ontology Integration. In I.Cruz, S.Decker, J.Euzenat, and D.McGuinness, (eds.), The Emerging Semantic Web _ Selected Papers from the First Semantic Web Working Symposium, (2002) 201-214
14. W3C (World Wide Web Consortium).: Resource Description Framework (RDF) Model and Syntax Specification. (1999) http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
15. W3C (World Wide Web Consortium).: Resource Description Framework (RDF) Schema Specification 1.0. (2002) http://www.w3.org/TR/2000/CR-rdf-schema-20000327/
16. Hewlett-Packard Semantic Web Lab.: Jena Semantic Web Framework. (2004) http://www.hpl.hp.com/semweb/

17. HP Semantic Web Lab.: RDQL-RDF Data Query Language. (2004) http://www.hpl.hp.com/semweb/rdql.htm
18. Hayes, P., et al.: RDF Semantics. (2004) http://www.w3.org/TR/rdf-mt/
19. Guha, R.V.: RDFDB: An RDF Database. (2004) http://www.guha.com/rdfdb/
20. McGuinness, D.L., et al.: An environment for merging and testing large ontologies. In: Cohn, A.G., Giunchiglia, F., and Selman, B. (eds.): In KR2000: Principles of Knowledge Representation and Reasoning. San Francisco, (2000) 483–493
21. Maedche, A., et al.: Managing multiple and distributed ontologies on the Semantic Web. In The VLDB Journal-Digital Object Identifier (DOI). Vol.12. (2003) 286-302
22. Davies, J., Duke, A., Stonkus, A.: OntoShare: Using Ontologies for Knowledge Sharing. In Proceedings of WWW'02 International Workshop on the Semantic Web. Hawaii. (2002)
23. Huynh, D., Karger, D., Quan, D.: Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. In Proceedings of WWW'02 International Workshop on the Semantic Web, Hawaii, (2002)
24. Nejdl, W., et al.: EDUTELLA: Searching and Annotating Resources within an RDF-based P2P Network. In Proceedings of WWW'02 International Workshop on the Semantic Web, Hawaii, (2002)

A Predicative Semantic Model for Integrating UML Models

Jing Yang[1,2], Quan Long[1,3], Zhiming Liu[1,⋆], and Xiaoshan Li[4]

[1] International Institute for Software Technology, The United Nations University, Macau
{yj, longquan, lzm}@iist.unu.edu
[2] Department of Computer Science, Guizhou University, Guiyang, China 550025
[3] LMAM, Department of Informatics, School of Mathematical Sciences,
Peking University, Beijing, China 100871
[4] Faculty of Science and Technology, University of Macau, Macau, SAR, China
xsl@umac.mo

Abstract. This paper presents a predicative semantic model for integrating models from UML class diagrams and sequence diagrams. The integrated model is used for dealing with consistency problems of UML class diagrams and sequence diagrams. We also define the notion of consistent refinement of these integrated models.

Keywords: UML, Formal semantics, Refinement, Model integration.

1 Introduction

In a UML-based development process, such as the RUP [26, 14], several kinds of UML models are used to represent and analysis the artifacts created in a certain phase of the system development, which reflect the multiple views of UML:

- *Static view*: class diagram for static analysis.
- *Interactive view*: sequence diagrams and collaboration diagrams for interactions between objects.
- *Behavioral view*: state machines for dynamic behavioral specification and validation.
- *Functional view*: OCL [27] specifications for functionalities of objects.

Under the multiple views of UML, the developers can decompose a software design into smaller parts of manageable scales. However, several challenging issues inevitably arise from such a multi-view approach [24]:

- Consistency: the models of various views need to be syntactically and semantically compatible with each other (i.e. horizontal consistency) [15, 9, 1].
- Transformation and evolution: a model must be semantically consistent with its refinement (i.e. vertical consistency) [9, 1].
- Traceability: a change in the model of a particular view leads to corresponding consistent changes in the models of other views.
- Integration: models of different views need to be seamlessly integrated before software production.

⋆ On leave from the Department of Computer Science, the University of Leicester, UK.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 170–186, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Consistency checking and formal analysis of UML models have been widely studied in recent years [6, 4, 9, 25, 1]. A formal semantic model is needed for precise and intensive treatment of the problems. The informal semantics of UML is deliberately left flexible and extendable in order to allow UML to be used for different purposes, such as for requirement analysis, refinement of designs, and for code generation and testing.

The majority of the existing works on formal support to UML-based development, e.g. [7, 2, 6, 5, 10, 25], focus on the formalization of individual diagrams and only treat the consistency of the models of one or two views. Another phenomenon in research on formal use of UML is that different communities intend to emphasize different notations and use the full or even extended power of, say sequence diagrams or state machines. This would lose the advantages of the multiple-view modelling. It also leads to the increase in the complexity of a certain kind of models and the reduction of the role that the other kinds of UML models can play. To our knowledge, there is little work on consistent refinement of complete UML models of systems. A complete model of a system here means a family of models for the different views of the system.

This paper is towards the development of a semantic model of UML. The primary use of this model is for model integration, refinement and code generation [23]. The integration is based on the Relational Calculus of Object Systems (rCOS) defined in [12] that is designed for object-oriented system development in general.[1] The refinement calculus for rCOS [11, 12] will be used to define consistent refinement of UML models. The refinement process will preserve the consistency and the correctness of the system. The proposed techniques are also intended to support *model-driven development* [24] for executable UML models. As a starting work towards UML model integration and code generation, in this paper we only consider sequential software systems for which the UML class diagrams and sequence diagrams are powerful enough. The future work will extend this approach to deal with concurrent systems for which we need the other UML models, i.e. component diagrams, activity diagrams and statecharts.

The rest of the paper is organized as follows. Section 2 describes the theoretical basics of programming and presents an overview of rCOS . Section 2.2 provides formalization of UML diagrams and system models using rCOS semantics with the definition of model consistency. We show how a model is refined consistently in Section 4. Finally, concluding remarks are given in Section 5.

2 The Theoretical Basics

2.1 Unifying Theories of Programming

Our work is based on Hoare and He's *Unifying Theories of Programming* [13], in which a program or a program command is identified as a *design*, which is represented by a pair (α, P), where

[1] In early publications, such as [12], the calculus for object-oriented design was named as OOL. rCOS is produced by LaTex command {\large r}\textsc{COS}.

- α denotes the set of variables of the program, called the alphabet of the design.
- P, called the contract of the design, is a predicate of the form

$$p(x) \vdash R(x, x') \stackrel{def}{=} (ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$$

where

- x and x' stand for the initial and final values of program variables x in α,
- predicate p, called the *precondition* of the program, characterizes the initial states in which the activation of the program will lead its execution to termination,
- predicate R, called the *post-condition* of the program, relates the initial states of the program to its final states, and
- we describe the termination behavior of a program by the Boolean variables ok and ok', where the former is true if the program is properly activated and the latter becomes true if the execution of the program terminates successfully.

Please see the details in [13].

2.2 Our Relational Calculus of Object Systems

rCOS [12] is an object-oriented language with a rich variety of features including subtypes, reference types, visibility, inheritance, dynamic binding, polymorphism and local variable nested declarations. The language is designed for reasoning about object-oriented software at different levels of abstraction including specifications, designs and programs. The syntax of rCOS includes *object-oriented systems*, *class declarations*, *commands* and *expressions*. The main part of the syntax is very similar to Java. Furthermore rCOS is equipped with an observation-oriented semantics which is based on UTP [13]. Due to the page limit, we neglect the content in this paper. Please see the details in [12].

3 A Formal Syntax and Semantics of UML Models

In this section, firstly we will give the syntax of the class diagram and sequence diagrams. After that we will describe the requirement model and design model in our framework. Finally we will investigate the conditions on the consistent issues and present the rCOS semantics of a consistent model.

3.1 Syntax of Class Diagrams

A class diagram Γ (see Fig.1 and 5) identifies the following information.

1. The first part provides the static information on classes and their inheritance relationships:
   - **CN**: the finite set of classes has always been identified.
   - **super**: this is the direct generalization relation over the set **CN**.

2. The second part describes the structures of classes. For each $\mathbf{C} \in \mathbf{CN}$, we use $\mathbf{attr}(\mathbf{C})$ to denote the set

   $$\{< a_1 : \mathbf{T}_1 >, \cdots, < a_m : \mathbf{T}_m >\}$$

   of primitive attributes of $\mathbf{C}$, where $\mathbf{T}_i$ stands for the type of attribute a_i of class $\mathbf{C}$.
3. The third part provides the information about associations among classes: the finite set in which elements are of the forms:

   $$< \mathbf{C}_1,\ m_1,\ \mathit{Ass},\ m_2,\ \mathbf{C}_2 > \ \mid\ (\mathbf{C}_1,\ m_1,\ \mathit{Ass},\ m_2,\ \mathbf{C}_2)$$

   The first notation represents a direct association *Ass* from $\mathbf{C}_1$ to $\mathbf{C}_2$. The second notation indicates a undirect association *Ass* between classes $\mathbf{C}_1$ and $\mathbf{C}_2$. Here m_1 and m_2 are of the forms: $1 \mid 0..1 \mid\ * \mid 1..*$, indicating multiplicities respectively. We call undirect association conceptual association, and direct association design association.
4. For each $\mathbf{C} \in \mathbf{CN}$, *method*$(\mathbf{C})$ identifies the set of all methods of class $\mathbf{C}$.

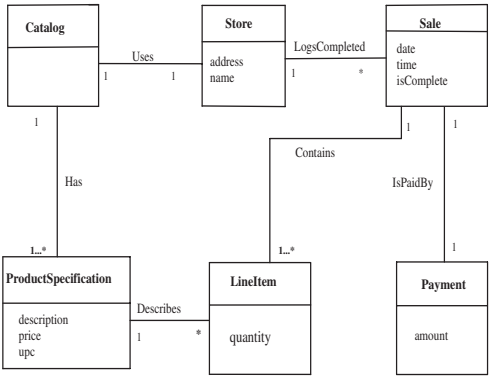


Fig. 1. A conceptual class diagram for an automated checking out system in a shop

The main condition of the well-formedness of a class diagram is that the inheritance relation does not introduce cycles between classes. Also we do not deal with multiple inheritance in a class diagram. The other issues are mainly naming problems.

In our framework, the class diagram defines the *system state space* allowed by the application, and each system state encompasses some objects and all links among these objects. Therefore, the class diagram plays the role as declarations of classes, types and variables in a program. A system state is a well-typed state of the variables. We can therefore easily specify (or translate) a class diagram Γ into a declaration section $cdecl_\Gamma$ in rCOS later.

3.2 Syntax of Sequence Diagrams

A sequence diagram consists of objects and ordered messages that describe how the objects communicate. An interaction occurs when one object invokes a method of another.

We now give the syntactic definition of sequence diagrams. We will allow *call back messages* in a sequence diagram (e.g. message "(5)" in Fig.2). The definition covers most features of UML 2.0 including combined fragments (except for the PAR one) [3], references to other sequence diagrams and nested sequence diagrams. The object "$: \underline{B}$" in Fig.2 represents a nested sequence diagram.

A sequence diagram *SD* consists of two parts:

1. A sequence of objects: $\langle obj_1, obj_2, \cdots, obj_n \rangle$. Each object obj_i has the following structure:
   - Each object *obj* is associated with a $\mathbf{type}$, denoted by $\mathbf{type}(obj)$, which is either a class name $\mathbf{C}$ in $\mathbf{CN}$ or a sequence diagram SD_1.
   - For each object *obj*, the property $\mathbf{multiob}(obj)$ equals *true* if *obj* is a multi-object (e.g. $: \underline{C}$ in Fig.2), otherwise $\mathbf{multiob}(obj)$ is *false*. Multi-objects represent one-to-many associations in a class diagram.
   - For each object *obj* there is a sequence of time-points $\langle p_1, p_2, \cdots, p_n \rangle$ which are totally ordered and represent the time points when an event occurs during the lifetime of the object. These points represent the ordering of messages sending and receiving, the combination fragments and the references to other sequence diagrams.

     We have a function *event* for each time-point p and $event(p)$ describes what happens at time-point p. For each time-point p, $event(p)$ can be one of the elements in the following set

     $$\{send,\ ack,\ receive,\ receiveack,\ option,\ loop,\ \ endfrag,\ ref,\ endref\}$$
2. A set *MSG* of messages: each message *msg* is one of the forms (src, m, tgt), (m, tgt) or (src, m) where
   - *src*, denoted by $source(msg)$, is a pair (obj, p) of an object and a time-point, and $source(msg) = (obj, p)$ means that object *obj* is the source of the message that occurs at time-point p, and we use $(obj, p).object$ to denote obj and $(obj, p).point$ to denote p.
   - *tgt* is a pair (obj, p) of an object and a time-point, represented by $target(msg)$ and
   - m, denoted by $method(msg)$, is any command in rCOS . Therefore, m can be a method call of the form $(ass, method())$ (sometimes it is simply written as $ass.m()$), which represents that *method*() of the target object is called by the source object via the association *ass*. Also, m can be a command, such as a design, an assignment, or any composite command of other kinds, but we require in this case the source object and the target object must be the same. This represents the execution of an internal action of the object. Finally, a message can be a return signal and in this case m is denoted as **return**.
   - A message (m, tgt) represents an incoming message to the sequence diagram and in this case m must be a method call. A message (src, m) shows an outgoing message from the sequence diagram and in this case m must be **return**.

   We do not show **return** in the diagram.

Fig.2 is an example of a sequence diagram in which

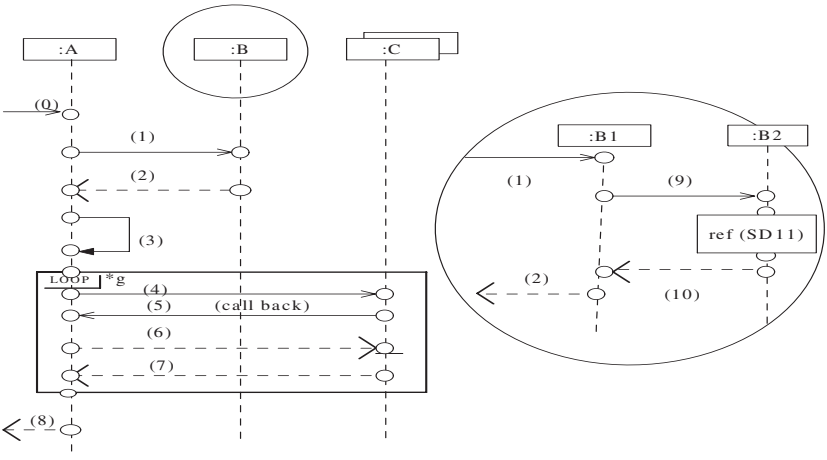


Fig. 2. An example sequence diagram

- *event*(p) = *send* shows a message is sent from this position of the current object. Similarly, *receive* means a message is reached at this position of the object. The message between a *send* point and a *receive* point will be drawn in a solid line with arrow to the *receive* point in the graph. For example, the message "(1)" and "(4)" in Fig.2.
- *ack* and *receiveack* points are used to denote a **return** message which are drawn in dotted line with open arrowhead back to the lifeline of the source object. For example, the message "(2)" and "(7)" in Fig.2.
- If an *event* of a point is *option* or *loop*, it will be equipped with another function, **guard**, which maps the point to its *guard* that is Boolean expression of the source object's attributes.
  The option combination fragment is used to represent a sequence that will be executed if the guard condition holds. An option combination fragment is used to model a "conditional choice" statement. The loop combination fragment is used to represent a repeated sequence. The body of the fragment will continue to be executed repeatedly until the guard condition becomes false. The *event*(p) = *endfrag* represents the end of a combination fragment.
- *event*(p) is *ref* means that from point p the current sequence diagram begins to call another sequence diagram and *endref* represents the end of the call. A *ref* point will be equipped with a **name** representing the sequence diagram it calls.

3.3 Well-Formedness of Sequence Diagrams

We need to ensure that a sequence diagram is well-formed. The well-formedness is concerned with the following conditions:

- For each message *msg* in the sequence diagram, the *event* of the source point of *msg* must be a *send* or *ack* and the *event* of target point of *msg* must be a *receive* or *receiveack*, respectively.

- If a point p_1 represents the beginning of combined fragments, i.e. *loop* or *option*, there must be one and exactly one corresponding *endfrag* point p_2 on the same object such that p_2 is later than p_1.
- For a point p_1 with $\mathit{event}(p_1) = \mathit{ref}$, there must be point p_2 on the same object such that $\mathit{event}(p_2) = \mathit{endref}$ and p_2 is later than p_1. The well-formedness of the referred sub-sequence diagram is checked recursively.
- If *obj* is a nested sequence diagram, then for every *matched* pair of sending and returning messages $(\mathit{src}, m, (\mathit{obj}, p_1))$ and $((\mathit{obj}, p_2), m', \mathit{tgt})$, there is a corresponding matched pair of messages (m, tgt_1) and (scr_1, m') of source-less (incoming) message and target-less (outgoing) message in the sub-sequence diagram **type**(obj). The order of these message is preserved in the sub-sequence diagram and the sub-sequence has to be well-formed.

The well-formedness of a sequence diagram has also to ensure the sequence diagram indeed represents a scenario of method calls. This means that 1). order of the message sending and receiving must be consistent, and for all messages from the same object, the earlier it is sent the earlier it is received by the target object; 2). if a message *msg* invokes message msg_1, then msg_1 must return before *msg* does.

For a sequence diagram, let *action* be the set of all its messages, combined fragments, and referred sub-sequence diagrams. For actions action_0 and action_1 from the same object, we use the notation

$$\mathit{returned}(\mathit{action}_0, \mathit{action}_1)$$

representing that the execution of action_0 is finished before the execution of action_1. In particular, if action_0 is a message with a method call meth_0, then its corresponding **return** message must be received by the object before the the execution of action_1.

Using the above notations, we give the following definitions for future use.

Definition 1 (Directly invoke). Let msg_0 be a message in a sequence diagram SD.

- **message.** Message msg_0 *directly invokes* msg_1, denoted by $\mathit{Invoke}(\mathit{msg}_0, \mathit{msg}_1)$ if $\mathit{target}(\mathit{msg}_0).\mathit{object} = \mathit{source}(\mathit{msg}_1).\mathit{object}$ and $\mathit{target}(\mathit{msg}_0).\mathit{point}$ is the latest point in the set

$$\{p | p \in \mathit{target}(\mathit{msg}_0).Points \wedge p < \mathit{source}(\mathit{msg}_1).\mathit{point} \wedge \neg \mathit{returned}(\mathit{msg}_0, \mathit{msg}_1)\}$$

  where $\mathit{target}(\mathit{msg}_0).Points$ is the time-points of the target object of msg_0 in SD.
- **fragment.** Let *freg* be a combined fragment and (obj, p) is the beginning point for *freg*. We say msg_0 *directly invokes freg*, denoted by $\mathit{Invoke}(\mathit{msg}_0, \mathit{freg})$, if $\mathit{target}(\mathit{msg}_0).\mathit{object} = \mathit{obj}$ and $\mathit{target}(\mathit{msg}_0).\mathit{point}$ is the latest point in the set

$$\{t | t \in \mathit{target}(\mathit{msg}_0).Points \wedge t < p \wedge \neg \mathit{returned}(\mathit{msg}_0, \mathit{freg})\}$$

- **ref.** Let SD_1 be another sequence diagram and $\mathit{rf} = (\mathit{obj}, p)$ is the *ref* point where SD_1 is called. We say msg_0 directly invokes *rf*, denoted by $\mathit{Invoke}(\mathit{msg}_0, \mathit{rf})$, if $\mathit{target}(\mathit{msg}_0).\mathit{object} = \mathit{rf}.\mathit{object}$ and $\mathit{target}(\mathit{msg}_0).\mathit{point}$ is the latest point in the set

$$\{p | p \in \mathit{target}(\mathit{msg}_0).Points \wedge p < \mathit{rf}.\mathit{point} \wedge \neg \mathit{returned}(\mathit{msg}_0, \mathit{rf})\}$$

Definition 2 (Directly follow). Let msg_0, msg_1 be two messages, $source(msg_0).object = source(msg_1).object$. We say msg_1 *directly follows* msg_0, denoted as $Follow(msg_0, msg_1)$, if $source(msg_1).point$ is the smallest element of the set

$$\{p | p \in source(msg_0).Points \wedge p > source(msg_0).point \wedge returned(msg_0, msg_1)\}$$

The directly follow relationship between other actions (fragments or the references to other sequence diagrams) is similar to the message case in **Definition 2**.

3.4 Requirement Models in UML

As in [16, 19], the development cycle of a software system starts with the construction of a *requirement model RM*. In UML, requirements are captured and described by a conceptual class diagram and some use cases. Conceptual class diagram is a class diagram in which every class has no method, and the associations are conceptual (i.e. undirect). Informally, such requirement model consists of a number of UML models, including a *conceptual class diagram* Γ_r, a *use-case diagram* U_r, a family Δ_r of *use-case* (or *system*) *sequence diagrams* (one for each use case), and some *activity diagrams* if concurrency is concerned. In this paper, we do not consider concurrency[2]. We can thus denote the requirement model by a triple $RM = \langle \Gamma_r, U_r, \Delta_r \rangle$. Each use case U is modelled as a use-case controller class U-*Controller* (see Fig. 3).

Class *U-Controller* {
private $\underline{T}\ \underline{x}$;
method $op_1(< T_{11}\ x_1 >, < T_{12}\ y_1 >, <>)\{c_1\}$;
$\cdots$;
$op_n(< T_{n1}\ x_n >, < T_{n2}\ y_n >, <>)\{c_n\}$
}

where the attributes $\underline{x}$ may include state control variables which are private to the controller class. For each method of the form $op_i(< T_{i1}\ x_i >, < T_i\ y_i >, <>)\{c_i\}$, x_i is a list of value parameters, y_i a list of result parameters, and c_i is a command.

This formalization implies that all attributes and associations in other classes are directly visible to all the use-case controller classes. The use-case diagram also provides the information about the design associations among the use-case controller classes. If use case U_1 includes use case U_2 in the use-case diagram, then there is an association from U_1-*Controller* to U_2-*Controller*. Use-case sequence diagrams Δ_r (see Fig.4), describe the the interaction between the actors and the system. Following the facade controller pattern in [16], for the operations in a use-case sequence diagram we declare the signatures in the corresponding use-case controller class. In a use-case sequence diagram, the receiver of each message is an object of the use-case controller class, and the sender is either an actor or an object of another use-case controller class.

Therefore, for each use case U, U-*Controller* declares the operations that appear in the use-case sequence diagram and the body of each method in U-*Controller* is defined from the system sequence diagram of the U-*Controller* class. Following the idea above,

[2] It is often not recommended to consider concurrency at this early stage in an iterative development process.

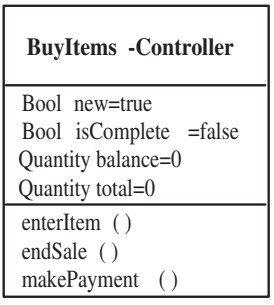


Fig. 3. The controller class for BuyItems

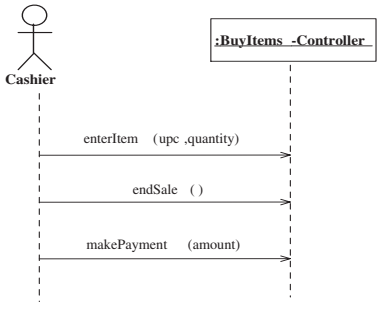


Fig. 4. The use-case sequence diagram for use case BuyItems

given a UML requirement model $RM = \langle \Gamma_r, U_r, \Delta_r \rangle$, the *normal form specification* in rCOS for *RM* is:

$$[\![RM]\!] \stackrel{def}{=} cdecls_{\Gamma_r}; U_1\text{-}Controller; \ldots; U_n\text{-}Controller$$

U_i, $i = 1, \ldots, n$, are the use cases of *RM*.

We will give the details of the meaning of $[\![\cdot]\!]$ later.

3.5 Design Models in UML

In UML, a design model DM should consist of a *design class diagram* Γ_d (see Fig.5), a family Δ_d of *object interaction* or *sequence diagrams*, at least one for each method in *U-Controller*. We can thus define a design model as an ordered couple $DM = \langle \Gamma_d, \Delta_d \rangle$.

Object sequence diagrams Δ_d operate on the design class diagram Γ_d. In general sequence diagrams do not contain many details for describing the functionality of the system. In an informal UML-based development, other means, such as textual description, is used to describe the functionality of the system. In our formal framework, we provide formal specification of the body of the methods that will ensure the behavior required by the sequence diagrams.

Therefore, a design model *DM* is also specified as the *normal form specification* in rCOS :

$$[\![DM]\!] \stackrel{def}{=} cdecl_1; \cdots; cdecl_n$$

In next subsection we will give the details of the definition of $[\![\cdot]\!]$.

3.6 rCOS Semantics for UML Models

In this subsection we will give the rCOS semantics for a UML model. A UML model is an ordered couple $\langle \Gamma, \Delta \rangle$. If it is a requirement model, then $\Gamma = \Gamma_r \cup U_r, \Delta = \Delta_r$; if it is a design model, then $\Gamma = \Gamma_d, \Delta = \Delta_d$.

The semantics of a UML model $\langle \Gamma, \Delta \rangle$, is a sequence of class declarations in rCOS .

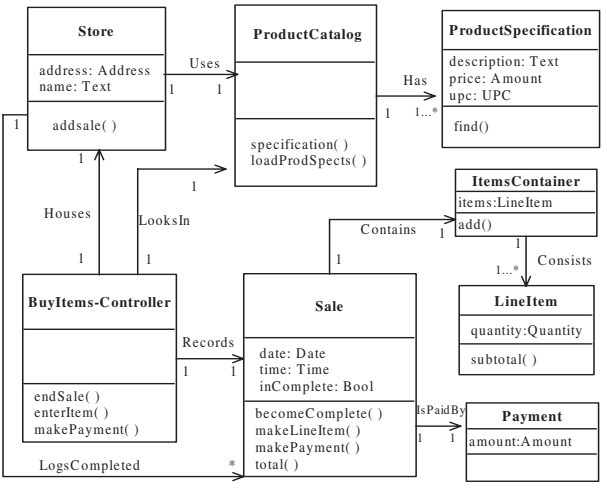


Fig. 5. A design class diagram for use case BuyItems

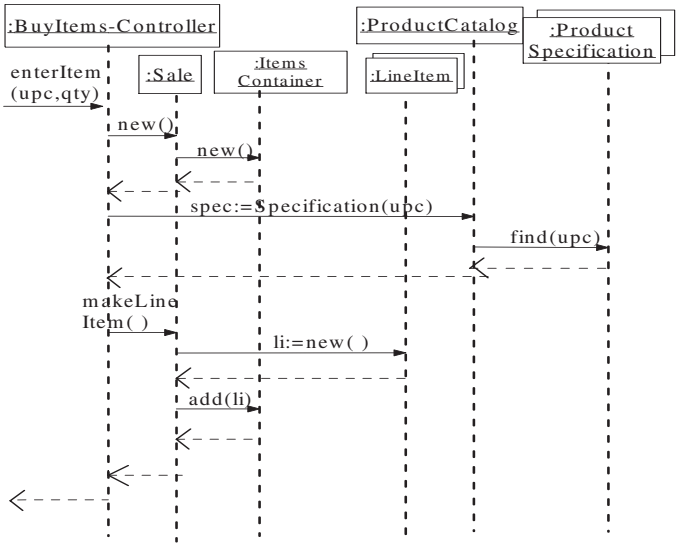


Fig. 6. Sequence diagram for *enterItem*()

Before giving the semantics from a class diagram and a set of sequence diagrams, we need to ensure that these diagrams are consistent.

For a well-formed class diagram and well-formed sequence diagrams, we give the following items as the definition of consistency. A violation of any of them will be considered as an inconsistency.

- **Association.** For each *msg* $\in$ *MSG*, there must be a corresponding association in the class diagram. Notice, this is static as it cannot ensure that the object which is sending the message in a particular state during the execution is currently associated with the target object of the message.
- **Class Name.** For the above-mentioned association, each of the two related classes in the class diagram must have the same name with the object related to *msg* in the sequence diagram.
- **Method.** Each method signature in the sequence diagram must be the same as the one in the class diagram. Furthermore, if $m()$ is the method of a message sent from $\underline{:C}$ to $\underline{:D}$ in the sequence diagram, then $m()$ must be a method of class D in the class diagram.
- **Attribute.** The variables used in the guard of a message should be directly accessible by the source object.
- **Multiplicity.** If an association in class diagram is one-to-many, the corresponding object in the sequence diagram must be a multi-object. Notice that multiplicity and other general class invariants should be ensured by the design of the sequence diagram, not by the consistency checking.

Now we give the semantics of consistent model $\langle \Gamma, \Delta \rangle$, denoted by $[\![\langle \Gamma, \Delta \rangle]\!]$, as follows:

- **Class.** A class $\mathbf{C}$ in Γ is declared as a *skeleton* of class declaration $cdecl$ in $[\![\langle \Gamma, \Delta \rangle]\!]$ with their attributes and method signatures. All attributes are declared to be public.
- **Association.** For association *role name* a from $\mathbf{C}_i$ to $\mathbf{C}_j$ in Γ, if the multiplicity of $\mathbf{C}_j$ is $0..1$ or 1, $cdecl_i$ for $\mathbf{C}_i$ has an attribute a with the type $\mathbf{C}_j$; and if the multiplicity of $\mathbf{C}_j$ is $1..*$ or $*$, $cdecl_i$ for $\mathbf{C}_i$ has an attribute a of type $\mathbf{C}_j$ and an attribute *a-set* of type of the powerset $\mathbb{P}\mathbf{C}_j$.
- **Constraints:** Constraints, such as invariants, multiplicity and aggregation, are specified in terms of pre-post conditions of methods in rCOS .
- **Reference attributes:** For a sequence diagram *SD* $\in \Delta$, a method of an object of type $\mathbf{C}_1$ is called by an object of type $\mathbf{C}_2$, there will be an attribute with the reference type of $\mathbf{C}_1$ in the class declaration for $\mathbf{C}_2$.
- **Method bodies:** The method bodies in each class will be determined by the scenarios of method call in sequence diagrams Δ. We will consider the following two items:
  - *Directly invoked actions:* For a message either *msg* $= (src, m, tgt) \in \Delta$ or *msg* $= (m, tgt) \in \Delta$, if m is a method signature, *SD* is a particular sequence diagram in which m is called, then the following sequence of actions, denoted by $body_{SD}$, is a path of the execution of an invocation of m;
    $action_1; action_2; \cdots action_n$
    where *Invoke*(*msg*,$action_1$) and *Follow*($action_i, action_{i+1}$) for each $i : 0 < i < n$, and no more actions directly follow $action_n$.
  - *Method appears in several sequence diagrams.* If a method m appears in several sequence diagrams, say, $SD_1, SD_2, \cdots, SD_n \in \Delta$ and
    $body_{SD_1}, body_{SD_2}, \cdots, body_{SD_n}$
    are the corresponding method bodies in these sequence diagrams. Then the body of m is the non-determined choice of them:
    $m()\{\sqcap_{i=1}^{n} body_{SD_i}\}$

The overall well-formedness of the class diagram and the sequence diagrams and their consistency are ensured by the *well-formedness* of the rCOS specification [12] obtained from the above definition. This translation can be automated and we have designed an algorithm in [23] for this purpose.

Example 1. The model composed of the class diagram in Fig.5 and the sequence diagram in Fig.6 has the semantics in rCOS as follows.

Class $Store$ {
  public $Address\ address, Text\ name, ProductCatalog\ prod,$
    $\mathbb{P}Sale\ sli, Sale\ s,$ ***BuyItems-Controller*** $buyctr;$
  method $addSale()$
}

Class $ProductCatalog$ {
  public $\mathbb{P}ProductSpecification\ proli, ProductSpecification\ pro;$
  method $specification(UPC\ upc, ProductSpecification\ spec)\{$
    $pro.find(upc)$
    $\}$
  $loadProdSpects()$
}

Class $ProductSpecification$ {
  public $Text\ description, Quantity\ price, UPC\ upc, \mathbb{P}LineItem\ li, Lineitem\ l;$
  method $find(UPC\ upc)$
}

Class $BuyItems - Controller$ {
  private $ProductCatalog\ p, Sale\ sale;$
  method $endsale();$
    $enterItem(UPC\ upc, Quantity\ qty)\{$
    $sale := Sale.new();$
    var $spec := p.specification(upc);$
    $sale.makeLineItem(spec, qty)$
    $\}$
    $makePayment(Amount\ amount, Amount\ balance)$
}

Class $Sale$ {
  public $Date\ date, Time\ time, Bool\ inComplete, ItemContainer\ icont,$
    $\mathbb{P}LineItem\ lset, LineItem\ lineItem, Payment\ p;$
  method $new()\{$
    $lineitem := LineItem.new()$
    $\}$
    $makeLineItem(ProductSpecification\ spec, Amount\ total)\{$
    var $li := lineItem.new();$
    $icont.add(li)$
    $\}$
    $makePayment(Amount\ amount, Amount\ balance);$
    $becomeComplete();$
    $total()$
}

Class $LineItem$ {
  public $Quantity\ quantity$;
  method $subtotal(Quantity\ quantity, Quantity\ price, Amount\ subtotal)$
}

Class $Payment$ {
  public $Amount\ amount$;
  method
}

Class $ItemContainer$ {
  public $\mathbb{P}Lineitem\ items, Lineitem\ item$;
  method $add(LineItem\ item, \mathbb{P}LineItme\ items)$
}

4 Model Refinement

Having given the definition of the semantics $[\![\cdot]\!]$ for UML models in the above section, now we can define the refinement and correct relationship between UML models.

Firstly, let us recall the following two refinement definitions in rCOS [12].

Definition 3 (Refinement between object systems). Let S_1 and S_2 be object programs which the same set of global variables **glb**. S_1 is a refinement of S_2, denoted by $S_1 \sqsupseteq_{sys} S_2$, if the behavior of S_1 is more predictable and controllable than that of S_2.

$S_1 \sqsupseteq_{sys} S_2 \stackrel{def}{=} \forall \underline{x}, \underline{x'}, ok, ok' \cdot (S_1 \Rightarrow S_2)$ *where* $\underline{x}$ *are the variables in* **glb**.

This means the external behavior of S_1, i.e.the pair of pre- and post-global states, is a subset of that of S_2.

Definition 4 (Refinement between declaration sections). Let $cdecls_1$ and $cdecls_2$ be two declaration sections. $cdecls_1$ is a refinement of $cdecls_2$, denoted by $cdecls_1 \sqsupseteq_{class} cdecls_2$, if the former can replace the latter in any object system:

$$cdecls_1 \sqsupseteq_{class} cdecls_2 \stackrel{def}{=} \forall P \cdot (cdecls_1 \bullet P \sqsupseteq_{sys} cdecls_2 \bullet P)$$

where P stands for a main method $(\mathbf{glb}, c)$.
Intuitively, it states that $cdecls_1$ supports at least the same set of service as $cdecls_2$.

Now we provide the definitions of model refinement and correctness.

Definition 5 (Model refinement). Let $\langle \Gamma_1, \Delta_1 \rangle$ and $\langle \Gamma_2, \Delta_2 \rangle$ be two UML models. $\langle \Gamma_2, \Delta_2 \rangle$ is refined by $\langle \Gamma_1, \Delta_1 \rangle$, denoted by $\langle \Gamma_1, \Delta_1 \rangle \sqsupseteq_{model} \langle \Gamma_2, \Delta_2 \rangle$if the former's semantics refines the latter's:

$$\langle \Gamma_1, \Delta_1 \rangle \sqsupseteq_{model} \langle \Gamma_2, \Delta_2 \rangle \stackrel{def}{=} [\![\langle \Gamma_1, \Delta_1 \rangle]\!] \sqsupseteq_{class} [\![\langle \Gamma_2, \Delta_2 \rangle]\!]$$

Definition 6 (Correct Design Model). A design model *DM* is correct with respect to the requirement model *RM*, denoted by $Correct(DM, RM)$ if it is a model refinement of *RM*:

$$Correct(DM, RM) \stackrel{def}{=} [\![DM]\!] \sqsupseteq_{model} [\![RM]\!]$$

As before-mentioned, the *normal form specifications* in rCOS for requirement model and design model only concern the class declaration in an object program and its main method corresponds to the application program using services which are provided by the methods of the classes in the design model. Thus, in this article, we are only interested in refinement relation between declaration sections. We allow the following refinement rules to a declaration section within the UML framework.

1. Adding a class declaration: this corresponds to adding a class into the class diagram, the methods of the new class into sequence diagrams.
2. Introducing a *fresh* attribute to a class: this corresponds to adding a fresh attribute of primitive type to the class or adding a design association from the class to another.
3. Introducing inheritance. If none of the attribute of class N is appeared in class M or any superclass of M, we can make M a direct superclass of N.
4. Moving some attributes from a class to its direct superclass: if all subclasses of class N have a common attribute, the common attribute can be moved to class N from all of its subclasses.
5. Data encapsulation: Suppose class M has a public attribute, and no method of other classes accesses this attribute except those of subclasses of M, we change the visibility of this attribute from public to protected. Suppose class M has a protected attribute, and no method of its subclasses accesses this attribute, we change the visibility of this attribute from protected to private.
6. Adding a *fresh* method into a class: This approach allows us to add a method signature into the class in the class diagram, and add a sequence diagram.
7. Refining the body command of a method $m()\{c\}$ in a declared class. This may lead to the replacement of the subsequence diagrams involving with $m()$.
8. Moving a method from a class to its direct superclass if the method body does not access any protected or private attribute of the class.
9. Copying a method from a class to its direct subclass.
10. Delegating some tasks of a class to its associated classes. If a method of a class contains a sub-command that can be realized by a method of another class, we can replace that sub-command with a method invocation to the latter class. Notice the sequence diagrams involved the method should be refined too.
11. Removing unused attributes: for a private attribute, it can be dropped if it does not appear in any method of the class; for a protected attribute, it can be dropped if it does not appear in any method of the class and its subclasses; for a public attribute, it can be dropped if it does not appear in any method of any class.
12. Removing unused methods: if a method is not called by other method or the main method in the object program, the method can be removed.

Example 2. The conceptual class diagram in Fig.1, the system sequence diagram in Fig.4 together with the use-case diagram *BuyItems-Controller* in Fig.3 form a requirement model. We can apply the refinement calculus to this model for the design of the method *enterItem*() (similarly to methods *endSale* and *makePayment*) according to the design class diagram in Fig.5, then achieve the object sequence diagram in Fig.6 step by step:

1. *BuyItems-Controller* delegates the responsibility of creating a new sale to *Sale*,
2. *BuyItems-Controller* delegates the job of finding the specification using upc matching to *Catalog*, which delegates the job further to the multi-object with type of *ProductSpecification*,
3. *BuyItems-Controller* delegates the task of making a line item to *Sale*.

5 Conclusions and Future Works

This work is towards a formal foundation for component and object systems development based on the formal object-oriented specification notation rCOS and its refinement calculus in [12, 11]. In rCOS , *normal form specification* of an object system is a sequence of class declarations and a main method. Each class declaration consists of some attributes, method signatures and method-body definitions, which has corresponding notations in UML. In this paper, we focused on the formalization of UML class diagrams and sequence diagrams in rCOS . With this formalization, we can integrate these two kinds of UML models and carry out consistent refinement of these diagrams. The consistency conditions between these two kinds UML models are treated as the well-formedness conditions of their corresponding integrated specification in rCOS .

Compared to most exitsing work, e.g. [6, 4, 9, 25, 1], our approach is also *transformational*. However, we also provide integration of UML models. Furthermore, because different sections in a rCOS specification clearly corresponding different UML diagrams, the formal specification of the integrated model can be transformed back to UML diagrams, i.e. the transformation is reversible. This is very important, as this allows us to obtain refined UML diagrams from a refined rCOS specification. Thus, this approach also supports *re-engineering*.

In our formalization, the horizontal consistency is mainly static: the syntactic consistency can be checked statically using an algorithm, while the invariants can be verified with model-checking tools. Vertical consistency (of refinement) is mainly semantic. This has fully justified our use of predicates in meeting the challenge of UML formalization [7]. Fundamental techniques of program and data refinement can be applied to UML transformation. This also supports UML-based model-driven development [8, 24].

Our future work will also include the integration of activity diagrams (for concurrency) in the framework and the extension of rCOS with the notation of components to support component diagrams in UML2.0. Tools have also been developed using this framework [17].

In our related works, general transition systems are introduced to provide an integrated model of conceptual class diagrams and use cases (without the treatment of sequence diagrams, state machines and use-case diagrams) [22]. A version of rCOS without reference types is presented in [11] and used in [21] for the specification of the integrated model of [22]. Article [18] uses rCOS for the specification of design class diagram and sequence diagrams, but without rules for model transformation. In [20], we show how rCOS is used a the foundation for a rigorous approach to object-oriented in general, and to UML-Base development in particular.

Acknowledgement. We would like show our appreciation to Yifeng Chen and Yuxin Cheng for their comments on earlier versions of the paper.

References

1. R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 2:18–40, 2000.
2. R.J.R. Back, L. Petre, and I.P. Paltor. Formalizing UML use cases in the refinement calculus. In *Proc. UML'99*. Springer-Verlag, 1999.
3. Donald Bell. UML's sequence diagram. Technical Report 3101, IBM, 2004.
4. S.Tyszberowicz B.Litvak and A.Yehudai. Behavioral consistency validation of uml diagrams. In *1st IEEE International Conference on Software Engineering and Formal Methods(SEFM), pages 118-125. IEEE Computer Society*, 2003.
5. A. Egyed. Scalable consistency checking between diagrams: The Viewintegra approach. In *Proc. 16th IEEE ASE*, San Diego, USA, 2001.
6. G. Engels, *et al*. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *The Proc. FSE-10*, Austria, 2001.
7. A. Evans, *et al*. Developing the UML as a formal modelling notation. In *Proc. UML'98, LNCS 1618*. Springer-Verlag, 1998.
8. M. Fowler. What is the point of UML. In P. Srevens, J. Whittle, and G. Booch, editors, *<<UML>> 2003 -The Unified Modeling Language, 6th International Conference, LNCS 2863*, San Fancisco, CA, USA, 2003. Springer.
9. J.M.Kuester G.Engels and L.Groenewegen. Consistent interaction of software components. In *IDPT2002*, 2002.
10. D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Israel, September 2000.
11. J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems (invited talk). In *Proc. ICCI02, Alberta, Canada*. IEEE Computer Society, 2002.
12. J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Pro. APLAS'2004, LNCS*, Taiwan, 2004. Springer.
13. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. P. Kruchten. *The Rational Unified Process – An Introduction (2nd Edition)*. Addison-Wesly, 2000.
15. L. Kuzniarz, G. Reggio, J.L. Sourruille, and Z. Huzar. Consistency problems in uml-based software development. In *Consistency Problems in UML-based Software Development: Workshop Materials*, 2002.
16. C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
17. X. Li, Z. Liu, J. He, and Q. Long. Generating a prototype from a UML model of system requirements. In *International Conference on Distributed Computing and Internet Technologies, (ICDCIT'04), LNCS 3347*, Bhubaneswar, India, 1999. Springer.
18. J. Liu, Z. Liu, J. He, and X. Li. Linking UML models of design and requirement. In *Pro. ASWEC'2004*, Melbourne, Australia, 2004. IEEE Computer Sciety.
19. Z. Liu. Object-oriented software development in UML. Technical Report UNU/IIST Report No. 259, UNU/IIST, P.O. Box 3058, Macau, SAR, P.R. China, July 2002.
20. Z. Liu, J. He, and X. Li. A rigorous approach to UML-base development. In A. Mota and A. Moura, editors, *Brazilian symposuim on Formal Formal Methods (SBMF04),*, Recife, Brazil, 2004. Editora Universitaria UFPE.

21. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.
22. Z. Liu, X. Li, and J. He. Using transition systems to unify uml models. Technical report, Dept. of Maths and Computer Science, the University of Leicester, England., May 2002.
23. Q. Long, Z. Liu, X. Li, and J. He. Consistent code generation from UML models. In *Pro. of Australian Software Engineering Conference (ASWEC'2005)*, Brisbane, Australia, 2005. IEEE Computer Sciety.
24. S.J. Mellor and M.J. Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, 2002.
25. G. Reggio, *et al*. Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *Proc. FASE 2001, LNCS 2029*. Springer, 2001.
26. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
27. J. Warmer and A. Kleppe. *The Object Constraint Language: precise modeling with UML*. Addison-Wesley, 1999.

An Automatic Mapping from Statecharts to Verilog

Viet-Anh Vu Tran[1], Shengchao Qin[2,3], and Wei Ngan Chin[2,3]

[1] Vietsoftware Company, Hanoi, Vietnam
tran.vu.viet.anh@vietsoftware.com
[2] Singapore-MIT Alliance
[3] National University of Singapore
{qinsc, chinwn}@comp.nus.edu.sg

Abstract. Statecharts is a visual formalism suitable for high-level system specification, while Verilog is a hardware description language that can be used for both behavioural and structural specification of (hardware) systems. This paper implements a semantics-preserving mapping from Graphical Statecharts to Verilog programs, which, to the best of our knowledge, is the first algorithm to bridge the gap between Statecharts and Verilog, and can be embedded into the hardware/software co-specification process [19] as a front-end.

1 Introduction

Statecharts [6, 7] is a visual formalism catering for high-level behaviourial specification of embedded systems. Its hierarchical structure, orthogonal and broadcast communication features make the system specification compact and intuitive to understand. It is a very good candidate for executable specification in system design [8]. Moreover, the semantics of Statecharts has been extensively investigated [9, 12, 14, 15, 13] in recent years. Some works also attempt to provide tools for formal verification of Statecharts specifications [4], [14], [20].

Verilog [22], [17] is a widely used language for hardware description in industry [2], [5], [11], [10] and also in research. Verilog is used to model the structure and behaviour of digital systems ranging from simple hardware building block to complete systems. Verilog semantics is based on the scheduling of events and the propagation of changes. One early attempt to investigate the semantics of Verilog is the work of Gordon [5] which explains how top-level modules can be simulated.

A Verilog program (or specification, as it is more frequently referred to) is a description of a device or process rather similar to a computer program written in C or Pascal. However, Verilog also includes constructs specifically chosen to describe hardware. One major difference from a language like C is that Verilog allows processes to run in parallel. This is obviously very desirable if one is to exploit the inherently parallel behaviour of hardware. In this work, we will make use of abstract Verilog [10], [18], that is described in the next chapter.

On the other hand, Verilog is a hardware description language that has been widely used by hardware designers. Its rich features make it a good candidate for low–level system specifications. The formal semantics of Verilog was first given by Gordon [5] in terms of simulation cycles. It has been thoroughly investigated afterwards [25], [24].

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 187–203, 2005.
© Springer-Verlag Berlin Heidelberg 2005

As the advantages of Statecharts and Verilog in embedded system design process are complementary to each other, a natural question that can be raised is, can we make use of both of them in system design? That is, can we use Statecharts as the high level specification, while use Verilog as the low level description? This question has motivated our work and this paper shall provide a positive answer by bridging the gap between Statecharts and Verilog. The compilation from Statecharts to Verilog can be embedded into the hardware/software co-specification process [19]. A mapping algorithm will be given in the following sections, where the soundness has been given in Qin and Chin [18].

The rest of this paper is organized as follows. Sec 2 gives a brief introduction to Statecharts and Verilog. Sec 3 presented the formal definition of the mapping function, followed by its implementation in Sec 4. Sec 5 illustrates our mapping results using two examples, while Sec 6 concludes the paper.

2 Preliminaries

2.1 Formal Syntax of Statecharts

Statecharts is a specification language derived from finite-state machines. The language is rather rich in features including state hierarchy and concurrency. Transitions can perform nontrivial computations unlike finite-state machines where they contain at most input/output pairs. In this section we will describe Statecharts presented by David Harel [6], [7], [9].

Statechart diagrams capture the behaviour of entities capable of dynamic behaviour by specifying their responses to the event occurrences. Typically, it is used for describing the behaviour of classes, but statecharts may also describe the behaviour of other model entities such as use cases, actors, subsystems, operations, or methods.

We use a simple textual representation of Statecharts, while our system can automatically translate a graphical representation to the textual representation. The statecharts language we adopt has some features that are not present in UML statecharts. For example, broadcast communication is supported in our language but not in UML statecharts.

As already mentioned in previous section, Statecharts is extensible by hierarchy, orthogonality or broadcast communication. In this paper, we use the formal syntax of statechart from [7] and [18]. The syntax of Statecharts formula is defined as follows (quoting from [18]):

$\mathcal{S}$: a set of names used to denote Statecharts. This is expected to be large enough to prevent name conflicts.
Π_e : a set of all abstract events (signals). We also introduce another set $\overline{\Pi}_e$ to denote the set of negated counterparts of events in Π_e , i.e. $\overline{\Pi}_e =_{df} \{\overline{e} \mid e \in \Pi_e\}$, where $\overline{e}$ denotes the negated counterpart of event e, and we assume $\overline{\overline{e}} = e$.
Π_a : a set of all assignment actions of the form $v = exp$.
$\sigma : Var \rightarrow Val$ is the valuation function for variables, where Var is the set of all variables, Val is the set of all possible values for variables. A snapshot for variables $\overline{v}$ is $\sigma(\overline{v})$.
$\mathcal{T}$: a set of transitions, which is a subset of $\mathcal{S} \times 2^{\Pi_e \cup \overline{\Pi}_e} \times 2^{\Pi_e \cup \Pi_a} \times \mathcal{B}_e \times \mathcal{S}$, where $\mathcal{B}_e$ is the set of boolean expressions.

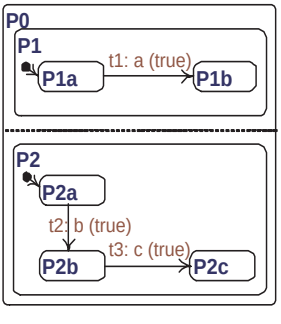
A term-based syntax of statecharts was introduced in [18] and [14], [15]. We re-introduce it here for the benefit of the reader. The set SC is a set of Statecharts terms that is constructed by the following inductively defined functions.

$$\texttt{Basic} : \mathcal{S} \rightarrow \texttt{SC}$$
$$\texttt{Basic}(s) =_{df} |[s]|$$
$$\texttt{Or} : \mathcal{S} \times [\texttt{SC}] \times \mathcal{T} \rightarrow \texttt{SC}$$
$$\texttt{Or}(s, [p_1, ..., p_l, ..., p_n], p_l, T) =_{df} |[s : [p_1, ..., p_l, ..., p_n], p_l, T]|$$
$$\texttt{And} : \mathcal{S} \times 2^{\texttt{SC}} \rightarrow \texttt{SC}$$
$$\texttt{And}(s, \{p_1, ..., p_n\}) =_{df} |[s : \{p_1, ..., p_n\}]|$$

Note that:
– $\texttt{Basic}(s)$: denotes a basic statechart named s.
– $\texttt{Or}(s, [p_1, ..., p_l, ..., p_n], p_l, \mathcal{T})$: represents an Or-statechart with a set of sub-states $\{p_1, ..., p_n\}$, where p_1 is the default sub-state, p_l is the current active sub–state, T is composed of all possible transitions among immediate sub-states of s.
– $\texttt{And}(s, \{p_1, ..., p_n\})$ is an And-statechart named s, which contains a set of orthogonal (concurrent) sub-states $\{p_1, ..., p_n\}$.

In this paper we use sub-state interchangeable as children of Or-state. Correspondingly, we use children and region of And-state interchangeably. For statecharts that we adopted in this work, we shall assume that each And-state will have at least two regions. Furthermore, each region shall be an Or-state.

We shall take the textual representation of statecharts as input data for our core mapping program. Our front-end algorithm will translate graphic charts to textual representation automatically. As an example, we give below a simple graphical Statechart and its corresponding textual representation.




```
P0 = |[ S1:  P1, P2  ]|
P1 = |[ S2: [ P1a, P1b ], P1a,  t1  ]|
P2 = |[ S3: [ P2a, P2b, P2c ], P2a,  t2, t3 ]|
P1a = |[ S4 ]|
P1b = |[ S5 ]||
P2a = |[ S6 ]|
P2b = |[ S7 ]|
P2c = |[ S8 ]|

t1 = < P1a,  a ,  , true, P1b >
t2 = < P2a,  b ,  , true, P2a >
t3 = < P2b,  c ,  , true, P2c >
```

Fig. 1. A simple example of a Statechart and its textual representation

2.2 Verilog

Verilog is a hardware description language that has been widely used in industry. Although the Verilog IEEE standard [22] was released around ten years ago, the formal semantics based on simulation cycles [5] has not been well-investigated until recently, e.g. [11], [10]. In our work, we shall use a behaviourial subset of Verilog introduced in

[10] and [18]. This more abstract version of Verilog can be used to express designs at various levels of hardware behaviour. Such an abstract design can be gradually refined into an equivalent counterpart in the Verilog HDL which can provide a closer match to the underlying architecture of the hardware. This process may be repeated until the design is at a sufficiently lower level such that the hardware device can be synthesised from it. There are two main features in abstract Verilog that are not present in Verilog HDL, namely guarded choice extension and recursion. The translation from general guarded choices to parallel composition in normal Verilog is achievable, although nontrivial. The conversion of recursion to iteration is harder but there exists standard conversion techniques to realise some subsets of them. Furthermore, for bounded recursion, it is possible to inline the abstract Verilog code so as to remove recursion.

A Verilog program can be a parallel or a sequential process, but only parallel process may contain sequence processes, not vice-versa. Here are some categories of syntactic elements:

1. Parallel process
   $P \ ::= \ S \ | \ P \parallel \ P$
   where, S is a sequential process.
2. Sequential process can be formally described as following
   $S \ ::= PC$ (primitive command) $\ | \ S; S$ (sequential composition)
   $\quad | \ s \lhd b \rhd S$ (condition) $\ | \ b * S$ (iteration)
   $\quad | \ (b \& g \ S) \ [] \ ... \ [] \ (b \& g \ S)$ (guarded choice) $\ | \ fix \ X \bullet S$ (recursion)

   where, b is boolean condition, and
   $PC ::= skip \mid sink \mid \perp \mid \ \rightarrow \ \eta$ (output event) $\mid v \ = \ ex$ (assignment)
   $g \quad ::= \rightarrow \ \eta \mid @(x = v)$ (assignment guard))
   $\quad \mid \#1$ (time delay) $\mid eg$ (event control)
   $eg \ ::= \eta \mid eg \ \& \ eg \mid eg \ \& \ \neg eg$
   $\eta \quad ::= \uparrow \ v$ (value rising) $\mid \ \downarrow \ v$ (value falling) $\mid \underline{e}$ (a set of abstract events)

Recall that a Verilog program can only be a parallel process at the top level, a sequential process cannot contain a parallel process. However, most real systems contain many parallel processes possibly organised hierarchically. To solve this restriction, we shall use algebraic laws [10] to expand a parallel process into a sequential one.

Here are some simple code examples:

- $(e \ \& \ (\rightarrow f) \ sink) \ [] \ (g \ \& \ (\rightarrow h) \ sink)$
- $\mu X \bullet (e \ (f \ X) \ )$
- $(a \ \& \ (\rightarrow e) \ sink) \parallel (b \ \& \ (\rightarrow f) \ sink)$

3 Semantic-Preserving Mapping

Our algorithm that takes as input graphical statecharts and generates as output Verilog code is based on the theoretical result presented in [18]. This mapping algorithm works in a top-down manner starting from the root of the statechart and then moving to its children. Each time, we consider the input statechart (each part of Statecharts) as a singleton statechart and continue until no further applicable.

We present the mapping function L as originally proposed in [18] which produces result based on the type of the source statechart:

Definition of Mapping Function L:

$$L : \texttt{SC} \rightarrow \texttt{Verilog}$$

maps any statechart description into a corresponding Verilog process. It keeps unchanged the set of variables employed by the source description, i.e.,

$$\forall sc \in \texttt{SC} \bullet \mathbf{vars}(L(sc)) = \mathbf{vars}(sc)$$

and it is inductively defined as follows.

- For a statechart $sc = |[s]|$ constructed by `Basic`, L maps its input into an idle program *sink* which can do nothing but let time advance, i.e.,
  $$L(sc) =_{df} \textit{sink}$$
- For a statechart $sc = |[s : \{p_1, ..., p_n\}]|$ constructed by `And`, L maps its input into a parallel construct in Verilog.
  $$L(sc) =_{df} \|_{1 \le i \le n} L(p_i)$$
- For a statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$ constructed by `Or`, we define L by exhaustively figuring out the first possible transitions of sc if any, otherwise it returns *sink*.
  $$L(sc) =_{df} \begin{cases} \textit{sink} & \text{if } T^*(sc) = \emptyset \\ P & \text{otherwise} \end{cases}$$
  where
  $$\begin{aligned} P =_{df}\ & []_{0 \le k \le \textit{or-depth}(sc)} []\{b_{\tau_k}\ \&\ g^i_{\tau_k}\ \&\ (\&_{0 \le j \le k}\ h_j)\ \&\ g^0_{\tau_k}\ L(resc(\tau_k, sc))\ \mid \\ & \tau_k \in T(active^k(sc))\ \wedge\ src(\tau_k) = active^{k+1}(sc)\ \wedge \\ & h_j = \&\{\neg g^i_\tau \mid \tau \in T(active^{j-1}(sc))\ \wedge\ src(\tau) = active^j(sc)\}\} \end{aligned}$$
  and
  $$\begin{aligned} active^0(sc) &=_{df} sc \\ active^1(sc) &=_{df} active(sc) \\ active^{i+1}(sc) &=_{df} active(active^i(sc)) \end{aligned}$$

For each statechart, we always assume each of its variables has bounded range, and the set of possible events is finite, which implies that the set of its configurations is finite. Therefore, the set of configurations (under transition relation) forms a well–founded quasi order, which indicates the mapping function L is terminating.

Following are some formal notations used in the above definition. Firstly, the function *or-depth* : $\texttt{SC} \rightarrow N$ to calculate the "or–depth" of a statechart, which is defined as follows:

- for a statechart $sc = |[s]|$ constructed by `Basic`, *or-depth*$(sc) =_{df}\ 0$;
- for a statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$ constructed by `Or`, *or-depth*$(sc) =_{df}$ *or-depth*$(p_l) + 1$;
- for a statechart $sc = |[s : \{p_1, ..., p_n\}]|$ constructed by `And`, *or-depth*$(sc) =_{df}\ 1$.

The *or-depth* of an `Or`-chart just records the depth of the path transitively along its active `Or`-sub-states. We stop going further once an `And`-state is encountered. The *or-depth* of an `And`-chart is simply 1.

Secondly, the source and target state functions, $src(\tau)$ and $tgt(\tau)$, respectively represent the source and target state of a transition τ. Given a transition $\tau = \&_{1 \le k \le m} \tau_{i_k} \in T$,

where $\tau_{i_k} \in T^*(p_{i_k})$, for $1 \leq k \leq m$, and $i_1, ..., i_n$ is a permutation of $1, ..., n$, we define its source and target state as follow:

$src(\tau) =_{df} (q_1, ..., q_n)$, where $q_{i_k} = src(\tau_{i_k})$, for $1 \leq k \leq m$, and $q_{i_k} = active(p_{i_k})$, for $m < k \leq n$;

$tgt(\tau) =_{df} (r_1, ..., r_n)$, where $r_{i_k} = tgt(\tau_{i_k})$, for $1 \leq k \leq m$, and $r_{i_k} = active(p_{i_k})$, for $m < k \leq n$.

Note that $T^*(p)$ contains all possible transitions inside p along its transitive active sub-state chain, i.e., $T^*(p) =_{df} \{\tau \mid \tau \in T \wedge src(\tau) = p_l\} \cup T^*(p_l)$. And $active(sc)$ denotes a current active sub-state of sc. With an `Or`-statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$, we have $active(sc) = p_l$. With an `And`-statechart $sc = |[s : \{p_1, ..., p_n\}]|$, we have the active state as a vector of the active states of these constituents, i.e., $active(sc) =_{df} (active(p_1), ..., active(p_n))$.

Thirdly, we need to know the resulting statechart after a transition is taken. When a transition τ occurs, any involved statechart can have changes in its (transitive) active sub-states. We use a function:

$resc : \mathcal{T} \times \texttt{SC} \rightarrow \texttt{SC}$

to return the modified statechart after performing a transition in a statechart. It is defined inductively with regard to the type of the statechart.

- for a `Basic`-statechart sc, and any transition τ, $resc(\tau, sc) =_{df} sc$;
- for an `Or`-statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$, and a transition τ,

$$resc(\tau, sc) =_{df} \begin{cases} sc_{[l \mapsto a2d(tgt(\tau))]}, \text{if } \tau \in T \wedge src(\tau) = p_l; \\ sc_{[l \mapsto resc(\tau, p_l)]}, \text{if } \tau \in T^*(p_l); \\ sc, \text{otherwise.} \end{cases}$$

- for an `And`-statechart $sc = |[s : \{p_1, ..., p_n\}]|$, and a transition τ,

$$resc(\tau, sc) =_{df} \begin{cases} sc_\tau, \text{if } \tau = \&_{1 \leq k \leq m} \tau_{i_k} \in T(sc); \\ sc, \text{otherwise.} \end{cases}$$

where $sc_\tau = sc[q_1/p_1, ..., q_n/p_n]$ is the statechart obtained from sc via replacing p_i by q_i, for $1 \leq i \leq n$, $q_{i_k} = resc(\tau_{i_k}, p_{i_k})$, for $1 \leq k \leq m$, and $q_{i_k} = p_{i_k}$, for $m < k \leq n$.

The function $a2d(sc)$ is used to change the active sub-state of sc into its default sub-state, and the same change is applied to its new active sub-state. This function is defined as:

- $a2d(|[s]|) =_{df} |[s]|$
- $a2d(|[s : [p_1, ..., p_n], p_l, T]|) =_{df} |[s : [p_1, ..., p_n], a2d(p_1), T]|$
- $a2d(|[s : \{p_1, ..., p_n\}]|) =_{df} |[s : \{a2d(p_1), ..., a2d(p_n)\}]|$

The substitution $sc_{[l \mapsto p_m]}$ for an `Or`-statechart $sc = |[s : [p_1, ..., p_n], p_l, T]|$ is defined by $sc_{[l \mapsto p_m]} =_{df} |[s : [p_1, ..., p_n], p_m, T]|$

4 Implementation

Our implementation consists of two parts: a statechart editor (called Statechart_E, is a stencil of MS Visio) and a mapping program from statechart into abstract Verilog (called AMSV-Automatic Mapping of Statechart into Verilog).

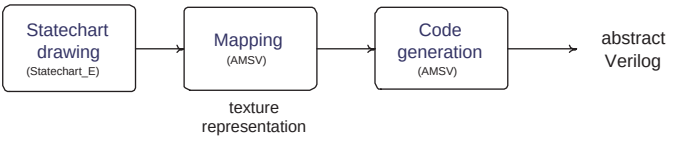


Fig. 2. Structure of the implementation

Fig. 2 shows the stages of using our system. Users first draw their statecharts, using Statechart_E, which also automatically generates the corresponding textual representations. AMSV will then generate abstract Verilog code from textual representation of these statecharts. In next two sections, we will discuss about Statechart_E, AMSV, and some other techniques used in the system.

4.1 Statechart Editor

Statechart_E is built with three main purposes:

- First, of course is for editing Statechart diagrams. The editor should be convenient to use and easy to draw.
- Second, it should also be easy to export textual representation of statechart. This is used by the mapping algorithm which converts statechart to abstract Verilog.
- Last, it should be easy to save the statecharts to other graphical formats (like bmp, jpg, ps, eps, etc) This is important for portability and for documentation.

From these requirements, we built Statechart_E as an add-on/embedded stencil in Microsoft Visio. We make use of MS. Visio because Visio is a very powerful graphical editor tool for drawing diagrams. Visio also supports many graphical formats for exporting our diagrams. Moreover, using Visio, we can not only draw statechart components but also other shapes from suitable drawing types or stencils.

Features of Statechart_E:

- A menu named *Statechart* is added to the menu bar of Visio. This menu contains two functions, namely: *Generate statechart* and *Add new statechart page*. The first function is used to export the current statechart to a textual file. This file is used as input for the mapping program which to transform to abstract Verilog. The second function is used to add a new page for current statechart diagram. To enable this menu and its functions, users must allow a macro to be accepted when opening the stencil.
- A set of masters is added to the stencil and this is used for constructing statecharts. It consists of a state master, a default master (common for all kind of states), 8

transition masters (to help build complex statecharts), and vertical/horizontal separators for And-state.
- Each master is accompanied by a program written in Visual Basic for Application (VBA) to check data, events and perform actions of each master. Some masters are linked to a window to allow input of needed data. This program also partially checks the supplied data such as duplicate name, etc.
- We also allow users to build hierarchical statecharts. Users can easily extend a given statechart by adding a new page (using the second function in menu *Statechart*) and continue to extend the current statechart in a hierarchical manner in the new page. Note that the *generate* function will read all components in all pages of the statechart.

4.2 AMSV - Core Mapping Program

The second part, called AMSV (Automatic Mapping of Statechart into Verilog), is essentially a Java program.

DFS Algorithm. As presented in section 3, the mapping algorithm has to deal with each state; `Basic`, `And`, and `Or` states. It can construct the corresponding Verilog code after the mapping algorithm has been applied to all states of the source statechart. Nevertheless, how do we traverse all states of the input statechart? In the AMSV, we make use of depth–first–search (DFS) algorithm [3] to reach all states of the statechart.

However, DFS works on each tree of nodes. To apply DFS we have to reconstruct the source statechart into a tree of states. Fig. 3 shows an example of hierarchy tree (b) for a simple statechart (a). Here, dashed arrows denote the children of an `And`-state (like arrow from P0 to P1, P2), while the doted arrows point to the active sub-states of `Or`-state (like arrow from P1 to P3 or P2 to P6). The solid arrows represent the transitions.

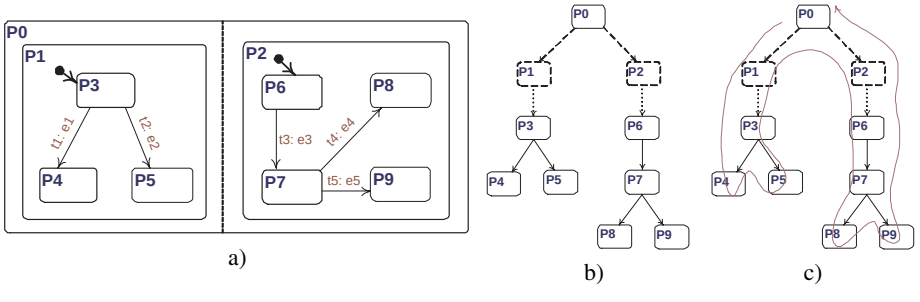


Fig. 3. Hierarchy tree. a) Statechart example, b) hierarchy tree, and c) DFS route

After reconstructing each statechart into a hierarchy tree, we apply a recursive function which maps each statechart to abstract Verilog. At each time, we only consider one state, called the current state. Through this recursive function, we apply the mapping

algorithm to all states of the source statechart to obtain Verilog process code. These codes are kept in a hash table for latter use. After that, we gather the output code (from sub-states or from target states of all transitions to the current state) to generate final abstract Verilog process.

For example, in the Fig. 3, first we start from the root state (like P0). After that, we invoke the function itself if it is possible to go to current state's children (P1, P2) or target states of transitions (P3 to P4, P5). A systematic way of finding the next state is described below. Fig. 3 c shows the route taken by our DFS traversal:

- each state is the target of transition: If there exists any transition from the current state, go to the target state of the transition. Like transitions from P3 to P4 or P5. The information of the transition will be memorized to generate output code. If there are more than one transitions from current state, process it one by one. The order between these transitions is not important.
- each state is a child of the `And`-state: If the current state is `And`-state, go to all children. Like from P0 to P1 or P2. Information of children in that `And`-state will be memorized during code generation, as acquired by the Verilog language.
- state is sub-state of `Or`-state: Just go to active state and continue as before. For example, P3 and P6 are the active states of P1 and P2.

Recursion. During the traversal to the states of a given statechart, it is possible for a transition to re-occur. This may be due to non-termination. To solve this problem we use a boolean array to remember all states which the program has already encountered. If a program reaches a marked state, it just uses that information to generate a loop, and then go back to previous state. This is meant to terminate a recursive transition.

Parallel Expansion. Recall from early discussion in Sec 2, we shall take into account the parallel expansion of `And`-state. Whenever an `And`-state is reached, all information (guards, conditions, etc) of the children of a current state are used for expansion. The only exception is when the current state is the root. In this case we generate Verilog code from all its children and gather it using the parallel operation ($\|$). This situation was discussed in [23].

5 Examples

In this section, we illustrate the mapping algorithm via the following examples: a CD player and a washing machine.

5.1 CD-Player

Specification. Fig. 4 shows the graphical statechart of a CD-player. It contains two orthogonal regions: *Play control* (`PlayCtr`) and *Track information* (`TrackCtr`), which are used to control the playing mode and record the track information respectively. The first region contains `Stop, Play, Pause` sub-states to control the playing mode,

while the second one contains only a sub-state, `Track`. Three buttons, `Next`, `Prev`, and `select a track`, are associated with the `Track` state. The variable ct (that is, current track) is used to keep record of the current position of the CD being played. We assume ct is initially 0 whenever the CD-player is switched on.

In this model, `Stop` and `Track` are respectively two default sub-states of two orthogonal regions. So when the CD-Player is switched on, both of them are entered simultaneously. Upon the arrival of event *Play_pressed* (that is, the `Play` button is pressed), transition $t1$ is taken and state `PlayingCtr` is entered, where the default sub-state `Playing` becomes active. Transitions $t4$ and $t3$ are used to alter between state `Playing` and `Paused`. Transition t2 connects state `PlayingCtr` with state `Stop`. When the control is in state `PlayingCtr` (either `Playing` or `Paused`), and $t2$ is enabled, it will yield the `Stop` state (that is, the CD-player will stop).

In the orthogonal state `TrackCtr`, upon the arrival of events *Next_pressed* or *Prev_pressed*, the variable ct (current track) will be changed according to the event. Conditions $(ct > 1)$ and $(ct < Max(track))$ are used to check the range of the ct. The transition $t7$ is taken if users select any track in the range.

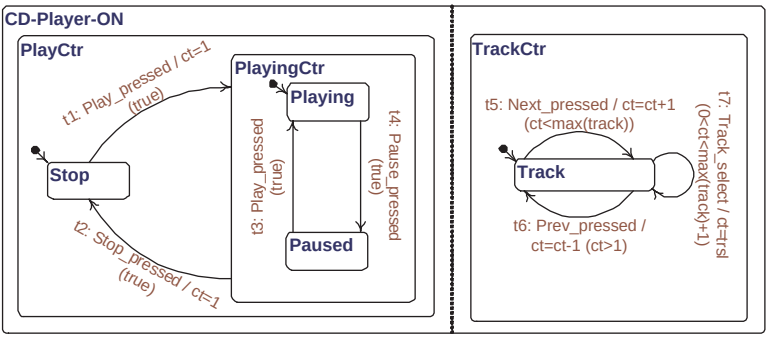


Fig. 4. CD player with track information (ct)

For simplicity, we only added track information in this specification of a CD-player. A real CD-player may contain other functionalities, like timer, forward, rewind, etc. We can add these setting as parallel regions in a similar way.

After drawing the statechart specification in Statechart_E, the following textual representation is automatically generated:

```
CD-Player-ON = |[ S1: { PlayCtr, TrackCtr } ]|
PlayCtr = |[ S2: [ Stop, PlayingCtr ], Stop, { t1, t2 } ]|
TrackCtr = |[ S3: [ Track ], Track, { t5, t7, t6 } ]|
Stop = |[ S4 ]|
PlayingCtr = |[ S5: [ Playing, Paused ], Playing, { t3, t4 } ]|
Playing = |[ S6 ]|
Paused = |[ S7 ]|
Track = |[ S8 ]|
```

```
t1 = < Stop, { Play_pressed }, { ct=1 }, true, PlayingCtr >
t2 = < PlayingCtr, { Stop_pressed }, { ct=1 }, true, Stop >
t3 = < Paused, { Play_pressed }, {  }, true, Playing >
t4 = < Playing, { Pause_pressed }, {  }, true, Paused >
t5 = < Track, { Next_pressed }, { ct=ct+1 }, ct<max(track),
       Track >
t7 = < Track, { Track_select }, { ct=trsl }, 0<ct<max(track)+1,
       Track >
t6 = < Track, { Prev_pressed }, { ct=ct-1 }, ct>1, Track >
```

The first 8 lines are information of states. The rest are transitions.

Result. The textual representation given in last section is taken as the input of our algorithm AMSV, the output we obtain is the following code in abstract Verilog:

```
Result:
L_PlayCtr || L_TrackCtr

Where:
L_PlayCtr =  fix X0. ( L_Stop )
L_TrackCtr =  fix X2. (
  ( ( ( Next_pressed & @( ct=ct+1 ) & ( ct<max(track) ) X2 )
    [] ( Track_select & @( ct=trsl ) & ( 0<ct<max(track)+1 ) X2 ) )
  [] ( Prev_pressed & @( ct=ct-1 ) & ( ct>1 ) X2 ) ) )
L_Stop = ( ( Play_pressed & @( ct=1 ) )
  ( ( Stop_pressed & @( ct=1 ) X0 ) []  fix X1. ( L_Playing ) ) )
L_Playing = ( ( Pause_pressed &  not Stop_pressed )
              ( ( ( Play_pressed &  not Stop_pressed ) X1 )
                [] ( Stop_pressed & @( ct=1 ) X0 ) ) )
```

note that we use fix (rather than μ) to denote the recursion. L_state is the corresponding result from $state$.

Here we can see that the L_PlayCtrl and L_TrackCtr are processes which are running in parallel, where the recursive identifiers X0, X1, X2 represent three loop points.

5.2 Washing Machine

Specification. In this subsection, we discuss a washing machine with five setting functions; `Timer`, `Hot water`, `Rinse level`, `Water level`, and `Pre-wash`. Fig. 5 shows the user interface of the washing machine. Fig. 6 gives the statechart specification of the washing machine corresponding to the interface, while Fig. 7 zooms into the substate `Washing-Ctr`. Statechart in Fig. 6 contains six parallel regions corresponding to five setting functions and the washing progress (*Wash-Ctr*). Each setting region contains a sub-statechart to change the value of its function. For example, in the `Timer-Ctr` region, the variable tm denotes the time that the washing machine has to wait before it starts to wash. It can be changed by `Inc` or `Dec` buttons. Other variables hw (hot water), rl (rinse level), wl (water level) and pw (pre-wash) are similar, and can be changed via pressing corresponding buttons. The default values of these variables are shown in Fig. 5 with black circles ($hw = 0$, $rl = 0$, $wl = 0$, and $pw = 0$) and default timer is 0.

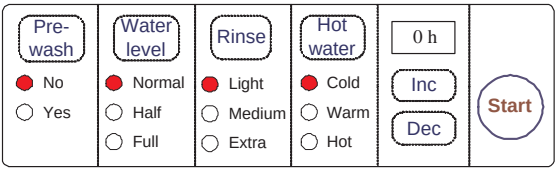


Fig. 5. Interface of the washing machine

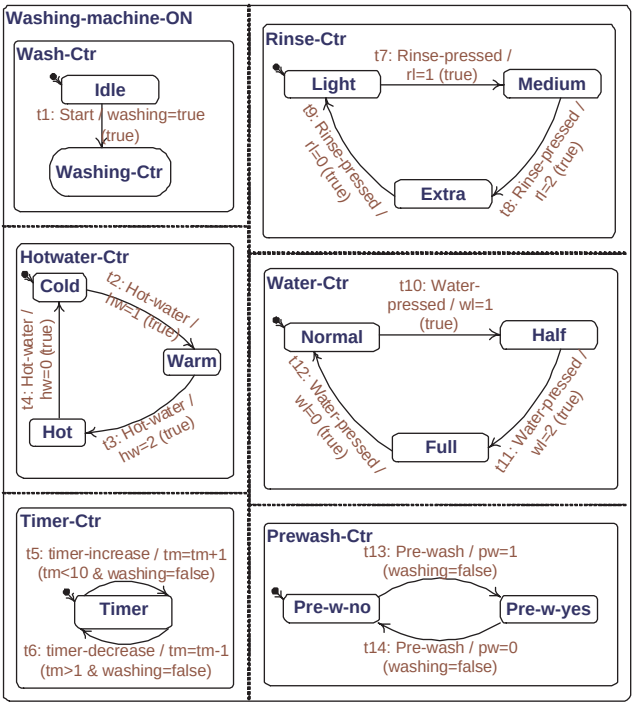


Fig. 6. Main statechart of a washing machine

The `Washing-Ctr` is an `Or`-state as given in Fig. 7. The state `Check-wait` is activated once state `Washing-Ctr` is entered. If tm is greater than 0, the machine keeps waiting for tm time before the control moves to `Pre-wash` state. The transition $t18$ calculates the value of the variable $washtime$ based on the *pre-wash* setting. For example, if pw is 0 then $washtime = 1$. The variable $washtime$ is used to keep record of the time that the clothes have been washed so far. It is explained as follows:

- $washtime = 0$: if $pw = 1$, need pre-wash.
- $washtime = 1$: if $pw = 0$, no need pre-wash, need powder, no spin.
- $washtime = 2$ *or* 3: wash without powder, spin.
- $washtime > 3$: finish.

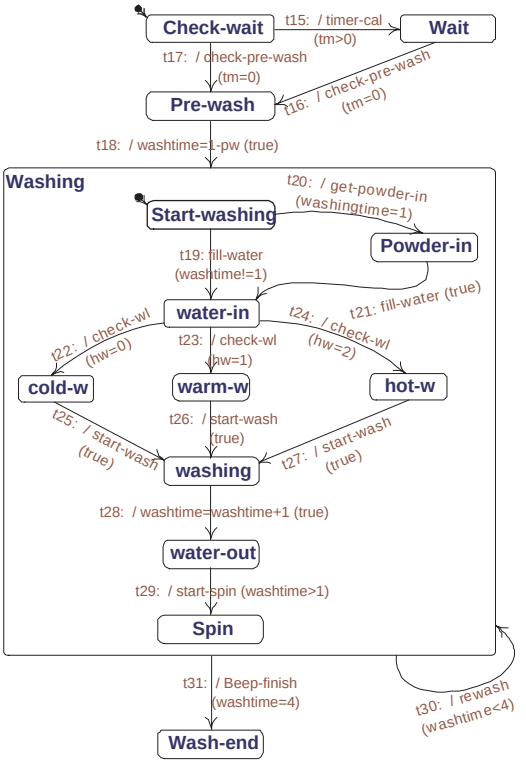


Fig. 7. Statechart of `Washing-Ctr` in the washing machine

Upon finishing, the machine beeps to inform the user.
The textual representation generated from Statechart_E is printed in [23].

Result. We then run the AMSV algorithm to generate the Verilog program for the washing machine. We only give some part of the target code here.
First of all, let us regard `Washing-Ctr` as a basic state (before we zoom into it). We have the following Verilog program:

```
Result:
L_Wash-Ctr || L_Timer-Ctr || L_Water-Ctr || L_Prewash-Ctr ||
  L_Hotwater-Ctr || L_Rinse-Ctr

Where:
L_Wash-Ctr = L_Idle
L_Idle = ( Start & @( washing=true ) sink )
L_Timer-Ctr =
  fix X0. ( ( ( timer-increase & @( tm=tm+1 ) &
                ( tm<10 & washing=false ) X0 )
              [] ( timer-decrease & @( tm=tm-1 ) &
                   ( tm>1 & washing=false ) X0 ) ) )
```

```
L_Water-Ctr =  fix X1. ( L_Normal )
L_Normal = ( ( Water-pressed & @( wl=1 ) ) L_Half )
L_Half = ( ( Water-pressed & @( wl=2 ) )
           ( Water-pressed & @( wl=0 ) X1 ) )
L_Light = ( ( Rinse-pressed & @( rl=1 ) ) L_Medium )
L_Medium = ( ( Rinse-pressed & @( rl=2 ) )
             ( Rinse-pressed & @( rl=0 ) X4 ) )
L_Prewash-Ctr =  fix X2. ( L_Pre-w-no )
L_Pre-w-no = ( ( Pre-wash & @( pw=1 ) & ( washing=false ) )
               ( Pre-wash & @( pw=0 ) & ( washing=false ) X2 ) )
L_Hotwater-Ctr =  fix X3. ( L_Cold )
L_Cold = ( ( Hot-water & @( hw=1 ) ) L_Warm )
L_Warm = ( ( Hot-water & @( hw=2 ) ) ( Hot-water & @( hw=0 ) X3 ) )
L_Rinse-Ctr =  fix X4. ( L_Light )
```

The `sink` process in `L_Idle` is used to denote the `Washing-Ctrl` process, as we regard it as a basic state. On the other hand, if we consider `Washing-Ctr` as a stand-alone statechart, the corresponding code for it is as follows:

```
Result:
L_Check-wait =
  ( ( (  & @( timer-cal ) & ( tm>0 ) ) L_Wait )
    [] ( (  & @( check-pre-wash ) & ( tm=0 ) ) L_Pre-wash ) )
L_Start-washing =
  ( ( ( fill-water & ( washtime!=1 ) ) L_water-in
      ( & @( rewash ) & ( washtime<4 ) X0 ) )
    [] ( ( & @( get-powder-in ) & ( washingtime=1 ) ) L_Powder-in
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
L_Wait = ( (  & @( check-pre-wash ) & ( tm=0 ) ) L_Pre-wash )
L_Pre-wash = ( (  & @( washtime=1-pw ) )
               fix X0. ( ( ( & @( rewash ) & ( washtime<4 ) X0 )
                         [] L_Start-washing ) ) )
L_water-in =
  ( ( ( (  & @( check-wl ) & ( hw=0 ) ) L_cold-w
        ( & @( rewash ) & ( washtime<4 ) X0 ) )
    [] ( (  & @( check-wl ) & ( hw=2 ) ) L_hot-w
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
    [] ( (  & @( check-wl ) & ( hw=1 ) ) L_warm-w
         ( & @( rewash ) & ( washtime<4 ) X0 ) ) )
L_cold-w = ( (  & @( start-wash ) ) L_washing
             ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_warm-w = ( (  & @( start-wash ) ) L_washing
             ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_hot-w = ( (  & @( start-wash ) ) L_washing
            ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_washing = ( (  & @( washtime=washtime+1 ) ) L_water-out
              ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_water-out = ( (  & @( start-spin ) & ( washtime>1 ) ) L_Spin
                ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_Powder-in = ( ( fill-water ) L_water-in
```

```
                ( & @( rewash ) & ( washtime<4 ) X0 ) )
L_Spin = ( & @( Beep-finish ) & ( washtime=4 ) sink
           ( & @( rewash ) & ( washtime<4 ) X0 ) )
```

In the final code, the `sink` process in `L_Idle` is replaced by the process `L_Check-wait`.

6 Conclusion

In this paper we proposed an automatic mapping algorithm to translate high-level Statecharts into low-level Verilog specifications. Our algorithm has been proved sound earlier [18].

The system that we have built in Java provides a graphical interface for users to draw their statecharts in MS Visio. Our mapping algorithm thus translates the graphical representation into a textual representation, and then generates the corresponding Verilog programs.

Some of related works on connecting Statecharts with other formalisms are presented in [1, 4, 16, 21, 20]. Beauvais et.al. [1] and Seshia et.al. [21] translate STATEMATE Statecharts to synchronous languages *Signal* and *Esterel* respectively, aiming to use supporting tools provided in the target formalisms for formal verification purposes. However, all these translations are based on the informal semantics [9] lacking correctness proofs. The authors of [4, 16] transform variants of Statecharts into hierarchical timed automata and use tools (UPPAAL, SPIN) to model check Statecharts properties. More recently, a translation from Statecharts to B/AMN is reported in [20]. However, no correctness issue has been addressed. In comparison, the translation from Statecharts to Verilog in this paper aims at code generation for system design. The mapping function that we implement in this paper is constructed based on formal semantics for both the source and target formalisms and has been proven to be semantics-preserving [18].

Our compilation from Statecharts into Verilog can be used as a front-end of hardware design or hardware/software co-design. After translating the input statechart specification into abstract Verilog code, we can proceed to obtain lower level descriptions, as a prelude to hardware implementation, or we can pass the Verilog specification to a hardware/software partitioning system [19].

In order to provide the concrete Verilog programs to users, future works include guarded choices elimination and the replacement of the other structures of abstract Verilog, so that the AMSV can generate also concrete Verilog program. This should make our tool especially useful for hardware designer.

References

1. J.-R. Beauvais, et. al. A Translation of Statecharts to Signal/DC+. Technical report, IRISA, 1997.
2. J. P. Bowen, J.-F. He, and Q.-W. Xu. An Animatable Operational Semantics of the VERILOG Hardware Description Language. In *Proc. ICFEM2000: 3rd IEEE International Conference on Formal Engineering Methods, IEEE Computer Society Press*, York, UK, September 2000.

3. T. H. Cormena, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press; 2nd edition, September 2001.
4. A. David, M. Oliver Möller, and Wang Y. Formal Verification of UML Statecharts with Real-time Extensions. In *Proc. of Fundamental Approaches to Software Engineering*, number 2306 in Springer LNCS, 2002.
5. M. J. C. Gordon. The Semantic Challenge of Verilog HDL. In *Proc. Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press*, pages 136–145, June 1995.
6. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.
7. D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5), 1988.
8. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7), 1997.
9. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), October 1996.
10. J.-F. He. An Algebraic Approach to the VERILOG Programming. In *Proc. of 10th Anniversary Colloquium of the United Nations University / International Institute for Software Technology (UNU/IIST)*. Springer, 2002.
11. J.-F. He and H. Zhu. Formalising Verilog. In *Proc. IEEE International Conference on Electronics, Circuits and Systems, IEEE Computer Society Press*, Lebanon, December 2000.
12. J.J.M. Hooman, S. Ramesh, and W.P. de Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science*, 101, 1992.
13. Q. Long, Z.Y. Qiu, and S.C. Qin. The Equivalence of Statecharts. In *International Conference on Formal Engineering Methods*, number 2885 in Springer LNCS, Singapore, November 2003.
14. G. Lüttgen, M. von der Beeck, and R. Cleaveland. A Compositional Approach to Statecharts Semantics. Technical Report 200012, NASA/CR2000210086, ICASE Report, March 2000.
15. A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of Statecharts. In *7th International Conference on Concurrency Theory (CONCUR'96)*, number 1119 in Springer LNCS, Pisa, Italy, August 1996.
16. E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing Statecharts in Promela/SPIN. In *the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*. IEEE Computer Society, 1999.
17. Open Verilog International (OVI). *Verilog Hardware Description Language Reference Manual*.
18. S.C. Qin and W.N. Chin. Mapping Statecharts to Verilog for Hardware/Software Co-Specification. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Formal Methods: International Symposium of Formal Methods Europe*, volume 2805, pages 282–299. Springer, 2003.
19. S.C. Qin, J.F. He, Z.Y. Qiu, and N.X. Zhang. Hardware/Software Partitioning in Verilog. In *International Conference on Formal Engineering Methods*, number 2495 in Springer LNCS, Shanghai, China, October 2002.
20. E. Sekerinski and R. Zurob. Translating Statecharts to B. In B. Butler, L. Petre, , and K. Sere, editors, *Proc. of the 3rd International Conference on Integrated Formal Methods*, number 2335 in Springer LNCS, Turku, Finland, 2002.
21. S. Seshia, R. Shyamasundar, A. Bhattacharjee, and S. Dhodapkar. A Translation of Statecharts to Esterel. In J. Wing, J. Woodcock, and J. Davies, editors, *FM99: World Congress on Formal Methods*, number 1709 in Springer LNCS, 1999.
22. IEEE Standard. *IEEE Standard Hardware Description Language based on the Verilog® Hardware Description Language*. 1995.

23. V.-A. V. Tran. Automatic Mapping from Statecharts to Verilog. Master's Thesis, School of Computing, The National University of Singapore, 2004.
24. H. Zhu, J. P. Bowen, and J.-F. He. Deriving Operational Semantics from Denotational Semantics for Verilog. Technical report, Technical Report SBU-CISM-01-16, South Bank University, London, UK, June 2001.
25. H. Zhu, J. P. Bowen, and J.-F. He. From Operational Semantics to Denotational Semantics for Verilog. In *Proc. CHARME 2001: 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, number 2144 in Springer LNCS, Livingston, Scotland, September 2001.

Reverse Observation Equivalence Between Labelled State Transition Systems*

Yanjun Wen, Ji Wang, and Zhichang Qi

National Laboratory for Parallel and Distributed Processing,
Hunan 410073, P.R. China
{y.j.wen, ji.wang}@263.net

Abstract. Labelled state transition system (LSTS) is a formalism intended to combine the benefits of both state-based and action-based models. However, its existent equivalence preserves many properties with the cost of poor reduction effects. A new equivalence is presented, namely called *reverse observation equivalence* which is defined in the opposite direction to observation equivalence and orients the invariant checking of LSTS. Experiments show that the new semantics is efficient in the context of compositional reachability analysis.

1 Introduction

Labelled state transition system (LSTS) is introduce in [1] as a formalism which combines labelled transitions systems (LTSs) with atomic state propositions and provides a way to benefit from both state-based and action-based models. Concretely, similar to Kripke structures [2], the states of LSTSs are labelled with the possible interpretations to the propositions. A notable characteristic of LSTS is that the state propositions do not affect synchronization between LSTSs. That is to say, LSTSs communicate only by the actions, as same as LTSs do.

Invariants are the simplest but most important kind of system requirements [3]. It predicates that all the reachable states of a system satisfy some property, which is in the form of boolean expressions.

Compositional reachability analysis (CRA) is a kind of hierarchy-based incremental analysis approach [4, 5, 6, 7]. The mechanism of "intermediate simplification during composition" in CRA can significantly increase the size of systems which are analyzable with given computer resources [8, 9]. In CRA, the simplification is usually based on some equivalence semantics, such as strong bisimilarity, weak bisimilarity (or observation equivalence), and the failure-divergence model of CSP [10]. In the chosen semantics, the final model gotten by CRA is equivalent to the one gotten by the direct "all-at-once" approach [6] (called traditional reachability analysis in [5]).

* Supported by National Natural Science Foundation of China under the grants 60233020, 90104007, and 60303013, and by the National Hi-Tech Programme of China under the grant 2001AA113202.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 204–219, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In [1], a semantic model, CFFD (choas free failures divergence) is presented for LSTS, which is an extension to the CFFD model of LTS. In the extension, the state propositions of LSTSs are encoded via an attachment to actions. CFFD is the weakest congruence that preserves deadlocks and all the properties that can be expressed with next-state free LTL [1]. Thus, of course, it preserves invariants. Although CFFD preserves many properties, it brings only small reduction (see Section 5). For invariant checking of LSTS, some more efficient equivalence semantics should be possible and necessary.

At a glance, one may think of observation equivalence, which can bring significant reduction for the CRA of LTSs. Indeed, it is feasible to define the observation equivalence of LSTS similarly as the CFFD-equivalence of LSTS. In this way, we must also encode the state propositions via an attachment to actions. That is to say, we must extend the notion of actions to include the information of the change of states propositions as transitions triggered. Thus the action set will be enlarged and accordingly, the effect of reduction based on observation equivalence will be decreased. Fortunately, with a slight extension to LSTS, we may define an equivalence relation in the opposite direction to observation equivalence, and the invariant properties can be preserved without extending the notion of actions. We call such a new equivalence as *reverse observation equivalence* (ROE).

We have implemented a reduction algorithm based on reverse observation equivalence in the framework of TVT toolkit [1]. The experiments show that the new equivalence is much more efficient than CFFD-equivalence with respect to the invariant checking of LSTS in the context of CRA. For example, in the tests of a token ring system with 6 nodes, the largest subsystem generated by CRA based on ROE contains only 1793 states and 9177 transitions while the largest subsystem generated by the CRA based on CFFD contains 266805 states and 1581601 transitions.

The rest of the paper is organized as follows. After Section 2 gives some concepts of LTS and LSTS, Section 3 presents the reverse observation equivalence. Then, Section 4 formalizes the CRA based on reverse observation equivalence and Section 5 makes a case study. The conclusion is summarized finally.

2 Labelled State Transition Systems

Labelled transition system is a fundamental formal model that can be used to model the behavior of many systems. It's widely used in the literature. Many formal models use LTS as their semantic basis. Notable examples are the many variants of automata, and process algebras such as CCS [11], CSP [12].

Definition 1 (Labelled Transition System). *A labelled transition system(LTS) is a quadruple*

$$P =< V_P, A_P, \Delta_P, q_P >$$

where

- V_P *is a set of* states*;*
- $A_P = \alpha P \cup \{\tau\}$*, where* αP *denotes the communicating* alphabet *of* P *which does not contain the internal action* τ*;*
- $\Delta_P \subseteq V_P \times A_P \times V_P$*, denotes a* transition relation *that maps from a state and an action onto another state;*
- q_P *is a state in* V_P *which indicates the* initial state *of* P*.*

For an LTS P, two states $s_1, s_2 \in V_P$, an action $a \in A_P$, and two finite action sequences $\alpha = a_1 a_2 \cdots a_n \in (A_P)^n$ and $\beta = b_1 b_2 \cdots b_n \in (\alpha P)^n$, we define operator $\hat{a}$ and several relations as follows.

- $\hat{a} = \varepsilon$ if $a = \tau$, and $\hat{a} = a$ otherwise, where"ε" denotes the empty action sequence. $\hat{\alpha} = \hat{a_1}\hat{a_2} \cdots \hat{a_n}$.
- $s_1 \xrightarrow{a}_P s_2$ iff $(s_1, a, s_2) \in \Delta_P$.
- $s_1 \xrightarrow{\alpha}_P s_2$ iff $s_1 \xrightarrow{a_1}_P \xrightarrow{a_2}_P \cdots \xrightarrow{a_n}_P s_2$. Especially, $s_1 \xrightarrow{\varepsilon}_P s_1$.
- $s_1 \stackrel{\varepsilon}{\Longrightarrow}_P s_2$ iff $s_1 (\xrightarrow{\tau}_P)^* s_2$.
- $s_1 \stackrel{a}{\Longrightarrow}_P s_2$ iff $s_1 \stackrel{\varepsilon}{\Longrightarrow}_P \xrightarrow{a}_P \stackrel{\varepsilon}{\Longrightarrow}_P s_2$.
- $s_1 \stackrel{\beta}{\Longrightarrow}_P s_2$ iff $s_1 \stackrel{b_1}{\Longrightarrow}_P \stackrel{b_2}{\Longrightarrow}_P \cdots \stackrel{b_n}{\Longrightarrow}_P s_2$.

where $*$ is reflexive and transitive closure and juxtaposition is a composition of relations.

The parallel composition of two LTSs is defined [5] similar to that used in CSP.

Definition 2 (Parallel Composition of LTSs). *The* parallel composition $P \parallel Q$ *of two LTSs* P *and* Q *is defined as LTS* $R =< V_R, A_R, \Delta_R, q_R >$ *where*

- $V_R = V_P \times V_Q$, $A_R = A_P \cup A_Q$, *and* $q_R = (q_P, q_Q)$.
- Δ_R *is given by the following three transition rules:*

$$\frac{s_P \xrightarrow{a}_P s'_P}{(s_P, s_Q) \xrightarrow{a}_R (s'_P, s_Q)} (a \notin \alpha Q) \qquad \frac{s_Q \xrightarrow{a}_Q s'_Q}{(s_P, s_Q) \xrightarrow{a}_R (s_P, s'_Q)} (a \notin \alpha P)$$

$$\frac{s_P \xrightarrow{a}_P s'_P \quad s_Q \xrightarrow{a}_Q s'_Q}{(s_P, s_Q) \xrightarrow{a}_R (s'_P, s'_Q)} (a \in \alpha P \cap \alpha Q)$$

External actions of an LTS can be hidden and become unobservable.

Definition 3 (Hiding of LTS). *The* hiding *of LTS* P *on action set* L *(*$\tau \notin L$*) is defined as LTS* $P \backslash L =< V_P, A_P \backslash L, \Delta_{P\backslash L}, q_P >$ *where* $\Delta_{P\backslash L}$ *is given by the following two transition rules:*

$$\frac{s \xrightarrow{a}_P s'}{s \xrightarrow{\tau}_{P\backslash L} s'} (a \in L) \qquad \frac{s \xrightarrow{a}_P s'}{s \xrightarrow{a}_{P\backslash L} s'} (a \notin L)$$

Labelled state transition system is introduce in [1] as a formalism which augments LTS with atomic state propositions and has the features of both state-based and action-based models. In this paper, we make a slight extension to it: several possible evaluations are allowed in a state.

Definition 4 (Labelled State Transition System). *A* labelled state transition system*(LSTS) is a tuple*

$$P =< V_P, A_P, \Delta_P, q_P; \Pi_P, \Upsilon_P, val_P >$$

where LTS $P^{LTS} =< V_P, A_P, \Delta_P, q_P >$ *is augmented with the set of propositions* Π_P, evaluation function $val_P : V_P \rightarrow 2^{2^{\Pi_P}}$, *and the set* Υ_P *of* permanent *propositions for which* $\Upsilon_P \subseteq \Pi_P$.

The evaluation function tells which interpretations of the propositions are possible in a state of the system. In the original definition of LSTS, val_P is a function of type $V_P \rightarrow 2^{\Pi_P}$ rather than $V_P \rightarrow 2^{2^{\Pi_P}}$. We make this change to benefit the merging of states when reducing models according to the equivalence semantics. The permanent propositions are used by the generalized parallel composition operator of LSTS (see below).

Figure 1 presents the LSTS of a client in a token-ring system that is introduced in [1]. Some adaption has been made. We will present in more detail in Section 5.

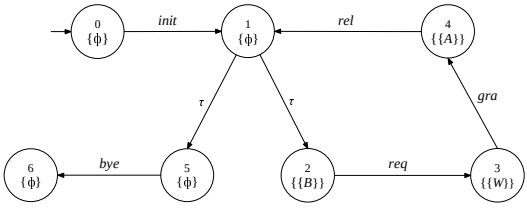


Fig. 1. LSTS of the clients

A remarkable characteristic of LSTS is that propositions do not affect synchronization. That is, when composed, several LSTSs communicate only by the actions, which is as same as LTSs. Concretely, the *generalized parallel composition* of LSTS is defined in [1], which is more capable than the parallel composition of LTS. It can do exactly the same transformations as the traditional parallel composition, hiding, and multiple renaming combined, when applied to LTSs [1]. In this paper, for the sake of simplicity, we introduce *basic parallel composition.*

Definition 5 (Basic Parallel Composition of LSTSs). *Given two LSTSs* P *and* Q, *their* basic parallel composition $P \parallel Q$ *is defined as the LSTS* $R =< V_R, A_R, \Delta_R, q_R; \Pi_R, \Upsilon_R, val_R >$ *where*

- $R^{LTS} =< V_R, A_R, \Delta_R, q_R >= P^{LTS} \parallel Q^{LTS}$.[1]
- $\Pi_R = \Pi_P \cup \Pi_Q$, $\Upsilon_R = \Upsilon_P \cup \Upsilon_Q$.
- $val_R((s_P, s_Q)) = \{\Omega_P \cup \Omega_Q \mid \Omega_P \in val_P(s_P) \wedge \Omega_Q \in val_Q(s_Q)\}$.

[1] "$\parallel$" is the parallel composition operator of LTS.

It can be seen that the basic parallel composition is consistent with the generalized parallel composition, which integrates the power of the basic parallel composition, hiding, multiple renaming and etc. In the following, the basic parallel composition will be called *parallel composition* shortly.

Similarly, we can define the hiding operator of LSTS.

Definition 6 (Hiding of LSTS). *The* hiding *of LSTS P on action set L ($\tau \notin L$) is defined as LSTS $P\backslash L =< V_P, A_P\backslash L, \Delta_{P\backslash L}, q_P; \Pi_P, \Upsilon_P, val_P >$ where $< V_P, A_P\backslash L, \Delta_{P\backslash L}, q_P >= P^{LTS}\backslash L$.*[2]

The multiple renaming operator of LSTS can be defined similar to that of LTS [13]. Given an LTS or LSTS P, we denote the set of all the reachable states of P as V_P^{Rch}.

Definition 7 (Invariant). *Consider an LSTS P. Let ϕ be a boolean expression over the set Π_P. Then ϕ is an* invariant *of P if for all states $s \in V_P^{Rch}$ and all evaluations $\Omega \in val_P(s)$, ϕ is satisfied under the interpretation I_Ω defined as follows.*

$$\forall q \in \Pi_P.\ I_\Omega(q) = \begin{cases} True & if\ \ q \in \Omega \\ False & otherwise \end{cases}$$

3 Reverse Observation Equivalence

Observation equivalence [14] is a concept to identify a pair of processes or LTSs which cannot be distinguished by an observer who is not able to observe internal τ-actions. It can be used in compositional reachability analysis to reduce the size of intermediate models. For our purpose of checking invariants of LSTS, it might be inappropriate to use a direct extension of observation equivalence of LTS if not extending the notion of actions, because it could not be a congruence again with respect to the parallel composition of LSTS while preserving invariants.

We find that the requirement can be satisfied if we define an equivalence relation in the reverse direction to observation equivalence. We call the new equivalence relation "reverse observation equivalence".

As observation equivalence is based on weak bisimulations, firstly we introduce the notion of *reverse weak bisimulation* between LSTSs. In the following, we assume that all LSTSs have the property "root-unwound" [14], that is, the initial state is not the destination of any transitions. For those LSTSs where the property does not hold, a simple adaptation can be made: add a state as the new initial state and a transition from the new state to the old initial state. Obviously, the adaptation does not affect the invariant checking of LSTS.

Firstly, we define several shorthands. Given a binary relation $R \subseteq M \times N$, $m \in M$ and $n \in N$, we denote the set of all the images of R as $R[*]$, the set of all the inverseimages of R as $R^{-1}[*]$, the set of all the images of m in relation R

[2] "\" is the hiding operator of LTS.

as $R[m]$, and the set of all the inverseimages of n in relation R as $R^{-1}[n]$. Their formal definitions are as follows.

$$\begin{aligned} R[*] &=_{df} \{s \in N \mid \exists s' \in M.\ (s', s) \in R\} \\ R^{-1}[*] &=_{df} \{s \in M \mid \exists s' \in N.\ (s, s') \in R\} \\ R[m] &=_{df} \{s \in N \mid (m, s) \in R\} \\ R^{-1}[n] &=_{df} \{s \in M \mid (s, n) \in R\} \end{aligned}$$

If saying that weak bisimulation is a relation that "looks forward", then reverse weak bisimulation is a relation that "looks backward".

Definition 8 (Reverse Weak Bisimulation between LSTSs). *Given two LSTSs P and Q, let $R \subseteq V_P^{Rch} \times V_Q^{Rch}$ be a binary relation. If $\alpha P = \alpha Q$, $\Pi_P = \Pi_Q$, $\Upsilon_P = \Upsilon_Q$, $(q_{_P}, q_{_Q}) \in R$, $R[*] = V_Q^{Rch}$, $R^{-1}[*] = V_P^{Rch}$, and for all $(s_{_P}, s_{_Q}) \in R$ the following three conditions are satisfied, then R is a* reverse weak bisimulation *between P and Q.*

1. *If $s'_{_P} \xrightarrow{a}_P s_{_P}$ and $s'_{_P} \in V_P^{Rch}$, then $s'_{_Q} \stackrel{\hat{a}}{\Longrightarrow}_Q s_{_Q}$ for some $s'_{_Q} \in V_Q^{Rch}$ such that $(s'_{_P}, s'_{_Q}) \in R$.*
2. *If $s'_{_Q} \xrightarrow{a}_Q s_{_Q}$ and $s'_{_Q} \in V_Q^{Rch}$, then $s'_{_P} \stackrel{\hat{a}}{\Longrightarrow}_P s_{_P}$ for some $s'_{_P} \in V_P^{Rch}$ such that $(s'_{_P}, s'_{_Q}) \in R$.*
3. *$val_{_P}(s_{_P}) \subseteq \bigcup_{s'_{_Q} \in R[s_{_P}]} val_{_Q}(s'_{_Q})$ and $val_{_Q}(s_{_Q}) \subseteq \bigcup_{s'_{_P} \in R^{-1}[s_{_Q}]} val_{_P}(s'_{_P})$.*

LSTSs P and Q are *reverse observation equivalent*, if there is a reverse weak bisimulation between them. When this is the case, we write $P \approx_r Q$. It is easy to prove that $\approx_r$ is an equivalence relation, which we called *reverse observation equivalence* (shortly ROE). Furthermore, states $s_{_P} \in V_P$ and $s_{_Q} \in V_Q$ are *reverse weak bisimilar*, or *reverse observation equivalent*, if there is a reverse weak bisimulation R between P and Q such that $(s_{_P}, s_{_Q}) \in R$.

In the definition of reverse weak bisimulation, we have borrowed the idea of backward simulation [15] of I/O automata. However, it is worth noting that ROE is not a trivial extension of backward simulation in the context of LSTS. To preserve invariant properties, a special form of constraint on the atomic state propositions is added to the definition.

From the definition above, it can be seen that two reverse observation equivalent states are undistinguishable in how the states can be reached from the

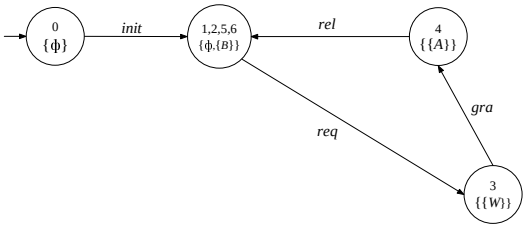


Fig. 2. LSTS of the clients after ROE reduction

initial states as to observers that can not see the internal actions of the system. So in a sense, reverse observation equivalence focuses on the "history", while observation equivalence focuses on the "future". It is worth noting that in the definition we require that $R[*] = V_Q^{Rch}$ and $R^{-1}[*] = V_P^{Rch}$. That is, only reachable states can be simulated and all the reachable states must be simulated. Figure 2 shows the ROE-reduced LSTS of the client in Figure 1 after hiding the action "bye" firstly. Because the states 1, 2, 5, and 6 of the original LSTS are reverse observation equivalent after the hiding, they are merged to a single state in the ROE-reduced LSTS.

The following theorem illustrates the intuition that reverse weak bisimulation preserves invariants of LSTS. It can be proved straightforwardly from the definition of reverse observation equivalence.

Theorem 1. *Given two reverse observation equivalent LSTSs P and Q, let $\Omega \in 2^{\Pi_P}$ (or $\Omega \in 2^{\Pi_Q}$), and ϕ be a boolean expression over Π_P (or Π_Q). Then the following two propositions hold:*

1. $(\exists s_P \in V_P^{Rch}.\ \Omega \in val_P(s_P)) \Leftrightarrow (\exists s_Q \in V_Q^{Rch}.\ \Omega \in val_Q(s_Q))$.
2. *ϕ is an invariant of P iff ϕ is an invariant of Q.*

The next lemma describes the essential property of $\approx_r$.

Lemma 1. *Consider two LSTSs P and Q that are reverse observation equivalent. Let $\approx$ be a reverse weak bisimulation between them. Let states $s_P \in V_P^{Rch}$ and $s_Q \in V_Q^{Rch}$ which satisfy $s_P \approx s_Q$ (that is, $(s_P, s_Q) \in \approx$). Let action sequence $\alpha = a_1 a_2 \dots a_n \in (\alpha P)^*$, $\beta = b_1 b_2 \dots b_n \in (\alpha Q)^*$. Then the following propositions hold:*

1. *If $s'_P \stackrel{\alpha}{\Longrightarrow}_P s_P$ and $s'_P \in V_P^{Rch}$, then $s'_Q \stackrel{\alpha}{\Longrightarrow}_Q s_Q$ for some $s'_Q \in V_Q^{Rch}$ such that $s'_P \approx s'_Q$.*
2. *If $s'_Q \stackrel{\beta}{\Longrightarrow}_Q s_Q$ and $s'_Q \in V_Q^{Rch}$, then $s'_P \stackrel{\beta}{\Longrightarrow}_P s_P$ for some $s'_P \in V_P^{Rch}$ such that $s'_P \approx s'_Q$.*

Proof. We only prove the proposition 1, since the other can be proved similarly. Suppose $s'_P \stackrel{\alpha}{\Longrightarrow}_P s_P$ and $s'_P \in V_P^{Rch}$. Then by the definition of "$\Longrightarrow$", there must exist an action sequence $\gamma = r_1 r_2 \dots r_m \in (A_P)^*$ and a group of states $s_0, s_1, \dots s_m \in V_P$ such that $\hat{\gamma} = \alpha$, $s_0 = s'_P$, $s_m = s_P$ and $s_i \stackrel{r_{i+1}}{\longrightarrow}_P s_{i+1}$ $(\forall i \in [0 \dots m-1])$. For $s'_P \in V_P^{Rch}$, we have $s_i \in V_P^{Rch}$ $(\forall i \in [0 \dots m])$. Since $s_{m-1} \stackrel{r_m}{\longrightarrow}_P s_m$ and $s_m \approx s_Q$ (that is, $s_P \approx s_Q$), by the definition of $\approx$ we know there exists a state $s'_{m-1} \in V_Q^{Rch}$ such that $s'_{m-1} \stackrel{\hat{r}_m}{\Longrightarrow}_Q s_Q$ and $s_{m-1} \approx s'_{m-1}$. Similarly, by $s_{m-2} \stackrel{r_{m-1}}{\longrightarrow}_P s_{m-1}$ and $s_{m-1} \approx s'_{m-1}$, we know that there exists a state $s'_{m-2} \in V_Q^{Rch}$ such that $s'_{m-2} \stackrel{\hat{r}_{m-1}}{\Longrightarrow}_Q s_{m-1}$ and $s_{m-2} \approx s'_{m-2}$. Deducing recursively, it can be known that there exists a group of states $s'_0, s'_1, \dots s'_m \in V_Q^{Rch}$ such that $s'_m = s_Q$ and $s_i \approx s'_i \wedge (s'_i \stackrel{\hat{r}_{i+1}}{\Longrightarrow}_Q s'_{i+1})$ $(\forall i \in [0 \dots m-1])$. Therefore, we have $s'_0 \stackrel{\hat{\gamma}}{\Longrightarrow}_Q s_Q$ and $s_0 \approx s'_0$. Let $s'_Q = s'_0$. Then we have $s'_Q \stackrel{\alpha}{\Longrightarrow}_Q s_Q$ and $s'_P \approx s'_Q$. $\square$

Since all the LSTSs have the property "root-unwound", those states that are reverse weak bisimilar with the initial state have a special property, as shown in the following lemma.

Lemma 2. *Given two reverse observation equivalent LSTSs P and Q, two states $s_P \in V_P^{Rch}$ and $s_Q \in V_Q^{Rch}$, let $\approx$ be a reverse weak bisimulation between them. Then the following two propositions hold:*

1. *If $s_P \approx q_Q$, then $q_P \stackrel{\varepsilon}{\Longrightarrow}_P s_P$.*
2. *If $q_P \approx s_Q$, then $q_Q \stackrel{\varepsilon}{\Longrightarrow}_Q s_Q$.*

Proof. We only prove proposition 1. Because $s_P \in V_P^{Rch}$, there exists an action sequence $\alpha = a_1 a_2 \ldots a_n \in (\alpha P)^*$ such that $q_P \stackrel{\alpha}{\Longrightarrow}_P s_P$. Suppose $s_P \approx q_Q$. Then by Lemma 1 there must exist a state $s'_Q \in V_Q^{Rch}$ such that $s'_Q \stackrel{\alpha}{\Longrightarrow}_Q q_Q$ and $q_P \approx s'_Q$. By the assumption about "root-unwound" property, we know $s'_Q = q_Q$ and $\alpha = \varepsilon$. Thus $q_P \stackrel{\varepsilon}{\Longrightarrow}_P s_P$. □

As to parallel composition and hiding, $\approx_r$ is a congruence. The following theorem makes it clear.

Theorem 2. *Given LSTSs P, P', Q, Q' and an action set L that does not contain the internal action τ, the following propositions hold:*

1. *$P \approx_r P' \Rightarrow P \setminus L \approx_r P' \setminus L$.*
2. *$P \approx_r P' \wedge Q \approx_r Q' \Rightarrow P \parallel Q \approx_r P' \parallel Q'$.*

Proof. Obviously, the proposition 1 is true. We only prove the proposition 2. Suppose $P \approx_r P' \wedge Q \approx_r Q'$. According to the definition of $\approx_r$, there exist reverse weak bisimulations $\approx_P$ between P and P', and $\approx_Q$ between Q and Q'. We construct a binary relation $\approx_{P\|Q}$ between $P \parallel Q$ and $P' \parallel Q'$ as the following set:

$$\{((s_P, s_Q), (s_{P'}, s_{Q'})) \in V_{P\|Q}^{Rch} \times V_{P'\|Q'} | (s_P, s_{P'}) \in \approx_P \wedge (s_Q, s_{Q'}) \in \approx_Q\}. \quad (1)$$

It is sufficient to prove that $\approx_{P\|Q}$ is a reverse weak bisimulation between $P \parallel Q$ and $P' \parallel Q'$. We prove it according to the definition of reverse weak bisimulation.

1. Firstly, we prove $\approx_{P\|Q} \subseteq V_{P\|Q}^{Rch} \times V_{P'\|Q'}^{Rch}$. Given any pair of states $((s_P, s_Q), (s_{P'}, s_{Q'})) \in \approx_{P\|Q}$, we will prove $((s_P, s_Q), (s_{P'}, s_{Q'})) \in V_{P\|Q}^{Rch} \times V_{P'\|Q'}^{Rch}$. By the definition of $\approx_{P\|Q}$, we have $((s_P, s_Q), (s_{P'}, s_{Q'})) \in V_{P\|Q}^{Rch} \times V_{P'\|Q'}$, $(s_P, s_{P'}) \in \approx_P$ and $(s_Q, s_{Q'}) \in \approx_Q$. So it is sufficient to prove that $(s_{P'}, s_{Q'}) \in V_{P'\|Q'}^{Rch}$. Because $(s_P, s_Q) \in V_{P\|Q}^{Rch}$, there exists an action sequence $\alpha = a_1 a_2 \ldots a_n \in (\alpha P \cup \alpha Q)^*$ such that $(q_P, q_Q) \stackrel{\alpha}{\Longrightarrow}_{P\|Q} (s_P, s_Q)$. Thus $q_P \stackrel{\alpha|_{\alpha P}}{\Longrightarrow}_P s_P$ [3] and $q_Q \stackrel{\alpha|_{\alpha Q}}{\Longrightarrow}_Q s_Q$. Because $(s_P, s_{P'}) \in \approx_P$ and $(s_Q, s_{Q'}) \in \approx_Q$, by Lemma 1 we know there exist states $s'_{P'} \in V_{P'}^{Rch}$, $s'_{Q'} \in V_{Q'}^{Rch}$ such that

[3] $\alpha|_{\alpha P}$ denotes the projection of α on αP.

$s'_{P'} \overset{\alpha|_{\alpha P}}{\Longrightarrow}_{P'} s_{P'}$, $q_P \approx_P s'_{P'}$ and $s'_{Q'} \overset{\alpha|_{\alpha Q}}{\Longrightarrow}_{Q'} s_{Q'}$, $q_Q \approx_Q s'_{Q'}$. Because $q_P \approx_P s'_{P'}$ and $q_Q \approx_Q s'_{Q'}$, by Lemma 2 we know $q_{P'} \overset{\varepsilon}{\Longrightarrow}_{P'} s'_{P'}$ and $q_{Q'} \overset{\varepsilon}{\Longrightarrow}_{Q'} s'_{Q'}$. Combining with the previous result, it can be known that $q_{P'} \overset{\alpha|_{\alpha P}}{\Longrightarrow}_{P'} s_{P'}$ and $q_{Q'} \overset{\alpha|_{\alpha Q}}{\Longrightarrow}_{Q'} s_{Q'}$. By the assumption that $P \approx_r P' \wedge Q \approx_r Q'$, it can be known that $\alpha P = \alpha P'$ and $\alpha Q = \alpha Q'$. Thus $q_{P'} \overset{\alpha|_{\alpha P'}}{\Longrightarrow}_{P'} s_{P'}$ and $q_{Q'} \overset{\alpha|_{\alpha Q'}}{\Longrightarrow}_{Q'} s_{Q'}$. By the property of parallel composition of LTSs, we know $(q_{P'}, q_{Q'}) \overset{\alpha}{\Longrightarrow}_{P'\|Q'} (s_{P'}, s_{Q'})$. Therefore, $(s_{P'}, s_{Q'}) \in V^{Rch}_{P'\|Q'}$. Thus $\approx_{P\|Q} \subseteq V^{Rch}_{P\|Q} \times V^{Rch}_{P'\|Q'}$. From the result, we know that $\approx_{P\|Q}$ can be defined equivalently as the following set:

$$\{((s_P, s_Q), (s_{P'}, s_{Q'})) \in V^{Rch}_{P\|Q} \times V^{Rch}_{P'\|Q'} | (s_P, s_{P'}) \in \approx_P \wedge (s_Q, s_{Q'}) \in \approx_Q\}. \quad (2)$$

Furthermore, by symmetry we know that $\approx_{P\|Q}$ can also be defined equivalently as the following set:

$$\{((s_P, s_Q), (s_{P'}, s_{Q'})) \in V_{P\|Q} \times V^{Rch}_{P'\|Q'} | (s_P, s_{P'}) \in \approx_P \wedge (s_Q, s_{Q'}) \in \approx_Q\}. \quad (3)$$

2. By $P \approx_r P' \wedge Q \approx_r Q'$, it can be known that $\alpha P = \alpha P'$, $\alpha Q = \alpha Q'$, $\Pi_P = \Pi_{P'}$, $\Pi_Q = \Pi_{Q'}$, $\Upsilon_P = \Upsilon_{P'}$, and $\Upsilon_Q = \Upsilon_{Q'}$. Thus $\alpha P \cup \alpha Q = \alpha P' \cup \alpha Q'$, that is, $\alpha(P \parallel Q) = \alpha(P' \parallel Q')$. Similarly, $\Pi_{P\|Q} = \Pi_{P'\|Q'}$ and $\Upsilon_{P\|Q} = \Upsilon_{P'\|Q'}$. By the algorithm of constructing $\approx_{P\|Q}$, it can be known straightforwardly that $((q_P, q_Q), (q_{P'}, q_{Q'})) \in \approx_{P\|Q}$. In the following proof, we use R as an alias of $\approx_{P\|Q}$. Next we will prove $R^{-1}[*] = V^{Rch}_{P\|Q}$ and $R[*] = V^{Rch}_{P'\|Q'}$. Since in the above we have proved that $\approx_{P\|Q} \subseteq V^{Rch}_{P\|Q} \times V^{Rch}_{P'\|Q'}$, thus $R^{-1}[*] \subseteq V^{Rch}_{P\|Q}$ and $R[*] \subseteq V^{Rch}_{P'\|Q'}$. So it is sufficient to prove that $R^{-1}[*] \supseteq V^{Rch}_{P\|Q}$ and $R[*] \supseteq V^{Rch}_{P'\|Q'}$. Given any state pair $(s_P, s_Q) \in V^{Rch}_{P\|Q}$, we have $s_P \in V^{Rch}_P$ and $s_Q \in V^{Rch}_Q$. By the definition of reverse weak bisimulation, we know $\approx_P^{-1}[*] = V^{Rch}_P$ and $\approx_Q^{-1}[*] = V^{Rch}_Q$. Thus there exist states $s_{P'} \in V^{Rch}_{P'}$ and $s_{Q'} \in V^{Rch}_{Q'}$ such that $s_P \approx_P s_{P'}$ and $s_Q \approx_Q s_{Q'}$. By the equation (1), we have $((s_P, s_Q), (s_{P'}, s_{Q'})) \in \approx_{P\|Q}$. So $(s_P, s_Q) \in R^{-1}[*]$. Therefore, $R^{-1}[*] \supseteq V^{Rch}_{P\|Q}$. On the other hand, because $\approx_{P\|Q}$ can also be defined equivalently in the form of equation (3), it can be proved symmetrically that $R[*] \supseteq V^{Rch}_{P'\|Q'}$.

3. Next, we will prove that for all $((s_P, s_Q), (s_{P'}, s_{Q'})) \in \approx_{P\|Q}$, the following three propositions hold:
   (a) If $(s'_P, s'_Q) \xrightarrow{a}_{P\|Q} (s_P, s_Q)$ and $(s'_P, s'_Q) \in V^{Rch}_{P\|Q}$, then there exists a state $(s'_{P'}, s'_{Q'}) \in V^{Rch}_{P'\|Q'}$ such that $(s'_{P'}, s'_{Q'}) \overset{\hat{a}}{\Longrightarrow}_{P'\|Q'} (s_{P'}, s_{Q'})$ and $((s'_P, s'_Q), (s'_{P'}, s'_{Q'})) \in \approx_{P\|Q}$.
   (b) If $(s'_{P'}, s'_{Q'}) \xrightarrow{a}_{P'\|Q'} (s_{P'}, s_{Q'})$ and $(s'_{P'}, s'_{Q'}) \in V^{Rch}_{P'\|Q'}$, then there exists a state $(s'_P, s'_Q) \in V^{Rch}_{P\|Q}$ such that $(s'_P, s'_Q) \overset{\hat{a}}{\Longrightarrow}_{P\|Q} (s_P, s_Q)$ and $((s'_P, s'_Q), (s'_{P'}, s'_{Q'})) \in \approx_{P\|Q}$.

(c) $val_{P\|Q}((s_P, s_Q)) \subseteq \bigcup_{t' \in R[(s_P, s_Q)]} val_{P'\|Q'}(t')$ and $val_{P'\|Q'}((s_{P'}, s_{Q'})) \subseteq \bigcup_{t \in R^{-1}[(s_{P'}, s_{Q'})]} val_{P\|Q}(t)$.

By the symmetry[4], it is sufficient to prove only (a) and (c). Suppose $(s'_P, s'_Q) \xrightarrow{a}_{P\|Q} (s_P, s_Q)$ and $(s'_P, s'_Q) \in V^{Rch}_{P\|Q}$. Then we have $s'_P \overset{\hat{a}|_{\alpha P}}{\Longrightarrow}_P s_P$, $s'_Q \overset{\hat{a}|_{\alpha Q}}{\Longrightarrow}_Q s_Q$, $s'_P \in V^{Rch}_P$ and $s'_Q \in V^{Rch}_Q$. Because $((s_P, s_Q), (s_{P'}, s_{Q'})) \in \approx_{P\|Q}$, we have $(s_P, s_Q) \in V^{Rch}_{P\|Q}$, $(s_{P'}, s_{Q'}) \in V^{Rch}_{P'\|Q'}$, $s_P \approx_P s_{P'}$ and $s_Q \approx_Q s_{Q'}$. By Lemma 1, there exist states $s'_{P'} \in V^{Rch}_{P'}$ and $s'_{Q'} \in V^{Rch}_{Q'}$ such that $s'_{P'} \overset{\hat{a}|_{\alpha P}}{\Longrightarrow}_{P'} s_{P'}$, $s'_P \approx_P s'_{P'}$, $s'_{Q'} \overset{\hat{a}|_{\alpha Q}}{\Longrightarrow}_{Q'} s_{Q'}$ and $s'_Q \approx_Q s'_{Q'}$. Therefore, $s'_{P'} \overset{\hat{a}|_{\alpha P'}}{\Longrightarrow}_{P'} s_{P'}$ and $s'_{Q'} \overset{\hat{a}|_{\alpha Q'}}{\Longrightarrow}_{Q'} s_{Q'}$. By the property of parallel composition of LTS, it can be known that $(s'_{P''}, s'_{Q''}) \overset{\hat{a}}{\Longrightarrow}_{P'\|Q'} (s_{P'}, s_{Q'})$. Because $(s'_P, s'_Q) \in V^{Rch}_{P\|Q}$, $s'_P \approx_P s'_{P'}$ and $s'_Q \approx_Q s'_{Q''}$ by the algorithm of constructing $\approx_{P\|Q}$ it can be known that $((s'_P, s'_Q), (s'_{P''}, s'_{Q''})) \in \approx_{P\|Q}$. Since it has been proved that $\approx_{P\|Q} \subseteq V^{Rch}_{P\|Q} \times V^{Rch}_{P'\|Q'}$, we know $(s'_{P''}, s'_{Q''}) \in V^{Rch}_{P'\|Q'}$. Thus the proposition (a) holds. Next, we will prove the proposition (c). By the definition of $\approx_P$ and $\approx_Q$, we have:

$$(1)\ val_P(s_P) \subseteq \bigcup_{t_{P'} \in (\approx_P [s_P])} val_{P'}(t_{P'})$$
$$(2)\ val_Q(s_Q) \subseteq \bigcup_{t_{Q'} \in (\approx_Q [s_Q])} val_{Q'}(t_{Q'}).$$

By (1), (2) and the definition of $P \parallel Q$, we have:

$$\begin{aligned}(3)\ val_{P\|Q}((s_P, s_Q)) &= \{\Omega_P \cup \Omega_Q \mid \Omega_P \in val_P(s_P) \wedge \Omega_Q \in val_Q(s_Q)\} \\ &\subseteq \{\Omega_P \cup \Omega_Q \mid \Omega_P \in \bigcup_{t_{P'} \in (\approx_P [s_P])} val_{P'}(t_{P'}) \wedge \\ &\qquad \Omega_Q \in \bigcup_{t_{Q'} \in (\approx_Q [s_Q])} val_{Q'}(t_{Q'}).\end{aligned}$$

By the definitions of R and $P' \parallel Q'$, we have:

$$(4)\quad R[(s_P, s_Q)] = (\approx_P [s_P]) \times (\approx_Q [s_Q])$$
$$(5)\ val_{P'\|Q'}((t_{P'}, t_{Q'})) = \{\Omega_{P'} \cup \Omega_{Q'} \mid \Omega_{P'} \in val_{P'}(t_{P'}) \wedge \Omega_{Q'} \in val_{Q'}(t_{Q'})\}.$$

where $(t_{P'}, t_{Q'}) \in V_{P'\|Q'}$. Let $t' = (t_{P'}, t_{Q'})$. By (4) and (5), we know:

$$\begin{aligned}(6)\ \bigcup_{t' \in R[(s_P, s_Q)]} val_{P'\|Q'}(t') &= \{\Omega_{P'} \cup \Omega_{Q'} \mid \exists t_{P'} \in (\approx_P [s_P]).\ t_{Q'} \in (\approx_Q [s_Q]). \\ &\qquad (\Omega_{P'} \in val_{P'}(t_{P'}) \wedge \Omega_{Q'} \in val_{Q'}(t_{Q'})\}) \\ &= \{\Omega_{P'} \cup \Omega_{Q'} \mid \Omega_{P'} \in \bigcup_{t_{P'} \in (\approx_P [s_P])} val_{P'}(t_{P'}) \wedge \\ &\qquad \Omega_{Q'} \in \bigcup_{t_{Q'} \in (\approx_Q [s_Q])} val_{Q'}(t_{Q'})\}.\end{aligned}$$

Furthermore, by (3) and (6) it can be concluded that:

$$val_{P\|Q}((s_P, s_Q)) \subseteq \bigcup_{t' \in R[(s_P, s_Q)]} val_{P'\|Q'}(t').$$

[4] Noting that $\approx_{P\|Q}$ can be defined equivalently in the forms of the equation (1) or (3), which are symmetrical.

Symmetrically, it can be proved that:

$$val_{P'\|Q'}((s_{P'}, s_{Q'})) \subseteq \bigcup_{t \in R^{-1}[(s_{P'}, s_{Q'})]} val_{P\|Q}(t).$$

Thus proposition (c) holds.

To sum up, the proposition 2 holds. □

As to reverse observation equivalence, several hiding operations can be merged. The following theorem illustrates it, which can be proved according to the definitions.

Theorem 3. *Given LSTSs P, Q and action sets L_1, L_2 that do not contain the internal action τ, the following propositions hold:*

1. $P \setminus L_1 \setminus L_2 \approx_r P \setminus (L_1 \cup L_2)$.
2. *If* $(\alpha P \cap L_1) = (\alpha P \cap L_2) = (L_1 \cap L_2) = \emptyset$, *then* $(P \setminus L_1) \parallel (Q \setminus L_2) \approx_r (P \parallel Q) \setminus (L_1 \cup L_2)$.

4 Compositional Reachability Analysis Based on Reverse Observation Equivalence

Compositional reachability analysis (CRA) is a kind of hierarchy-based incremental analysis approach [4, 5, 6, 7]. Given a hierarchy of models, it incrementally composes, hides and reduces subsets of the models according to the hierarchy, and finally produces a model that is equivalent to the one produced by the straightforward *all-at-once* approach, i.e. composing all the models simultaneously and then hiding all unobservable actions. Because the intermediate models can be reduced in CRA, the final model produced by CRA is usually much smaller than the one gotten by the "all-at-once" approach. The CRA approach is called *compositional LTS construction* in [10], and *incremental composition and reduction method* in [16]. In this section, we formalize the theory of CRA of LSTS based on reverse observation equivalence.

A hierarchy of LSTSs denotes a set of LSTSs with a hierarchy structure. Formally, we can define recursively a hierarchy H of LSTSs in BNF format:

$$H ::= \ (H_1 \parallel H_2 \parallel \cdots \parallel H_n) \setminus L \ \mid \ LSTS \setminus L$$

where $LSTS$ denotes an LSTS and L denotes an action set ($\tau \notin L$). Intuitively, a hierarchy is either the composition of a set of sub-hierarchies or simply an LSTS, with the actions in L hidden.

Given an LSTS P and a hierarchy H of LSTSs, let $red(P)$ denote a reduction operation to P based on reverse observation equivalence (that is, $red(P) \approx_r P$), and $cra(H)$ denote the final LSTS of CRA of H based on reverse observation equivalence. Then the algorithm of CRA can be represented as follows.

$$cra(H) = \begin{cases} red((\prod_{i=1}^{n} cra(H_i)) \setminus L) & \text{if } \ H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n) \setminus L \\ red(P \setminus L) & \text{if } \ H = P \setminus L \end{cases}$$

where $\prod_{i=1}^{n} cra(H_i)$ denotes $cra(H_1) \parallel cra(H_2) \parallel \cdots \parallel cra(H_n)$.

In order to illustrate the principle of CRA based on reverse observation equivalence, we define recursively several operators on hierarchies: $lsts(H)$, $act(H)$ and $hid(H)$, which denote respectively all LSTSs, all the actions and all the hidden actions of the hierarchy H. Their formal definitions are as follows.

$$lsts(H) = \begin{cases} \bigcup_{i=1}^{n} lsts(H_i) & \text{if } H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n)\backslash L \\ \{P\} & \text{if } H = P\backslash L \end{cases}$$

$$act(H) = \begin{cases} \bigcup_{i=1}^{n} act(H_i) & \text{if } H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n)\backslash L \\ \alpha P & \text{if } H = P\backslash L \end{cases}$$

$$hid(H) = \begin{cases} (\bigcup_{i=1}^{n} hid(H_i)) \cup L & \text{if } H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n)\backslash L \\ L & \text{if } H = P\backslash L \end{cases}$$

In the following discussion, we make an assumption that for all hierarchies $H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n)\backslash L$, the below formula holds.

$$\forall i, j \in [1 \ldots n].\ (i \neq j) \rightarrow ((hid(H_i) \cap act(H_j)) = (hid(H_i) \cap hid(H_j)) = \emptyset) \quad (4)$$

The aim of the assumption is to avoid the possible disturbance when deferring the hiding operations in CRA (see below). Obviously, when the assumption is not satisfied, renaming operations can be made to turn the assumption satisfied, without changing the behaviors of the LSTSs. So, the restriction does not cause any loss of generality.

Given a hierarchy H, let $lsts(H) = \{T_1, T_2, \ldots, T_n\}$. We denote the LSTS produced by the "all-at-once" approach as $once(H)$, that is:

$$once(H) =_{df} (\prod_{i=1}^{n} T_i)\backslash hid(H).$$

The next lemma shows that $once(H)$ can also be calculated compositionally.

Lemma 3. *Given a hierarchy of LSTSs* $H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n)\backslash L$, *then* $once(H) = (\prod_{i=1}^{n} once(H_i))\backslash L$.

Proof. Let $lsts(H_i) = \{T_i^1, T_i^2, \ldots, T_i^{n_i}\}$, and $\prod lsts(H_i) =_{df} \prod_{j=1}^{n_i} T_i^j$. Then we deduce as follows.

$$\begin{aligned} once(H) &= (\textstyle\prod_{i=1}^{n} \prod lsts(H_i))\backslash((\bigcup_{i=1}^{n} hid(H_i)) \cup L) \\ once(H_i) &= (\textstyle\prod lsts(H_i))\backslash hid(H_i) \\ (\textstyle\prod_{i=1}^{n} once(H_i))\backslash L &= (\textstyle\prod_{i=1}^{n} ((\prod lsts(H_i))\backslash hid(H_i)))\backslash L \\ (\textstyle\prod_{i=1}^{n} ((\prod lsts(H_i))\backslash hid(H_i)))\backslash L &= (\textstyle\prod_{i=1}^{n} \prod lsts(H_i))\backslash(\bigcup_{i=1}^{n} hid(H_i))\backslash L \\ &= (\textstyle\prod_{i=1}^{n} \prod lsts(H_i))\backslash((\bigcup_{i=1}^{n} hid(H_i)) \cup L) \end{aligned}$$

(by assumption (4) and Theorem 3)

$$once(H) = (\textstyle\prod_{i=1}^{n} once(H_i))\backslash L$$

□

The following theorem presents the essential principle of CRA based on reverse observation equivalence, i.e. the CRA approach and all-at-once approach coincide in the semantics of reverse observation equivalence.

Theorem 4. *For a hierarchy H of LSTSs, $cra(H) \approx_r once(H)$.*

Proof. We prove recursively according to the definition of hierarchy as follows.

1. If $H = P\backslash L$ where P is an LSTS, then $cra(H) = red(P\backslash L)$ and $once(H) = P\backslash L$. Because $red(P\backslash L) \approx_r P\backslash L$, $cra(H) \approx_r once(H)$ holds.
2. If $H = (H_1 \parallel H_2 \parallel \cdots \parallel H_n)\backslash L$ where H_i $(i \in [1 \ldots n])$ is the sub-hierarchy of H and satisfies $cra(H_i) \approx_r once(H_i)$, then we deduce as follows.

$$H' =_{df} (cra(H_1)||cra(H_2)||\cdots||cra(H_n))\backslash L$$

$$\begin{array}{lrcll}
 & cra(H) & = & red((\prod_{i=1}^{n} cra(H_i))\backslash L) & \text{by definition} \\
(1) & cra(H) & = & red(H') & \\
(2) & red(H') & \approx_r & H' & \\
 & cra(H) & \approx_r & H' & \text{by (1), (2)} \\
 & once(H) & = & (\prod_{i=1}^{n} once(H_i))\backslash L & \text{by Lemma 3} \\
(3) & cra(H_i) & \approx_r & once(H_i)(\mathrm{i} \in [1 \ldots \mathrm{n}]) & \text{by premise} \\
 & \prod_{i=1}^{n} cra(H_i) & \approx_r & \prod_{i=1}^{n} once(H_i) & \text{by Theorem 2 and (3)} \\
 & (\prod_{i=1}^{n} cra(H_i))\backslash L & \approx_r & (\prod_{i=1}^{n} once(H_i))\backslash L & \\
 & H' & \approx_r & once(H) & \\
 & cra(H) & \approx_r & once(H) &
\end{array}$$

To sum up, we have $cra(H) \approx_r once(H)$. □

5 Case Study

We have implemented a ROE-reduction algorithm in the framework of TVT toolkit. To show its actual effects, we give a simple example, a demand-driven token ring system for mutual exclusion which was introduced in [1]. We check the mutual exclusion property (an invariant property) of the system by three different approaches: CRA based on ROE, CRA based on CFFD, and the "all-at-once" approach. A comparison is made between them.

The system is depicted in Figure 3. It can be seen that there are n clients and n servers in the system. A client interacts with a server when intending to access the mutual exclusive resources. The servers are organized in a ring structure, in which one single token is passed clockwise (*tr* for sending and *tl* for receiving) and the demands for the token are passed counterclockwise (*dl* for sending and *dr* for receiving) [1]. The LSTS of clients have been presented in Figure 1 and the LSTS of servers can be found in [1]. It is worthy to mention that the LSTSs are adapted to satisfy the "root-unwound" property.

Our experiments show that the mutual exclusion property can be checked successfully whenever the models are correct or not by using the above three approaches. Figure 4 shows a hierarchy of the system with 3 clients and 3 servers, whose models are correct. Results are represented in the figure when adopting the approach of CRA based on ROE.

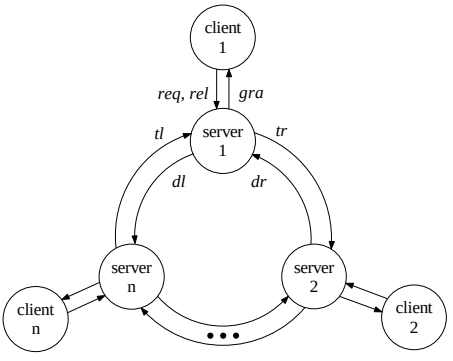


Fig. 3. Topology of a simplified token-ring system

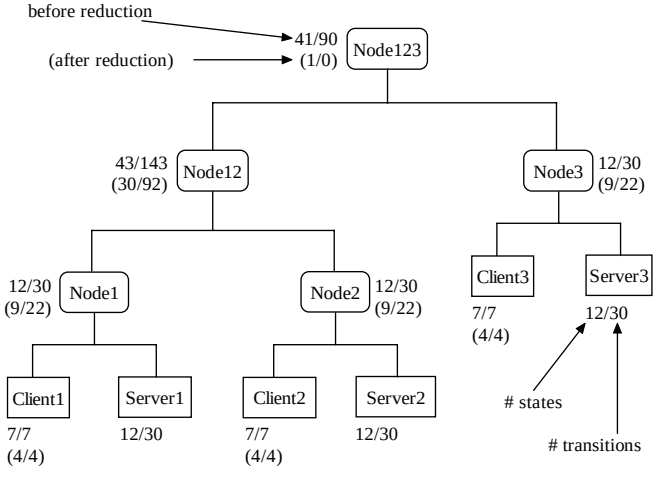


Fig. 4. A hierarchy of the token-ring system for CRA based on ROE

Comparing with the CRA based on ROE, a similar test is made by the approach of CRA based on CFFD (the original LSTSs are adopted without adaption for "root-unwound" property). The results are shown in Figure 5. It can be seen that the largest subsystem $Node12$ that is generated by CRA based on ROE contains 43 states and 143 transitions, while the largest subsystem $Node123$ that is generated by CRA based on CFFD contains 482 states and 1255 transitions. Thus, for invariant checking of LSTS, the former approach can produce much smaller models than the latter one.

Table 1 shows the sizes of the largest subsystems generated by CRA based on hierarchies similar to Figure 4 when the system contains $2 \sim 6$ nodes (one node contains a client and a server). From the table, it can be seen that ROE is much more efficient than CFFD when checking invariants of LSTS.

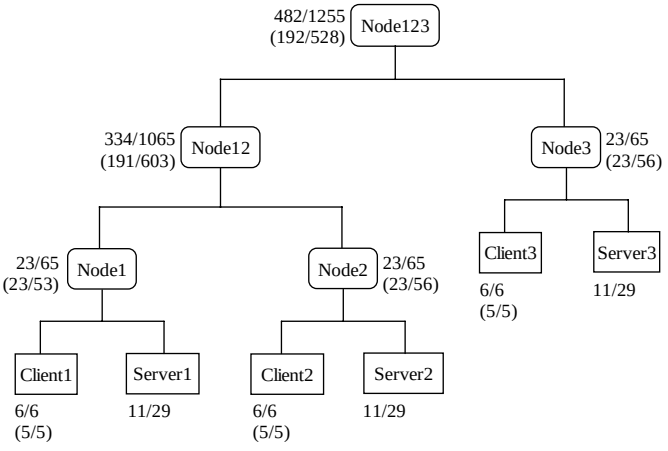


Fig. 5. A hierarchy of the token-ring system for CRA based on CFFD

Table 1. The largest subsystems generated by CRA and All-at-once approaches

#nodes	CRA based on ROE		CRA based on CFFD		All-at-once	
	#states	#transitions	#states	#transitions	#states	#transitions
2	13	21	90	176	110	288
3	43	143	482	1255	825	2820
4	153	593	3075	12244	5500	23200
5	525	2367	27999	138242	34375	172500
6	1793	9177	266805	1581601	206250	1200000

6 Conclusion

The paper presents a new equivalence, reverse observation equivalence, which orients the invariant checking of LSTS. It is proved that the new equivalence is a congruence with respect to the basic parallel composition. The experiments show that ROE is quite efficient for the invariant checking of LSTS in the context of compositional reachability analysis.

Our future work will consider checking the deadlock and livelock properties of LSTS by combining the approaches based on observation equivalence and reverse observation equivalence.

References

1. Hansen, H., Virtanen, H., Valmari, A.: Merging state-based and action-based verification. In: Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03). (2003) 150–156
2. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

3. Alur, R., Henzinger, T.A.: Computer-Aided Verification: An Introduction to Model Building and Model Checking for Concurrent Systems. (1998)
4. Cheung, S.C., Kramer, J.: Checking subsystem safety properties in compositional reachability analysis. In: Proceedings of the 18th international conference on Software engineering, IEEE Computer Society (1996) 144–154
5. Cheung, S.C., Kramer, J.: Enhancing compositional reachability analysis with context constraints. In: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering, ACM Press (1993) 115–125
6. Tai, K.C., Koppol, V.: Hierarchy-based incremental reachability analysis of communication protocols. In: Proceedings of the IEEE International Conference on Network Protocols. (1993)
7. Yeh, W.J., Young, M.: Compositional reachability analysis using process algebra. In: Proceedings of the symposium on Testing, analysis, and verification, ACM Press (1991) 49–59
8. Valmari, A.: Compositionality in state space verification methods. In: Proc. 17th International Conference in Application and Theory of Petri Nets (ICATPN'96), Osaka, Japan. Volume 1091 of LNCS., Springer-Verlag (1996) 29–56
9. Cheung, S.C., Kramer, J.: Context constraints for compositional reachability analysis. ACM Trans. Softw. Eng. Methodol. **5** (1996) 334–377
10. Valmari, A.: Compositional state space generation. Technical Report A-1991-5, Department of Computer Science, University of Helsinki, Finland (1991)
11. Milner, R.: Communication and concurrency. Prentice-Hall, Inc. (1989)
12. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM (JACM) **31** (1984) 560–599
13. Karsisto, K.: A New Parallel Composition Operator for Verification Tools. PhD thesis, Tampere University of Technology Publications 420, Tampere, Finland (2003)
14. van Gabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM (JACM) **43** (1996) 555–600
15. Lynch, N., Vaandrager, F.: Forward and backward simulations part 1: Untimed systems. Information and Computation **121** (1995) 214–233
16. Sabnani, K., Lapone, A., Uyar, M.: An algorithmic procedure for checking safety properties of protocols. IEEE Trans. Commun. **37** (1989) 940–948

Minimal Spanning Set for Coverage Testing of Interactive Systems

Fevzi Belli and Christof J. Budnik

University of Paderborn, Warburger Str., 100, 33098 Paderborn, Germany
{belli, budnik}@adt.upb.de

Abstract. A model-based approach for minimization of test sets for interactive systems is introduced. Test cases are efficiently generated and selected to cover the behavioral model and the complementary fault model of the *system under test* (*SUT*). Results known from state-based conformance testing and graph theory are used and extended to construct algorithms for minimizing the test sets, considering also structural features of the SUT.

1 Introduction

Testing is the traditional validation method in the software industry. There is no justification, however, for any assessment on the correctness of the SUT based on the success (or failure) of a single test, because there can potentially be an infinite number of test cases, even for very simple programs. To overcome this shortcoming of testing, formal methods have been proposed, which introduce models that represent the relevant features of the SUT. The modeled, relevant features are either functional behavior or the structural issues of the SUT, leading to *specification-oriented testing* or *implementation-oriented testing*, respectively. This paper is on specification-oriented testing; i.e., the underlying model represents the system behavior interacting with the user's actions. The system's behavior and user's actions will be viewed here as *events,* more precisely, as *desirable events* if they are in accordance with the user expectations. Moreover, the approach includes modeling of the faults as *undesirable events* as, mathematically spoken, a complementary view of the behavioral model.

Based on [3], this paper introduces a novel, graphical representation of both the behavioral model and the fault model of the SUT. Algorithms are introduced for the coverage of these models by a minimal set of test cases (*minimal spanning set for coverage testing*). The next section summarizes the related work before Section 3 introduces the fault model and the test process. The optimization of the test case set is discussed in Section 4. Section 5 considers the structure of the SUT to avoid unnecessary and/or infeasible tests. Supporting tools are introduced in Section 6. Section 7 summarizes the results and sketches the research work planned.

2 Related Work

Methods based on finite-state automata have been used for almost four decades for the specification and testing of system behavior, e.g., for specification of software

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 220–234, 2005.
© Springer-Verlag Berlin Heidelberg 2005

systems [8], as well as for conformance and software testing [6, 1, 20, 18]. Also, the modeling and testing of interactive systems with a state-based model has a long tradition [19, 13, 21, 25]. These approaches analyze the SUT and model the user requirements to achieve sequences of *user interaction* (*UI*), which then are deployed as test cases. [25] introduced a simplified state-based, graphical model to represent UIs; this model has been extended in [3] to consider not only the desirable situations, but also the undesirable ones. This strategy is quite different from the combinatorial ones, e.g., *pairwise testing,* which requires that for each pair of input parameters of a system, every combination of these parameters' valid values must be covered by at least one test case. It is, in most practical cases, not feasible [22] to test UIs.

A similar fault model as in [3] is used in the mutation analysis and testing approach which systematically and stepwise modifies the SUT using *mutation operations* [10]. This approach has been well understood, is widely used and, thus, has become quite popular. Although originally applied to implementation-oriented unit testing, mutation operations have also been extended to be deployed at more abstract, higher levels, e.g., integration testing, state-based testing, etc. [9]. Such operations have also been independently proposed by other authors, e.g., "state control faults" for fault modeling in [7], or for "transition-pair coverage criterion" and "complete sequence criterion" in [18]. However, the latter two notions have been precisely introduced in [3] and [25], respectively, earlier than in [18].

Another state-oriented group of approaches to test case generation and coverage assessment is based on model checking, e.g., the Software Cost Reduction method, as described in [12]. These approaches identify negative and positive scenarios to generate and select test cases automatically from formal requirements specifications. A different approach, especially for *graphical user interface* (*GUI*) testing, has been introduced in [16]; it deploys methods of knowledge engineering to generate test cases, test oracles, etc., and to deal with the test termination problem. All of these approaches use some heuristic methods to cope with the state explosion problem.

This paper also presents a method for test case generation and test case selection. Moreover, it addresses test coverage aspects for test termination, based on [3], which introduced the notion of "minimal spanning set of complete test sequences", similar to "spanning set", that was also discussed in [15]. The present paper considers existing approaches to optimize the round trips, i.e., the Chinese Postman Problem [1], and attempts to determine algorithms of less complexity for the spanning of walks, rather than tours, related to [24, 17].

3 Fault Model and Test Process

This work uses *Event Sequence Graphs (ESG)* for representing the system behavior and, moreover, the facilities from the user's point of view to interact with the system. Basically, an *event* is an externally observable phenomenon, such as an environmental or a user stimulus, or a system response, punctuating different stages of the system activity.

3.1 Preliminaries

Definition 1. An Event Sequence Graph ESG=(V,E) is a directed graph with a finite set of *nodes* (*vertices*) $V \neq \emptyset$ and a finite set of *arcs* (*edges*) $E \subseteq V \times V$.

For representing user-system interactions, the nodes of the ESG are interpreted as events. The operations on identifiable components of the UI are controlled/perceived by input/output devices, i.e., elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of V and lead interactively to a succession of user inputs and system outputs.

Definition 2. Let V, E be defined as in Def. 1. Then any sequence of nodes $\langle v_0,\ldots,v_k\rangle$ is called an (legal) *event sequence* (*ES*) if $(v_i, v_{i+1}) \in E$, for $i=0,\ldots,k-1$.

Furthermore, α (*initial*) and ω (*end*) are functions to determine the initial node and end node of an ES, i.e., $\alpha(ES)=v_0$, $\omega(ES)=v_k$. Finally, the function l (*length*) of an ES determines the number of its nodes. In particular, if $l(ES)=1$ then $ES=\langle v_i\rangle$ is an ES of length 1. An $ES=\langle v_i, v_k\rangle$ of length 2 is called an *event pair* (*EP*).

The assumption is made that there is an ES from the single node ε to all other nodes, and from all nodes there is an ES to the single node γ ($\varepsilon, \gamma \notin V$). ε is called the *entry* and γ is called the *exit* of the ESG.

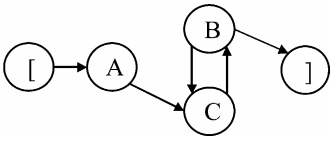


Fig. 1. An ESG with [as entry and] as exit

The entry and exit, represented by '[' and ']', respectively, are not included in V. They enable a simpler representation of the algorithms to construct minimal spanning test case sets (Section 4).

Definition 3. An ES is called a *complete ES* (*Complete Event Sequence*, *CES*), if $\alpha(ES)=\varepsilon$ is the entry and $\omega(ES)=\gamma$ is the exit.

CESs represent *walks* from the entry '[' of the ESG to its exit ']'.

Definition 4. The node w is a *successor event* of v and the node v is a *predecessor event* of w if $(v, w) \in E$. The *difference* of a node $u \in V$ *diff(u)* is defined as the number of predecessor events reduced by the number of successor events.

Definition 5. Let two ESGs be defined as $ESG_i = (V_i, E_i)$, $i = 1,2$. ESG_1 is a *subgraph* of ESG_2 if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. ESG_1 is an induced subgraph by a set of nodes.

3.2 Fault Model and Test Terminology

Definition 6. For an $ESG=(V, E)$, its *completion* is defined as $\widehat{ESG} = (V, \hat{E})$ with $\hat{E} = V \times V$.

Definition 7. The *inverse* (or *complementary*) *ESG* is then defined as $\overline{ESG} = (V, \overline{E})$ with $\overline{E} = \hat{E} \setminus E$ (\: set difference operation).

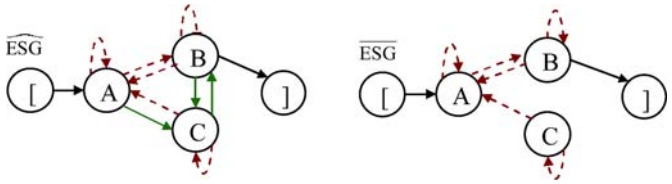


Fig. 2. The completion $\widehat{ESG}$ and inversion $\overline{ESG}$ of Fig. 1

Note: Entry and exit are not considered while constructing the $\overline{ESG}$.

Definition 8. Any EP of the $\overline{ESG}$ is a *faulty event pair* (*FEP*) for ESG.

Definition 9. Let $ES=\langle v_0,\ldots,v_k\rangle$ be an event sequence of length $k+1$ of an ESG and $FEP=\langle v_k, v_m\rangle$ a faulty event pair of the according ESG . The concatenation of the ES and FEP forms then a *faulty event sequence* $FES=\langle v_0,\ldots,v_k, v_m\rangle$.

Definition 10. An *FES* will be called *complete* (*Faulty Complete Event Sequence*, *FCES*) if $\alpha(FES)=\varepsilon$ is the entry. The ES as part of a FCES is called a *starter*.

3.3 Test Process

Definition 11. A *test case* is an ordered pair of an input and expected output of the SUT. Any number of test cases can be compounded to a *test set* (or, a *test suite*).

Once a test set has been constructed, tests can be run applying the test cases to the SUT. If it behaves as expected, the SUT *succeeds* the test, otherwise it *fails* the test. The approach introduced in this paper uses event sequences, more precisely CES, and FCES, as test inputs. If the input is a CES, the SUT is supposed to proceed it and thus, to succeed the test. Accordingly, if a FCES is used as a test input, a failure is expected to occur. The latter case represents an exception that must be properly handled by the system, i.e., the SUT is supposed to refuse the proceeding and produce a warning. The test process is sketched in Algorithm 1.

Algorithm 1. Test Process

n := number of the functional units (modules) of the system that fulfill a well-defined task
length := required length of the test sequences

```
FOR function1 TO n DO
  Generate appropriate ESG and ESG
  FOR k:=2 TO length DO                                    //Section 4.3
    Cover all ESs of length k by means of CESs
    subject to minimizing the number and total length of the CES   //Section 4.1
  Cover all FEPs of by means of FCESs
  subject to minimizing the total length of the FCESs      //Section 4.2
Apply the test set to the SUT
Observe the system output to determine whether the system response is in
compliance with the expectation
```

To determine the point in time in which to stop testing, a criterion is necessary to systematize the test process and to judge the efficiency of the test cases. The approach converts this problem into the *coverage of the ES and FES of length k of the* $\widehat{ESG}$.

The test costs are given by the minimized total length of the CESs and FCESs. The length of the ESs can be increased stepwise. This enables a scalability of the test costs which are proportional to the length of the ESs.

4 Minimizing the Spanning Set

The union of the sets of CESs of minimal total length to cover the ESs of a required length is called *Minimal Spanning Set of Complete Event Sequences (MSCES).*

If a CES contains all EPs at least once, it is called an *entire walk*. A legal entire walk is *minimal* if its length cannot be reduced. A minimal legal walk is *ideal* if it contains all EPs exactly once. Legal walks can easily be generated for a given ESG as CESs, respectively. It is not, however, always feasible to construct an entire walk or an ideal walk. Using some results of the graph theory [24], MSCESs can be constructed as follows:

- Check whether an ideal walk exists.
- If not, check whether entire walks exist. If yes, construct a minimal one.
- If there is no entire walk, construct a set of walks with minimal total length to cover all ES.

4.1 An Algorithm to Determine Minimal Spanning Complete Event Sequence

A similar problem to the determination of MSCESs is the *Directed Chinese Postman Problem* [23]. In the following, some results are summarized that are relevant to determine the test costs and enable scalability of the test process.

The Algorithm 2 determines a set of walks with the minimal total length to cover all EPs and requires that this graph be strongly connected, which can be done through an additional arc from the final to the entry (Fig. 3). The figures within the nodes in Fig. 3 indicate the calculated differences (Definition 4) of these nodes. These balance values determine the number of additional EPs that will be identified by searching the all-shortest-path and solving the optimization problem by the Hungarian method [14]. The problem can then be transferred to the construction of the Euler tour for this graph [24].

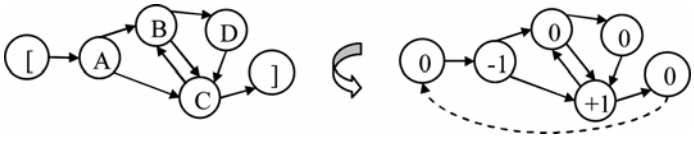


Fig. 3. Transferring walks into tours and balancing the nodes

Algorithm 2. Determination of the MSCES

Input: *ESG=(V, E)*; ε=[, γ=]
Output: *MSCES*

```
addArc(ESG,(γ,ε));                    //insert arc from ] to [
sets A,B,M,MSCES := ∅                 //empty sets
FOR EACH v∈V DO
  IF (diff(v)>0) THEN
    A := A ∪ {v_i |i∈{1,..,diff(v)}};
  IF (diff(v)<0) THEN
    B := B ∪ {v_i |i∈{1,..,diff(v)}};
m := |A| := |B|;                      //cardinality
D[1..m][1..m];                        //distance matrix D
FOR EACH v∈A DO          //compute all shortest paths from v to all b∈B
  computeShortestPaths(v,B,D);  //shortest distances are saved in D
M := solveAssignmentProblem(D);
 //M = {(i, j) | one-to-one mapping: i∈{1,..,m}→j∈{1,..,m}} by Hungarian
method
FOR EACH (i,j)∈M DO
  Path := getShortestPath(i,j);
  FOR EACH e∈Path DO
    addArc(ESG,e);
EulerTourList := computeEulerTour(ESG);  //tour starts in ε
                          //EulerTourList = (ε,...,γ,ε,...,γ,ε,...,γ,ε)
start := 1;
FOR i:=2 TO length(EulerTourList)-1 DO
  IF (getElement(EulerTourList,i) = γ) THEN
    MSCES := MSCES ∪ get-
PartList(EulerTourList,start,i);
    start := i+1;          //MSCES = {(ε,...,γ),(ε,...,γ),(ε,...,γ),...}
RETURN MSCES;
```

The function `addArc(ESG,(u,v))` inserts a new arc from the node *u* to the node *v* of the *ESG*. The function `computeShortestPath()` determines all shortest paths from a node *v* to all $b \in B$ with BFS algorithm and stores these shortest distances in the matrix *D* for later usage by the function `getShortestPath()`. The function `solveAssignmentProblem()` returns a one-to-one mapping of the unbalanced nodes. The function `computeEulerTour()` determines the Euler tour of the ESG. The euler tour will be decomposed into subsequences by the function `getPartList()`.

In Algorithm 2, the ESG is represented by its adjacency matrix. The algorithm consists of three sections:

- Determination of all-shortest-paths by Floyds algorithm with the complexity O(|V|3) [2]. However, because the ESG is a non-weighted digraph, the complexity can be decreased by using the Breadth-First-Search down to O(|V|·|E|). This results from the fact that:

  - Breadth-First-Search algorithm determines the shortest path from one node of the ESG to all other ones in $O(|E|)$ as $|E|>|V|+1$.
  - Breadth-First-Search algorithm iterates $|V|$ times to handle all nodes.
- The optimizing problem, which is solved in accordance with [14] by the Hungarian method, with the complexity $O(|V|^3)$.
- Computation of an Euler tour with the complexity of $O(|V|\cdot|E|)$ [24].

To sum up, the MSCES can be solved in $O(|V|^3)$ time. Note that no entire walk exists for the example. Therefore, an ideal walk cannot be constructed.

4.2 Minimal Spanning Set for the Coverage of Faulty Event Sequences

The union of the sets of FCESs of the minimal total length to cover the FESs of a required length is called *Minimal Spanning Set of Faulty Complete Event Sequences (MSFCES).*

In comparison to the interpretation of the CESs as legal walks, *illegal walks* are realized by FCESs that never reach the exit. An illegal walk is *minimal* if its starter cannot be shortened.

Assuming that an ESG has n nodes and d arcs as EPs to generate the CESs, then exactly $u:=n^2-d$ arcs are FEPs. Thus, at most u FCESs of minimal length, i.e., of length 2, are available; those FCESs emerge when the node(s) after entry is (are) followed immediately by a faulty input. The number of FCESs is precisely determined by the number of FEPs. FEPs that represent FCES are of constant length 2; thus, they also cannot be shortened. It remains to be noticed that only the starters of the remaining FEPs can be minimized, e.g., using the algorithm given in [11].

While constructing the MSCESs one can exclude the ESs that are already used to form starters to construct MSFCESs. This can help save costs if the test budget is very limited, as is very often the case in practice.

4.3 Generating Event Sequences with Length > 2

A phenomenon in testing interactive systems most testers are familiar with, is that faults can be frequently detected and reproduced only in some context. This makes a test sequence of a length>2 necessary since repetitive occurrences of some subsequences are needed to cause an error to occur/re-occur.

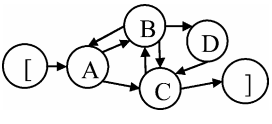


Fig. 4. Static faults vs. dynamic faults

Consider the following scenario: Based on the ESG given in Fig. 4, the tester assumedly observes that the EP given by ***BC*** always reveals a fault, no matter if executed within *[A**BC**]*, *[ABA**BC**]*, or *[ABDC**BC**]*; i.e., the test cases containing *BC* al-

ways detect the fault in any context. In this case, the fault is said to be a *static* one, as it can be detected without a context. Furthermore, the same scenario (so the assumption) demonstrates that the EP ***BA*** reveals another fault, but only in the context of *[ABC**BA**C]*, and never within *[A**BA**C]*, or *[A**BA**CBDC],* etc. In this case the fault is said to be a *dynamic* one.

Such observations clearly indicate that the test process must be applied to longer ESs than 2 (EPs).

Therefore an ESG can be transformed into a graph in which the nodes can be used to generate test cases of length > 2, in the same way that the nodes of the original ESG are used to generate EPs and to determine the appropriate MSCES.

Fig. 5 illustrates the generation of ESs of length=3. In this example adjacent nodes of the extended ESG are concatenated, e.g., AB is connected with *BD*, leading to *ABBD*. The shared event, i.e., *B*, occurs only once producing *ABD* as an ES of length=3. In case ESs of length=4 are to be generated, the extended graph must be extended another time using the same algorithm.

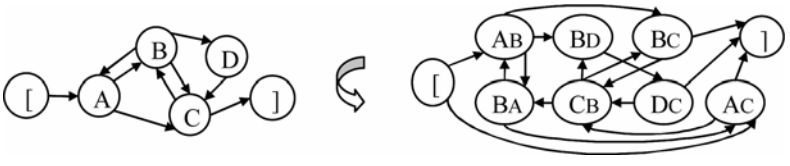


Fig. 5. Extending the ESG for covering ESs of length=3

The common valid of this approach is given by Algorithm 3. Therein the notation `ES(ESG,i)` represents the identifier, e. g., *AB*, of the node *i* of the *ESG*. This identifier can be concatenated with another identifier `ES(ESG,j)` of the node *j*, e.g., *CD*. This is represented by *AB* ⊕ *CD*, or `ES(ESG,i)` ⊕ `ES(ESG,j)`, resulting in the new identifier *ABCD*. Note that the identifiers of the newly generated nodes to extend the ESG will be made up using the identifiers of the existing nodes. The function `addNode()` inserts a new ES of length *k*. Following this step, a node *u* is connected with a node *v* if the last *n-1* events that are used in the identifier of *u* are the same as the first *n-1* events that are included in the identifier of *v*. The function `addArc()` inserts an arc, connecting *u* with *v* in the ESG. The pseudo nodes [,] are connected with all the extensions of the nodes with which they were connected before the extension. In order to avoid traversing the entire matrix, arcs which are already considered are to be removed by the function `removeArc()`.

Apparently, the Algorithm 3 has a complexity of $O(|V|^2)$ because of the nested FOR-loops to determine the arcs in the *ESG'*. A further algorithm to generate FESs of length > 2 is not necessary because such faulty sequences will be constructed through the concatenation of the appropriate starters with the FEPs. Algorithm 2 can be applied to the outcome of the Algorithm 3, i.e., to the extended ESG, to determine the MSCES for *l(ES)* > 2.

Algorithm 3. Generating ESs and FESs with length > 2

Input: *ESG=(V, E)*; $\varepsilon = [$, $\gamma =]$, *ESG'=(V', E')* with $V'=\varnothing$, $\varepsilon'=[$, $\gamma'=]$;
Output: *ESG'=(V', E')*, $\varepsilon'=[$, $\gamma'=]$;

```
FOR EACH (i,j)∈ E with (i<>ε) AND (j<>γ) DO
  addNode(ESG',(ES(ESG,i) ⊕ ω(ES(ESG,j)));
  removeArc(ESG,(i,j));
FOR EACH i∈ V' with (i<>ε') AND (i<>γ') DO
  FOR EACH j∈ V' with (j<>ε') AND (j<>γ') DO
    IF(ES(ESG',i) ⊕ ω(ES(ESG',j)) =
                    α(ES(ESG',i)) ⊕ (ES(ESG',j)) THEN
      addArc(ESG',(i,j));
  FOR EACH (k,l)∈E with k=ε DO
    IF(ES(ESG',i) = ES(ESG,l) ⊕ω(ES(ESG',i)) THEN
      addArc(ESG',(ε',i));
  FOR EACH (k,l)∈ E with l=γ DO
    IF(ES(ESG',i) = α(ES(ESG',i)) ⊕ ES(ESG,k) THEN
      addArc(ESG',(i,γ'));
RETURN ESG';
```

5 Exploiting the Structural Features

The approach has been applied to the testing and analysis of the GUIs of different kind of systems, leading to a considerable amount of practical experience. A great deal of test effort could be saved considering the structural features of the SUT. Thus, there is further potential for the reduction of the cost of the test process.

5.1 A Practical Example

Fig 6 depicts a small part of the GUI of an MS WordPad-like word processing system. This GUI will usually be active when a text portion is to be loaded from a file, or to be manipulated by cutting, copying, or pasting. The GUI will also be used for saving the text to the current file (or to another one). The optional events are abbreviated in the Fig. 7 with capital letters. There are still more window components, but they will not be explained here further. The described components are used to traverse through the entries of the menu and sub-menus, creating many combinations and accordingly, many applications.

The GUI represented in Fig. 6 is transferred to an ESG (Fig. 7). is easy to understand, but an informal and imprecise presentation of the GUI, while Fig. 7 is a formal presentation that neglects some aspects, e.g., the hierarchy, while still being precise.

The conversion of Fig. 6 into Fig. 7 is the most abstract step of the approach that must be done manually, requiring some practical experience and theoretical skill in

designing GUIs. Example 1 lists the FCESs to cover the FEPs of the ESGs `Main/Open` given in Fig. 7.

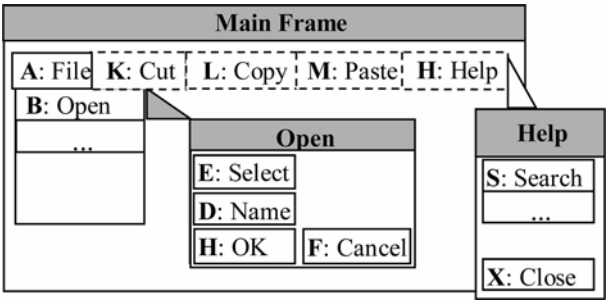


Fig. 6. Top-level GUI of WordPad, modal/modeless windows

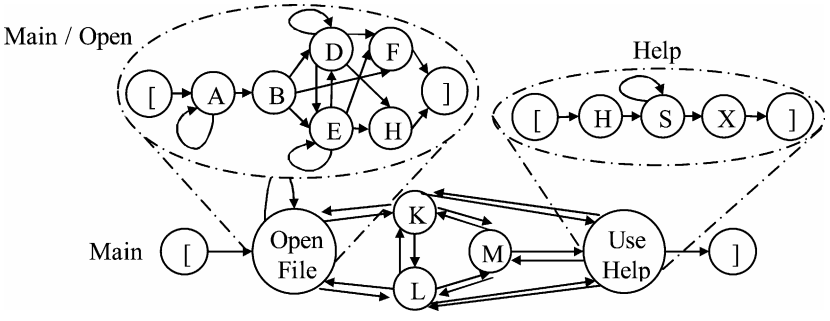


Fig. 7. ESG of the GUI represented in

Example 1. *AD*, *AE*, *AF*, *AH*, A*BA*, A*BB*, A*BH*, AB*DA*, AB*DB*, AB*EA*, AB*EB*, AB*FB*, AB*FF*, AB*FE*, AB*FD*, AB*FH*, AB(E+D)*HA*, AB(E+D)*HB*, AB(E+D)*HD*, AB(E+D)*HE*, AB(E+D)*HF*, AB(E+D)*HH*

5.2 Modal and Modeless Windows

Analysis of the structure of the GUIs, e.g., the example GUI in Fig. 6., delivers the following features:

- Windows of commercial systems are nowadays mostly hierarchically structured, i.e., the root window invokes children windows that can invoke further (grand) children, etc.
- Some children windows can exist simultaneously with their siblings and parents; they will be called *modeless* (or *non-modal*) windows. Other children, however, must “die”, i.e., close, in order to resume their parents (*modal* windows).

For the main frame of the WordPad, the child window `Help` is a modeless window; the other child window, `Open`, is a modal one. Fig. 8 represents these windows as a "family tree". In this tree, a unidirectional edge indicates a modal parent-child relationship. A bidirectional edge indicates a modeless one.

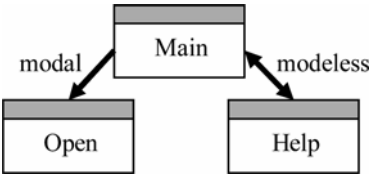


Fig. 8. Modal windows vs. modeless windows

Modal windows must be closed before any other window can be invoked. Therefore, modal windows can be tested without taking the other windows into account, i.e., it is not necessary to consider the combinations of the ESs and FESs of the parent and children. Thus, similar to the strong-connectedness and symmetrical features [21], the modality feature is extremely important for testing since it avoids unnecessary test efforts. Note that this is true only for the FCESs and MSFCESs as test inputs considering the structure information might impact the structure of the ESG, but not the number of the CESs and MSCESs as test inputs. Fig. 9 represents the modified ESG of the WordPad. The modification, which separates the events *A* and *B* from `Open`, takes the modality into account that avoids unnecessary combinations of EPs and FEPs. Example 2 lists the MSFCESs to cover FEPs of the sub-graph `Open` given in Fig. 9.

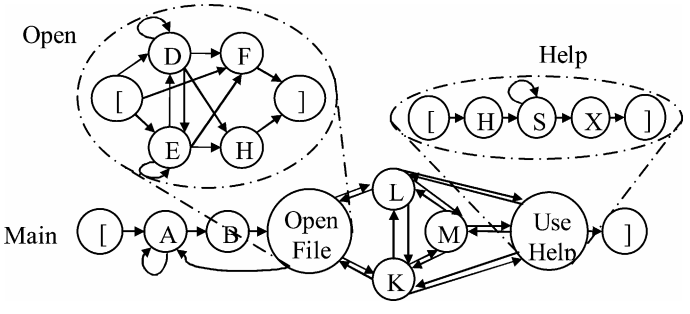


Fig. 9. Modified ESG of the GUI in Fig. 8, taking the modality feature into account

Example 2. *(E+D)FD*, *(E+D)FE*, *(E+D)FF*, *(E+D)FH*, *(E+D)HF*, *(E+D)HD*, *(E+D)HF*, *(E+D)HH*

Already this example, i.e., the comparison of Example 1 (22 FEPs) with Example 2 (8 FEPs), demonstrates the efficiency increase through the exploitation of the structural features of the SUT.

6 Tool Support

The determination of the MSCESs/MSFCESs can be very time consuming when carried out manually. Also, the gaining of the structural information that is necessary to reduce the number of MSFCESs is frequently a rather costly process. Thus, tools of different categories are necessary for both purposes.

A good software engineering practice ensures that the system behavior has been modeled during the system design. Otherwise the model has to be constructed manually afterwards, according to the specification.

6.1 Test Case Generation

For the generation of test cases the tool *GenPath* [5] has been developed to accept the adjacency matrix of the ESG as input. The user can, however, input several ESGs which can also be subgraphs of the vertices of the ESG itself under consideration. Fig. 10 represents the GUI of GenPath which generates MSCESs for ESs of required length. Moreover, it represents the ESG under consideration and marks its EPs with the underlying algorithm traces.

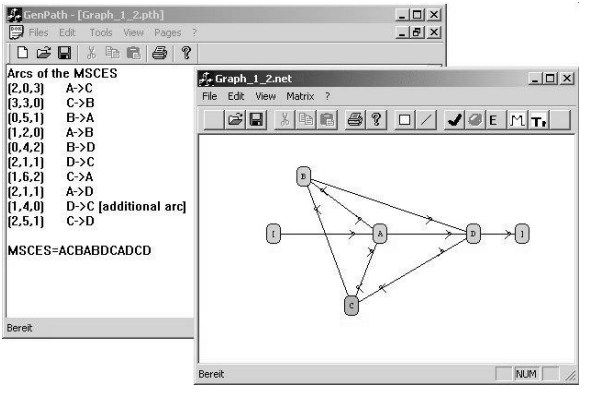


Fig. 10. GenPath to generate MSCES

6.2 Generation of GUI Structure

Section 5 explained the necessity to consider the specific information on the structure of the SUT in order to reduce the number of test cases. This structural information can be obtained with a commercially available Capture-Playback facility, as to WinRunner of Mercury Interactive [26] delivers.

This tool can identify all available windows of a GUI-application and generates automatically information on the windows hierarchy that can be assembled to determine modal/modeless windows of the SUT. Fig. 11 represents a part of WordPad that the test environment has traced. The keyword `opened_by` identifies the child window `Open`. The parent window can be traced via the keyword `menu_select_item()`.

```
Open:
{class: window,
  label: Open,
  enabled: 1,
  module_name: "C:\\Windows\\Tool\\WordPad.exe",
  nchildren: 17
}
   { rtree_state: open,
     ltree_state: open,
     lrn_app_stat: done,
     parent_win: "WordPad - [Document 1]",
     opened_by: "menu_select_item(\"Open... Strg+O\");"
   }
```

Fig. 11. Excerpt out of the WinRunner file with information on a GUI structure

7 Validation

A separate study has applied the proposed approach to a selected significant function of the personal music management system RealJukebox (RJB), Version 2, of RealNetworks. This function enables the user to load a CD, select a track, and play it. The user can then change the mode, replay the track, or remove the CD, load another one, etc.

Table 1. Reducing the number of test cases

Length	#CES	#MSCES	Cost Reduction ES
2	40	15	62.5 %
3	183	62	66.1 %
4	849	181	78.7 %
Sum	1072	258	76.0 %
Length	#MSFCES without structural information	#MSFCES with structural information	Cost Reduction MSFCES
2	75	58	22.7 %
3	339	218	35.7 %
4	1587	632	60.2 %
Sum	2001	908	54.6 %

For a comprehensive testing, several strategies have been developed with varying characteristics of the test inputs, i.e.,

- the length and number of the test sequences, and
- the type of the test sequences, i.e., CES- and FCESs-based.

This study delivered following findings:

- The test cases of the length 4 were more effective in revealing dynamic faults than the test cases of the lengths 2 and 3. They were, however, considerably more expensive in terms of costs per detected fault.
- The CES-based test cases as well as the FCES-based cases were effective in detecting faults.

To summarize the test process, one student tester, who acted also as oracle, carried out 1166 tests semi-automatically over a period of 2 days, working, on average, 8 hours per day, thus spending a total of 78560 seconds. These figures result in approximately 67 seconds per test. A total of 32 faults were detected.

The results of the research for minimizing the spanning set of the test cases (MSCES and MSFCES), as described in Section 4, has been applied to the testing of the selected significant function. Table 1 summarizes that the algorithmic minimization (Section 4.1 and 4.2) could save about 75 % of the test costs, while the exploitation of the structural information of the SUT could save up to almost 50%!

A more detailed discussion about the benefits, e.g., concerning the number of detected errors in dependency of the length of the test cases, is given in [4].

8 Conclusion and Future Work

This paper has introduced an integrated approach to coverage testing of interactive systems, incorporating modeling of the system behavior with fault modeling and minimizing the test sets for the coverage of these models. The framework is based on the concept of "event sequence graphs (ESG)". Event sequences (ES) represent the human-computer interactions. An ES is complete (CES) if it produces desirable, well-defined and safe system functionality. The notion of complete faulty event sequences mathematically complements this view.

The objective of testing is the construction of a set of CESs of minimal total length that covers all ESs of a required length. A similar optimization problem arises for the validation of the SUT under exceptional, undesirable situations which are modeled by faulty event sequences (FESs) and complete FESs (FCESs). The paper applied and modified some algorithms known from graph theory to these problems. Furthermore, it was shown how the structure of interactive systems can be algorithmically exploited by a commercial test tool to reduce the test sets by infeasible and/or unnecessary test cases.

In the case of *safety*, the threat originates from within the system due to potential failures and its spillover effects causing potentially extensive damage to its environment. The goal for future work is to design defense actions, which is an appropriately enforced sequence of events, to prevent faults that could potentially lead to such failures. Further future work concerns cost reduction through automatic, or semiautomatic modification of a given ESG in order to consider modality of interaction structures.

References

1. A. V. Aho, A. T. Dahbura, D. Lee, M. Ü. Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours", *IEEE Trans. Commun*. 39, pp. 1604-1615, 1991
2. R. K. Ahuja, T. L. Magnanti, J. B. Orlin, "Network Flows-Theory, Algorithms and Applications", Prentice Hall, 1993.
3. F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces", Proc. 12th ISSRE, pp. 34-43, 2001

4. F. Belli, N. Nissanke, Ch. J. Budnik, "A Holistic, Event-Based Approach to Modeling, Analysis and Testing of System Vulnerabilities"; Technical Report TR 2004/7, Univ. Paderborn, 2004
5. Ch. J. Budnik, A. Hollmann, R. Moge, „GenPath – A Tool to Generate Paths of different Lengths of an Event Sequence Graph", Technical Report TR 2004/9, Univ. Paderborn, 2004
6. R.V. Binder, "*Testing Object-Oriented Systems*", Addison-Wesley, 2000
7. G. V. Bochmann, A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing", *Softw. Eng. Notes, ACM SIGSOFT*, pp. 109-124, 1994
8. Tsun S. Chow, "Testing Software Designed Modeled by Finite-State Machines", *IEEE Trans. Softw. Eng.* 4, pp. 178-187, 1978
9. M.E. Delamaro, J.C. Maldonado, A. Mathur, "Interface Mutation: An Approach for Integration Testing", *IEEE Trans. on Softw. Eng.* 27/3, pp. 228-247, 2001
10. R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *Computer* 11/4, pp. 34-41, 1978
11. Edsger. W. Dijkstra, "A note on two problems in connexion with graphs.", Journal of Numerische Mathematik, Vol. 1, pp. 269-271, 1959
12. A. Gargantini, C. Heitmeyer, „Using Model Checking to Generate Tests from Requirements Specification", *Proc. ESEC/FSE '99*, ACM SIGSOFT, pp. 146-162, 1999
13. J. Jorge, N.J. Nunes, J.F. Cunha (Eds.), "Interactive Systems – Design, Specification, and Verification", LNCS 2844, Springer-Verlag, 2003
14. D.E. Knuth, "The Stanford GraphBase", Addison-Wesley, 1993
15. M. Marré, A. Bertolino, "Using Spanning Sets for Coverage Testing", *IEEE Trans. on Softw. Eng.* 29/11, pp. 974-984, 2003
16. A. M. Memon, M. E. Pollack and M. L. Soffa, "Automated Test Oracles for GUIs", *SIGSOFT 2000*, pp. 30-39, 2000
17. S. Naito, M. Tsunoyama, "Fault Detection for Sequential Machines by Transition Tours", *Proc. FTCS*, pp. 238-243, 1981
18. J. Offutt, L. Shaoying, A. Abdurazik, and Paul Ammann, "Generating Test Data From State-Based Specifications", *The Journal of Software Testing, Verification and Reliability, 13(1):25-53, Medgeh 2003.*
19. D.L. Parnas, "On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System", Proc. 24th ACM Nat'l. Conf., pp. 379-385, 1969
20. B. Sarikaya, "Conformance Testing: Architectures and Test Sequences", *Computer Networks and ISDN Systems* 17, North-Holland, pp. 111-126, 1989
21. R. K. Shehady and D. P. Siewiorek, "A Method to Automate User Interface Testing Using Finite State Machines", in *Proc. Int. Symp. Fault-Tolerant Computing FTCS-27*, pp. 80-88, 1997
22. K. Tai, Y. Lei, "A Test Generation Strategy for Pairwise Testing", *IEEE Trans. On Softw. Eng.* 28/1, pp. 109-111, 2002
23. H. Thimbleby "The Directed Chinese Postman Problem", School of Computing Science, Middlesex University, London
24. D.B. West, "Introduction to Graph Theory", Prentice Hall, 1996
25. L. White and H. Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences", in Proc ISSRE, IEEE Comp. Press, pp. 110-119, 2000
26. WinRunner, Mercury Interactive, `http://www.mercuryinteractive.com`

An Approach to Integration Testing Based on Data Flow Specifications*

Yuting Chen, Shaoying Liu, and Fumiko Nagoya

Faculty of Computer and Information Sciences, Hosei University,
Koganei-shi, Tokyo 184-8584, Japan
{i04t9001, sliu, i02t9012}@k.hosei.ac.jp

Abstract. Integration testing of programs based on formal specifications can benefit considerably from the comprehensibility of specifications. In this paper, we describe an approach to testing programs based on data-flow-oriented specifications by analyzing data flow paths and discussing criteria for test case generation. This approach suggests a specific way to generate test cases directly from formalized data flow diagrams and the associated textual specifications. We apply the approach in a case study of testing part of an ATM system to evaluate its effectiveness in fault detection and to uncover its weakness for further improvement.

Keywords: data flow diagrams, specification-based testing, SOFL, test cases generation.

1 Introduction

Testing programs based on formal specifications is an effective way to uncover faults leading to violation of their specifications and to therefore enhance their reliability [1][2]. The advance of research in this area also contributes to the spreading of application of formal specification techniques in industry due to the additional value provided by formal specifications serving as a firm foundation for program testing.

It is worth noticing that much work on specification-based testing so far has focused on operation level (i.e., unit testing) where an operation is defined using pre and postconditions [3][4][5][6][7][8]. An important reason for this situation is that test cases can be rigorously generated based on the pre and postconditions of an operation, and a test oracle can be easily derived for test result analysis from the specification. However, when we try to apply this principle to integration testing of a program composed of many operations (e.g., procedures, methods), two major problems will inevitably arise. Firstly, test cases can no longer be easily generated from a pre-post expression, because there is usually no explicit

* This work is supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research on Priority Areas (No.16016279).

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 235–249, 2005.
© Springer-Verlag Berlin Heidelberg 2005

pre-post expression for an integrated specification based on which the entire program is implemented. In many cases, derivation of a pre-post expression for an integrated specification (e.g., Z, VDM) is extremely difficult, if not impossible. Secondly, assuming test cases have been generated properly, evaluation of the test results becomes a difficulty, since no test oracle can be easily derived from an integrated specification for this purpose.

Although some of researchers, including ourselves, have experienced a way to generate test cases based on path coverage criteria in finite state machines [9][10][11][12][13], this does not resolve the problem with test results analysis: there is still a lack of effective way to derive a test oracle. In this particular case, a practical solution is to let human being (e.g., customer or the analyst) to act as an oracle. To this end, the comprehensibility of the formal specification becomes important, because a specification with poor understandability would not facilitate the human being to make correct judgments.

In accordance with our past experience of using both control flow and data flow specification languages, data flow specifications, such as SOFL (Structured Object-Oriented Formal Language) [14][15], is more understandable for ordinary customers or end-users than control flow specifications because of the similarity in concepts and functional representations between data flow specifications and their real world systems. For this reason, we believe that adopting data flow specifications in software development can be more effective in requirements acquisition and validation. While there have been studies on test case generation for integration testing of programs based on formal control flow specifications, there is few research on test case generation on the basis of data flow specifications. According to our experience, it is extremely difficult to correctly select data flow paths and to generate test cases based on a conventional data flow diagram that has no precise operational semantics.

In this paper, we describe a path oriented approach to integration testing of programs based on a formalized data flow diagram, known as *condition data flow diagram* (or CDFD for short), which is employed in the SOFL specification language [15]. In a SOFL specification, CDFDs are adopted to describe the architecture of the entire specification, while mathematically-based formal notation is used to define its components in the associated modules. CDFDs distinguish from the classical data flow diagrams in that they may use structures for controlling data flows conditionally, and nondeterministic processes to express alternative data flows. The essential idea of our approach is to select paths from a CDFD to cover all branches of the control structures and process functions, and then to generate test cases using the constraints defined by the associated textual specifications. Thus, test cases can experience the data transitions among processes in the CDFD, which are intuitive for testers to analyze the path coverage and the test results, and to make correct judgments on the existence of faults in the systems.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to the SOFL specification language. Section 3 describes an approach to test case generation for integration testing of programs based on a SOFL

specification. Section 4 presents a case study of testing part of an Automated Teller Machine (ATM) system to evaluate the proposed approach. Section 5 introduces the related work on testing based on data flow diagrams. Finally, in Section 6, we conclude the paper and point out future research.

2 Brief Introduction to SOFL

A SOFL specification is composed of a set of related modules in a hierarchy of *condition data flow diagrams* (CDFDs), as illustrated in Figure 1.

A CDFD (e.g., **A1** in Figure 1(a)) is a formalized DFD that specifies how processes work together to provide functional behaviors. A CDFD usually consists of control structures (conditional nodes, merging or separating nodes, and diverging nodes), processes, data flows, and data stores. A data flow, which represents a data transmission between two processes, is labeled with a variable. A process consists of five parts: name, input ports, output ports, precondition, and postcondition. It performs an action to transform input data flows satisfying precondition into output data flows that satisfy the postcondition. A conditional node denoted by a diamond or box allows a choice in moving data items between processes. A merging node composes input data flows into a single composite data flow, while a separating node is opposite to the merging node in that it breaks up the composite data flow into its components. A diverging node transforms an input data flow to either one of the output data flows or all of the output data flows, depending on the type of the diverging structure. The graphical constructs of control structures are given in Figure 2.

In Figure 2, (a) is a binary choice and the data flow x will flow along the upper branch when the condition C(x) is satisfied by x, otherwise x will flow

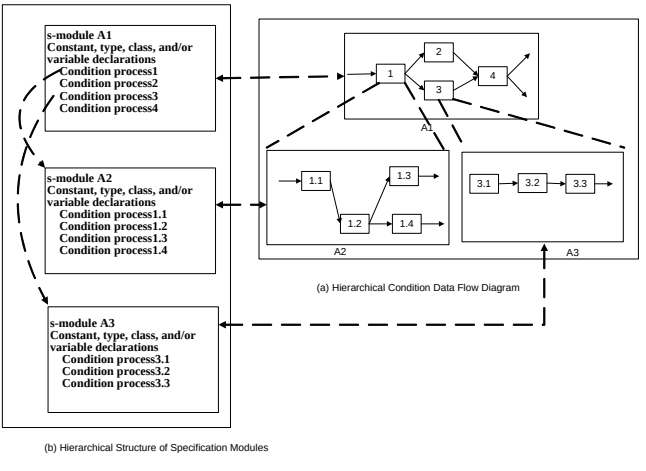


Fig. 1. Structure of a SOFL system

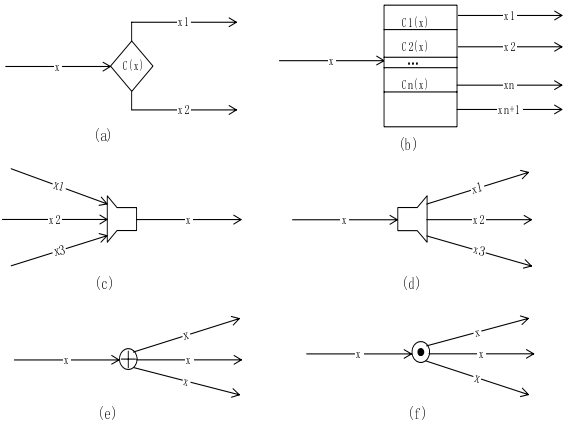


Fig. 2. Control structures in SOFL

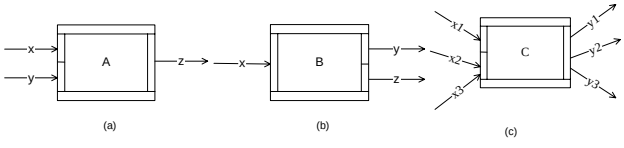


Fig. 3. Process with multiple ports

along the lower arc. (b) is a multiple selection structure and data from x will flow along a corresponding branch when x satisfy $\mathbf{C_i(x)}$**,** $\mathbf{i = 1...n}$, otherwise it will flow along the lowest level arc as x_{n+1}. (c) is a merging node to compose input data flows x1, x2, and x3 into a single composite data flow x, and (d) is a separating node to break up x into x1, x2, and x3. (e) and (f) are diverging nodes, in which the data flow x is nondeterministic to flow along one of the following arcs in (e), and (f) is a broadcasting node and x will flow along all of the output data flow arcs.

A process in a CDFD is different from that in the conventional DFDs in that it allows nondeterministic inputs and outputs; it may denote an abstraction of multiple functions. Figure 3 shows an example of nondeterministic processes.

Process A has two input ports receiving data flow x or y, and one output port holding the data flow z. When either x or y is available, process A takes x or y, but not both, as input, and produces z as output. Process B has one input port receiving data flow x, and produces either y or z, but not both. Process C supports inputs and outputs nondeterministically.

For each CDFD, a specification module (e.g., **module A1** in Figure 1(b)) is provided to define necessary types, related variables, and the functions of all the processes occurring in the CDFD.

3 A Path Coverage Approach

Assuming a SOFL specification is both internally consistent and valid, we introduce a path oriented approach to integration testing of a program based on its specification. This approach suggests the following steps:

1. Use the formal module as a foundation for eliminating nondeterministic processes in the CDFD;
2. Determine a set of data flow paths to cover all control branches and subprocesses in the CDFD. Here a data flow path is a sequence of data flows satisfying a certain condition;
3. Generate test cases from the formal module to ensure that every path is necessarily traversed and effectively tested;
4. Test the program and evaluate the testing results.

3.1 Processes Decomposition

The goal of decomposing a process is to divide a nondeterministic process into a set of deterministic subprocesses, each defining a single function of the original process. Thus, testing of the nondeterministic process can be divided into testing of its subprocesses, which is less complex and more controllable in accordance with the "divide and conquer" principle.

To facilitate discussions in this paper, we express a process, say P, in a SOFL specification as a 4-tuple $(I, O, pre, post)$ where

1. I is a finite set of inputs. If P is nondeterministic and has m input ports, we write $I = I_1|...|I_m$, denoting its input ports where
   (a) $I_i (1 \leqslant i \leqslant m)$ is the set of inputs received by the i^{th} input port;
   (b) $I_i \cap I_j = \phi$ $(1 \leqslant i, j \leqslant m, i \neq j$, ϕ is the empty set);
2. O is a finite set of outputs. If P has n output ports, we write $O = O_1|...|O_n$ where
   (a) $O_i (1 \leqslant i \leqslant n)$ is the set of outputs produced from the i^{th} output port;
   (b) $O_i \cap O_j = \phi$ $(1 \leqslant i, j \leqslant n, i \neq j)$;
3. *pre* and *post* are predicate expressions, denoting the pre and postconditions of process P, respectively. *Pre* may involve the variables defined in I, while *post* allows all the variables defined in both I and O to be used.

P can be divided into a set of deterministic subprocesses $\{sp_1, sp_2, ..., sp_k\}$, and we represent it as $P \twoheadrightarrow sp_1|sp_2|...|sp_k$, where

1. sp_i is called a sibling of sp_j $(1 \leqslant i, j \leqslant k, i \neq j)$.
2. each subprocess is deterministic to represent a single function of P. A subprocess sp_i is as well defined as $sp_i = (I^i, O^i, pre^i, post^i)$ where
   (a) $I^i \in \{I_1, ..., I_m\}$, $O^i \in \{O_1, ..., O_n\}$;
   (b) pre^i is a predicate expression denoting the precondition of sp_i; it involves only the variables defined in I^i;
   (c) $post^i$ is a predicate expression denoting the postcondition of sp_i; it may involve the variables defined in both I^i and O^i.

Consider the following definition of process C in Figure 3(c):

I : $\{x_1\} \mid \{x_2, x_3\}$
O : $\{y_1\} \mid \{y_2, y_3\}$
pre : $x1 > 10$ or $x1 < 5$ or $x2 > x3$ and ($x2 > 100$ or $x3 < 50$)
$post$: $y1 > x1 + 1$ or $y1 < x1$ or $y1 > x2$ and $y1 > x3$ or $y2 > x1$ and $y3 < x1$ or $y2 > x2$ or $y3 > y2$

then, process C is divided into the following four subprocesses

$$sp_1 = (I^1, O^1, pre^1, post^1)$$
$$sp_2 = (I^2, O^2, pre^2, post^2)$$
$$sp_3 = (I^3, O^3, pre^3, post^3)$$
$$sp_4 = (I^4, O^4, pre^4, post^4)$$

where

- $I^1, I^2 : \{x1\}$
- $I^3, I^4 : \{x_2, x_3\}$
- $O^1, O^3 : \{y_1\}$
- $O^2, O^4 : \{y_2, y_3\}$
- $pre^1, pre^2 : x1 > 10$ or $x1 < 5$
- $pre^3, pre^4 : x2 > x3$ and ($x2 > 100$ or $x3 < 50$)
- $post^1 : y1 > x1 + 1$ or $y1 < x1$
- $post^2 : y2 > x1$ and $y3 < x1$ or $y3 > y2$
- $post^3 : y1 > x2$ and $y1 > x3$
- $post^4 : y2 > x2$ or $y3 > y2$

3.2 Path Selection

After all the nondeterministic processes are decomposed into deterministic subprocesses, the next important issue to address is how to define data flow paths for testing and how to select them in a CDFD. The classical Yourdon DFDs are not widely used for testing in industry since that these diagrams do not support the conditional controls and formal semantics explicitly, and therefore they are intricate for test paths selection. In SOFL, a CDFD is an extended data flow diagram holding control structures, as well formalized notations are used to define the processes in the CDFD. Thus independent data flow paths can be effectively extracted for testing by using these control structures and its module. In our approach, a data flow path is defined as a deterministic sequence of data flows with its processes that lead to output data flows of the CDFD from an input data flow sets. Formally, we have the following criteria for test path selection:

Criterion 1. Each data flow of a CDFD is covered by the set of test paths.
Criterion 2. Each subprocess is used at least in one path.

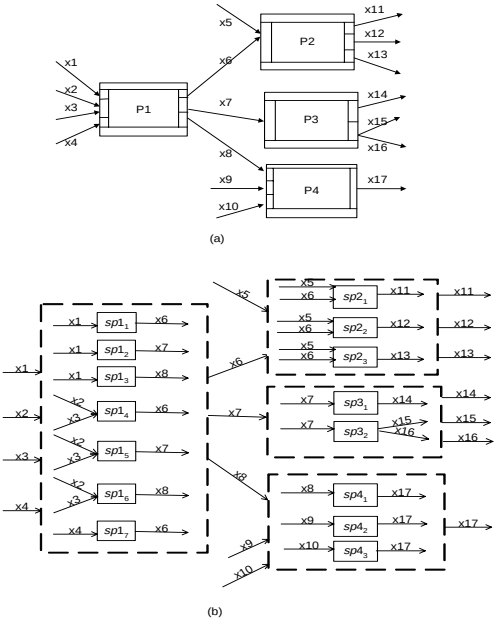


Fig. 4. A CDFD and its deterministic CDFD

Criterion 1 is a mutation of the general control structure testing, but considers more types of control structures for test path selection:

- Each data flow is transferred to the correct process/subprocess/control node;
- Every branch of a conditional node should be tested at least once;
- If a merging node is traversed, all of its input flows should be covered by the given test path;
- When a separating or a broadcasting node is added to a data flow path, the path may cover part of its output flows for testing if they are not affected by the others during the following execution;
- When a nondeterministic node is added to a test path, we select one of its output flows to traverse.

Criterion 2 is to ensure that every function in a process will be tested. Each function of a process is represented by a subprocess, and testing of the process needs to cover all of its subprocesses. For example, the nondeterministic processes **P**1, **P**2, **P**3 and **P**4 in Figure 4(a) are decomposed as follows (Figure 4(b)):

$$\mathbf{P1} \twoheadrightarrow sp1_1|sp1_2|sp1_3|sp1_4|sp1_5|sp1_6|sp1_7$$
$$\mathbf{P2} \twoheadrightarrow sp2_1|sp2_2|sp2_3$$
$$\mathbf{P3} \twoheadrightarrow sp3_1|sp3_2$$
$$\mathbf{P4} \twoheadrightarrow sp4_1|sp4_2|sp4_3$$

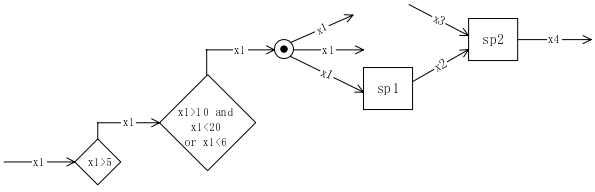


Fig. 5. A data flow path

In this CDFD, we identify twelve simple data flow paths, which cover all the subprocesses occurring in the diagram. These paths and the corresponding subprocesses covered are given as follows in the format: *path* → *subprocess set*.

1. $[x1, \{x5, x6\}, x11] \rightarrow \{sp1_1, sp2_1\}$
2. $[x1, \{x5, x6\}, x12] \rightarrow \{sp1_1, sp2_2\}$
3. $[x1, \{x5, x6\}, x13] \rightarrow \{sp1_1, sp2_3\}$
4. $[x1, x7, x14] \rightarrow \{sp1_2, sp3_1\}$
5. $[x1, x7, \{x15, x16\}] \rightarrow \{sp1_2, sp3_2\}$
6. $[x1, x8, x17] \rightarrow \{sp1_3, sp4_1\}$
7. $[\{x2, x3\}, \{x5, x6\}, x11] \rightarrow \{sp1_4, sp2_1\}$
8. $[\{x2, x3\}, x7, x14] \rightarrow \{sp1_5, sp3_1\}$
9. $[\{x2, x3\}, x8, x17] \rightarrow \{sp1_6, sp4_1\}$
10. $[x4, \{x5, x6\}, x12] \rightarrow \{sp1_7, sp2_2\}$
11. $[x9, x17] \rightarrow \{sp4_2\}$
12. $[x10, x17] \rightarrow \{sp4_3\}$

3.3 Test Cases Generation

Having selected paths to satisfy **Criterion 1** and **Criterion 2**, it is reasonable to generate test cases so that they will execute through every branch or subprocess in the CDFD.

The control nodes and the specifications of the subprocesses in a path provide a set of ordered constraints on the execution of the path by the selected test cases. The goal of test case generation is therefore to meet these constraints. The strategies for generating test cases to meet the specification include *boundary value analysis*, *domain testing*, and so on.

For example, the data flow path in Figure 5 covers subprocesses (sp1,sp2), as well two conditional nodes and a broadcasting node. Suppose sp1 and sp2 are defined as follows:

subprocess sp1(x1: **real**)x2: **real**	**subprocess** sp2(x2, x3: **real**)x4: **real**
pre TRUE	**pre** x3 > x2 **and** x3 < x2 + 10
post x2 = x1 + 2	**post** x4 = x3 + x2
end	**end**

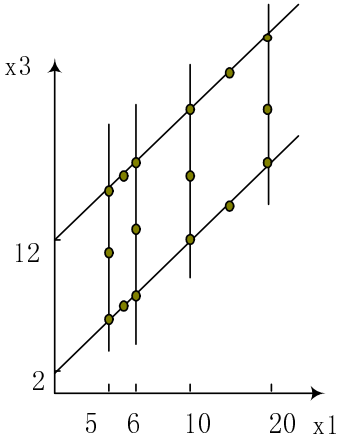


Fig. 6. The test points of T1

The set of exterior inputs in Figure 5 is {x1, x3}, and its constraints, $T1$, can be computed by using conditional constraints, precondition and postcondition of subprocesses, as:

$x1 > 5$ and ($x1 > 10$ and $x1 < 20$ or $x1 < 6$) and ($x3 > x1+2$ and $x3 < x1+12$)

The domain of the data flow path is illustrated in Figure 6, and we select the following points as test cases

$$
\begin{aligned}
(x1, x3) = \{ & \\
& (5.001, 7.001), \quad (5.001, 16.999), \quad (5.999, 8.001), \quad (5.999, 17.999), \\
& (5.001, 12.001), \quad (5.999, 12.999), \quad (5.500, 7.501), \quad (5.500, 17.499), \\
& (10.001, 17.001), (15.000, 17.001), (19.999, 26.999), (15.000, 26.999), \\
& (10.001, 12.001), (10.001, 21.999), (19.999, 22.001), (19.999, 31.999), \\
& \}.
\end{aligned}
$$

To enhance the usability of this approach, we are working on a software tool to support the automation of selecting data flow paths and generating test cases in accordance with the path coverage criterion. Since it is still a primitive prototype, we will report it in our future publication after it is completed properly.

4 Case Study

In order to evaluate the effectiveness of the suggested testing approach, we applied our approach to testing part of an ATM (Automated Teller Machine) system implemented based on a formal specification written using the SOFL specification language by the second author [16]. The ATM system includes 5 basic

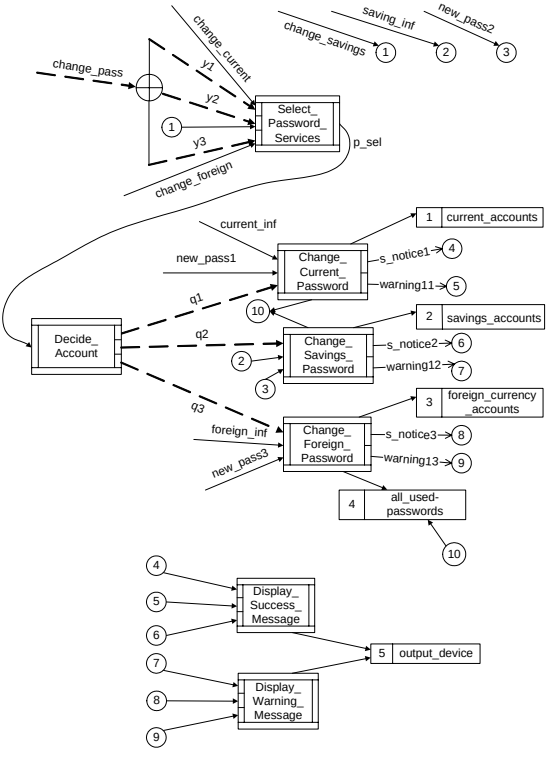


Fig. 7. The CDFD of Change_Password_Decom module

functional services: (1) operations on current account, (2) operations on saving account, (3) transfer money between accounts, (4) manage foreign currency account, and (5) change password. The entire specification contains sixty-nine pages of descriptions. For the sake of space, we choose only one of its modules, *Change_Password_Decom*, for the case study.

The *Change_Password_Decom* module includes the three primary functions:

1. Change password for the current account;
2. Change password for the savings account;
3. Change password for the foreign currency account.

The CDFD of the *Change_Password_Decom* module is shown in Figure 7. In this module, process *Select_Password_Services* receives a command for changing a password of the current account, savings account, or foreign currency account, and passes it to process *Decide_Account*. Process *Decide_Account* then determines to send a *control data flow* to processes *Change_Current_Password*, *Change_Savings_Password*, or *Change_Foreign_Password*. If the new password provided is satisfactory according to the criterion, one of these three processes will send a successful message to process *Display_Success_Message* for display-

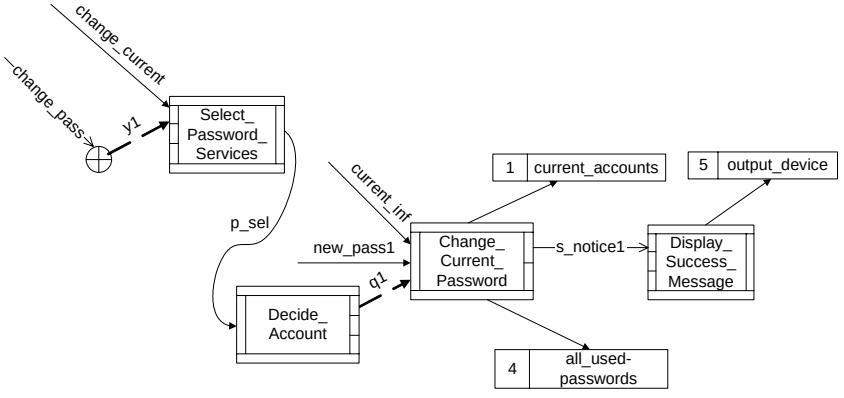


Fig. 8. A path of the CDFD of Change_Password_Decom

ing on an output device; otherwise, a warning message will be sent to process *Display_Warning_Message* for displaying.

4.1 Generating Test Cases

We generate test cases before the implementation of the specification. The module includes the following processes: *Select_Password_Services, Decide_Account, Change_Current_Password, Change_Savings_Password, Change_Foreign_Password, Display_Success_Message,* and *Display_Warning_Message.* Some of the processes in the *Change_Password_Decom* module take input data flows or produce output data flows nondeterministically (e.g., *Select_Password_Services*). The tester decomposes them into eighteen equivalent deterministic subprocesses.

When generating test cases for integration testing, we extract six paths from the original CDFD by using the data flow path coverage strategy. Each of these paths defines how outputs are generated from the related inputs. One of the paths representing the successful change of current password is shown in Figure 8. The inputs and external variables consumed by processes are {change_pass, change_current, new_pass1, current_inf, current_accounts, all_used_passwords, output_device}. The constraints enforced by the path are as follows:

```
(bound(change_current) and p_sel = <1>) and
(p_sel = <1> and bound(q1))             and
(let old_pass = current_inf.pass in
      if current_inf inset dom(current_accounts)
      then
          current_accounts = override(domrb({current_inf}, ~current_accounts),
                {modify(current_inf, pass --> new_pass1) -->
              ~current_accounts(current_inf) }  ) and
           all_used_passwords = diff(~all_used_passwords, {old_pass}) and
           s_notice1 = "successful")    and
(bound(s_notice1) and output_device = conc(~output_device, [s_notice1]))
```

We generate twelve test cases for that cover all the possible paths, but for the sake of space, we present only some of the test cases below. Since the Current_accounts is an external file that may consist of millions of account records and output_device represents a physical output device, we omit the concrete values of those data structures for brevity.

1 change_pass=<!>, change_current=<!>,
current_inf=<1000000,00000>, new_pass1=<00001>,
current_accounts_file=<..\Records\currents1.dat>,
all_used_passwords=<00000,00001, ...>, output_device=<...>;
2 change_pass=<!>, change_current=<!>,
current_inf=<1000000,00000>, new_pass1=<33333>,
current_accounts_file=<..\Records\currents1.dat>,
all_used_passwords=<...>, output_device=<...>;
3 change_pass=<!>, change_current=<!>,
current_inf=<1000000,00000>, new_pass1=<00000>,
current_accounts_file=<..\Records\currents1.dat>,
all_used_passwords=<...>, output_device=<...>;
4 change_pass=<!>, change_current=<!>,
current_inf=<1000000,00000>, new_pass1=<99999>,
output_device=<...>;
...

4.2 Test Results Analysis

Test cases generated from a formal specification need to be translated into a form suitable for being used to execute the program that implements the specification in a specific programming language. In this case study, we implement the program in C++ and a third party deliberately insert twenty nine errors in it, most of which are errors on the condition predicates and the domain errors of each process. Table 1 summarizes the results of the testing. The error detection rate indicates that 73% of the errors inserted independently are found. In addition, the tester detects forty three errors of the original program by this test.

Table 1. Testing results analysis

process	inserted errors	inserted errors found	errors detection rates	original errors found
Select_Password_Services	6	4	67%	3
Decide_Account	3	3	100%	1
Change_Current_Password	7	2	28%	8
Change_Savings_Password	3	2	67%	8
Change_Foreign_Password	2	2	100%	8
Display_ Success_Message	4	4	100%	8
Display_Warning_Message	4	4	100%	9
Amount	29	21	73%	43

The results of our case study demonstrates that our proposed testing approach and strategies are relatively effective in detecting predicate violations

and domain errors, while they are less effective in detecting the errors violating invariants defined in the specification.

5 Related Work

There exist many methods for testing programs from the data flow point of view [17][18], but almost none of them utilizes data flow information from the specifications of the programs. Rather, they derive data flow information from the control flow structures of programs, and therefore they are still techniques for structural testing. Since the final goal of testing is to ensure that programs satisfy their specifications, functional testing based on specifications is extremely important [19].

Since only a few of formal specification languages (e.g., SOFL [15] and FOCUS [20]) and semi-formal specification languages (e.g., the activity diagrams in UML) are designed on the basis of data flow notion, most of the research on specification-based testing so far are focused on the use of control flow specifications. Aynur Abdurazik and Jeff Offutt provided a set of strategies and a Rose-based test data generation tool from state-based specifications, which is applicable to the activity diagrams in the UML [21]. F. Basanieri developed a Cow_Suite tool in which the derivation of test cases was based on the software analysis and design document, and used the UML-based original test methodology UIT (Use Interaction Test) [22][23]. Chris Rudram extended the syntax and semantics of Activity Diagrams with Formal Activity Diagrams (FAD) to show user interaction with system, and then divided a FAD into segments for testing [24]. However, the Activity Diagrams provide the whole structure of a system with the collaborations of the other diagrams (e.g., Use Case Diagrams), and we need to consider the infections of these factors for testing. Furthermore, the diagrams in the UML do not describe the precise semantics of a system, and therefore the corresponding test cases are less rigorously generated. The activity diagrams in UML do not support nondeterministic mechanism useful for modeling of systems either.

6 Conclusions and Future Work

In this paper we have described an approach to integration testing of programs based on their data flow specifications. The essential idea of this approach is to test programs using test cases generated based on the data flow path coverage in their specifications. We have provided two criteria for test case generation and presented a case study of testing part of an ATM system written in SOFL to evaluate the effectiveness of the suggested testing approach. The case study result shows that the approach is effective in detecting faults leading to predicate and domain violations.

As future work, we plan to establish more powerful strategies and coverage criteria to discover greater classes of errors. In addition, we will try to address

some difficult issues concerned with program testing based on data flow specifications, such as invariants checking, recursive functions, and data flow loops. We are also interested in investigating techniques for automatic test case generation and in constructing an effective tool to enhance the degree of automation.

References

1. McDermid, J., ed.: Software Engineering's Reference Book. Butterworth-Heinemann Ltd, Linacre House, Jordan Hill, Oxford OX2 8DP (1991)
2. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall, 2nd edition (2002)
3. Hoffman, D.M., Strooper, P.: Automated module testing in prolog. IEEE Transactions on Software Engineering **17** (1991) 934–943
4. Chang, J., Richardson, D.J., Sankar, S.: Structural speicifcation-based testing with ADL. In: Proceedings of the 1996 International Symposium on Software Testing and Analysis. (1996)
5. Stocks, P., Carrington, D.: A framework for specification-based testing. IEEE Transactions on Software Engineering **22** (1996) 777–793
6. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys (CSUR) **29** (1997) 366–427
7. Stocks, P.: Applying Formal Methods to Software Testing. PhD thesis, the Department of Computer Science, the University of Queensland (1993)
8. Offutt, J., Liu, S.: Generating Test Data from SOFL Specifications. Journal of Systems and Software **49** (1999) 49–62
9. Chow, T.: Testing software designs modeled by finite-state machines. IEEE Transactions on Software Engineering **4** (1978) 178–187
10. Fujiwara, S., Bochman, G., Khendek, F., Amalou, M., Ghedasmi, A.: Test selection based on finite state models. IEEE Transactions on Softare Engineering **17** (1991) 591–603
11. Turner, C.D., Robson, D.J.: The state-based testing of object-oriented programs. In: Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM-93), Montreal, Quebec, Canada (1993)
12. Gargantini, A., Riccobene, E.: ASM-based testing: Coverage criteria and automatic test sequence generation. Journal of Universal Computer Science (2001)
13. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Software Testing, Verification and Reliability **13** (2003) 25–53
14. Liu, S., Offutt, A.J., Ho-Stuart, C., Sun, Y., Ohba, M.: SOFL: a formal engineering methodology for industrial applications. IEEE Transactions on Software Engineering **24** (1998) 337–344 Special Issue on Formal Methods.
15. Liu, S.: Formal Engineering for Industrial Software Development using the SOFL Method. Springer-Verlag (2004)
16. Liu, S.: A case study of modeling an ATM using SOFL. Technical report HCIS-2003-01, http://cis.k.hosei.ac.jp/tr/, Faculty of Computer and Information Sciences, Hosei University, Tokyo, Japan (2003)
17. Beizer, B.: Software Testing Techniques. second edn. Van Nostrand Reinhold, New York (1990)
18. C.Jorgensen, P.: Software Testing: A Craftsman's Approach. second edn. CRC Press LLC, 2000 NW Corporate Boulevard, Boca Raton, Florida 33431, USA (2002)

19. Beizer, B.: Black-Box Testing. Wiley (1995)
20. Broy, M., Stolen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer-Verlag (2001)
21. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: Proceedings of the Second International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, Springer-Verlag, LNCS (1999) 416–429
22. F.B., et al: An automated test strategy based on UML diagrams. In: Ericsson Rational User Conference, Upplands Vasby Sweden (2001)
23. Basanieri, F., Bertolino, A., Marchetti, E.: The cow suite approach to planning and deriving test suites in UML projects. In: Proc. Fifth International Conference on the Unified Modeling Language - the Language and its Applications UML 2002, LNCS 2460, Dresden, Germany (2002) 383–397
24. Rudram, C.: O poGenerating test cases from UML. (2000)

Combining Algebraic and Model-Based Test Case Generation

Li Dan[1,⋆] and Bernhard K. Aichernig[2]

[1] Guizhou Academy of Sciences, Guiyang,
Guizhou, 550001, China
lidan_gz@163.com
[2] The United Nations University,
International Institute for Software Technology,
P.O. Box 3058, Macau
bka@iist.unu.edu

Abstract. The classical work on test case generation and formal methods focuses either on algebraic or model-based specifications. In this paper we propose an approach to derive test cases in the RAISE method whose specification language RSL combines the model-based and algebraic style. Our approach integrates the testing techniques of algebraic specifications and model-based specifications. In this testing strategy, first, every function definition is partitioned by Disjunctive Normal Form (DNF) rewriting and then test arguments are generated. Next, sequences of function calls are formed. Finally, the test cases are built by replacing the variables, on both sides of the axioms, with the sequences of functions calls. These kinds of test cases not only provide the data for testing, but also serve as test oracles. Based on this combined approach, a test case generation tool has been developed.

Keywords: Test case generation, RAISE, RSL, formal method.

1 Introduction

Designing test cases is difficult, expensive and tedious. Recently, formal specifications have played an important role in test case generation for black-box software testing. Specification-based testing is concerned with deriving testing information from a specification, rather than from source code. The formal specification of the test object forms the basis for a systematic selection of test data, the sequencing of test cases, and the evaluation of the test results.

Currently, a lot of research is going on in the area of specification-based software testing. For model-based formal specification languages, such as VDM, Z and B, the work focuses on partition techniques that generate Disjunctive

⋆ The work is partly supported by research grant (#3010) of Science and Technology Foundation of Guizhou Province, 2001.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 250–264, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Normal Form (DNF) and how to construct a Finite State Model (FSM) from a DNF [10, 15, 5, 3]. On the other hand, for algebraic specification languages, like Larch and CASL, people pay more attention to the techniques of applying term rewriting rules and how to judge the observational equivalence of abstract objects [14, 8, 11]. There is a clear distinction between methods used for model-based specifications and algebraic specifications.

The Raise Specification Language (RSL) [16] is a language suitable for formal specification and development of software systems. RSL is a "wide spectrum" language. This means that it has features allowing its use for very abstract, initial specifications and also for more concrete developments of an initial specification that can be easily (or even automatically) translated into a programming language. RSL can be taken as a combination of model-based and algebraic specification languages. The ability to mix different styles within the same module is very helpful in test case generation.

In this paper, we propose an approach to derive test cases from an RSL specification that has both a model-based part and an algebraic part. The approach is based on the ideas that we choose some techniques which are suitable for RSL and easy to implement from both model-based and algebraic testing techniques, and combine them together. In this approach, roughly speaking, first partition analysis is applied to functions and test calls corresponding to sub-domains are extracted, then terms are constructed through iterative invocations of the test calls, and test cases are generated by instantiating the variables in the left-hand side and right-hand side of an axiom. The test cases can be executed and the results are compared to check whether the test passes. A tool has been developed to derive test cases from RSL specifications using this approach.

In Section 2 of this paper, we give the basic concepts used in our work. Section 3 describes the approach that derives test cases from RSL specifications. Section 4 is devoted to compare our work with other related work. The tool is briefly described in Section 5. In Section 6, we conclude our work and make suggestions for future work.

2 Basic Concepts

The RAISE Specification Language (RSL) is a modular language. Specifications are in general collections of related modules. A scheme is the basic unit for constructing modules through class expressions. In applicative style, a scheme module S is defined as a triple:

$$S = \langle \mathcal{T}, \mathcal{V}, \mathcal{A} \rangle$$

where $\mathcal{T}$ is the *type declaration* and $\mathcal{T} = (T_1, \ldots, T_n)$, T_i $(0 \leq i \leq n)$ is called a *type definition*. $\mathcal{V}$ is the *Value declaration* and $\mathcal{V} = (V_1, \ldots, V_m)$, V_i $(0 \leq i \leq m)$ is a *value definition*. $\mathcal{A}$ is the *axioms declaration* and $\mathcal{A} = (A_1, \ldots, A_l)$. We say A_i $(0 \leq i \leq l)$ is an *axiom* of the module.

The specification in Figure 1 is an example of RSL module that describes a "first in – first out" queue buffer with input and output characters (represented

```
scheme buffer =
  class
    type Buff = Int*

    value
      empty : Buff = ⟨⟩,

      input : Int × Buff → Buff
      input(i, b) ≡
        if i ≥ 32 ∧ i ≤ 126 then
          if i ≥ 97 ∧ i ≤ 122 then
            let j = i − 32 in b ⌢ ⟨j⟩ end
          else b ⌢ ⟨i⟩ end
        else b end,

      output : Buff → Int × Buff
      output(b) ≡ case b of
                    empty → (0, empty),
                    ⟨h⟩ ⌢ t → (h, t)
                  end,

      count : Buff → Int
      count(b) ≡ len (b),

      has_value : Buff → Bool
      has_value(m) ≡ m ≠ empty,

      first_value : Buff → Int
      first_value(b) ≡ case b of
                         empty → 0,
                         ⟨h⟩ ⌢ t → h
                       end,

      last_value : Buff → Int
      last_value(b) ≡ if len(b) > 1 then last_value(tl b)
                      else first_value(b) end
    axiom
      [a1]
        ∀ i : Int, b : Buff •
          count(let (j, m) = output(input(i, b)) in m end) ≤
            count(b),
      [a2]
        ∀ i : Int, b : Buff •
          input(i, b) ≡ b pre i < 32 ∨ i > 126
  end
```

Fig. 1. RSL specification of an input-output buffer for ASCII characters

by their ASCII values). The buffer accepts valid characters (with ASCII values between 32 to 126), changes small case letters to upper case letters (transfer ASCII values 97–122 to 65–90, respectively) and delivers the characters stored in it when required. This module is a model-based specification which is developed from an algebraic specification but some axioms remained in the module.

In RSL, there are three kinds of types [16]: *built-in data types*, *abstract data types* and *compound data types*. In a RSL module, there may be many type definitions, but only few of them are really modeling the system state, these types can be referred as *type of interest* [12]. In this paper, we call the type of interest a *major type* which represents the state of a module. State information should be minimal in a module. Thus, there should be only one major type in a well designed RSL module [12].

The following is the definition of *major type*:

Definition 1. *Suppose* $\mathcal{T} = \{T_1, \ldots, T_n\}$ *are type definitions of module* S*. A type definition* T_k *depends on type definition* T_l *if* T_l *appears in the type expression of* T_k*, where* $1 \leq k$*,* $l \leq n$*. A type definition* T_i $(1 \leq i \leq n)$ *is called a* **major type** *of* S *if and only if none of the type definitions in* $\mathcal{T}$ *depend on* T_i*. The major type is denoted as* T_m*.*

In RSL, there are two kinds of value definitions: *function* and *constant*. For the purpose of this paper, we take *constant* as a special kind of function which has no input variable and only one output variable. So we consider all value definitions as functions. RSL allows the user to describe functions either implicity, in terms of their interfaces, or explicity, in terms of the details of their operations.

We define a function $V \in \mathcal{V}$ as:

$$V = \langle \mathcal{I}, \mathcal{O}, \mathcal{P}, \mathcal{R} \rangle$$

where $\mathcal{I}$ are input parameters and $\mathcal{I} = (i_1 : I_1, \ldots, i_n : I_n)$. Each input parameter here is an input variable i_i of type I_i where $I_i \in \mathcal{T}$. $\mathcal{O}$ are output parameters and $\mathcal{O} = (o_1 : O_1, \ldots, o_m : O_m)$. Each output parameter here is an output variable o_j of type O_j where $O_j \in \mathcal{T}$. $\mathcal{I}$ and $\mathcal{O}$ represent the interface of the function V. $\mathcal{P}$ is a value expression which describe the *precondition* of the function. $\mathcal{R}$ is the body part of the function in explicit style, or the *postcondition* of the function in implicit style.

The concept of dividing operations into different categories is an important idea in the testing of algebraic specifications [7, 8]. Inspired by [4], we give the definition of *creator*, *modifier* and *observer* which are used to classify the functions in RSL modules.

Definition 2. *Suppose* T_m *is a major type of an RSL module. A function* V *is called a* **creator** *of* T_m *if* T_m *appears in the output parameters and does not appear in the input parameters of* V*. If* T_m *appears in both input parameters and output parameters of* V*,* V *is called a* **modifier** *of type* T_m*. A function* V *is called an* **observer** *of* T_m *if* T_m *appears in the input parameters and does not appear in the output parameters of* V*.*

We use V^c to represent a *creator*, V^m to represent a *modifier*, V^o to represent an *observer*.

In the example shown in Figure 1, for instance, the function *empty* is a creator; *input*, *output* are modifiers; and the functions *count*, *has_value*, *first_value* and *last_value* are observers.

The creator functions give us a way to build a new sample of type T_m and modifiers provide the methods to handle the type. If T_m is an abstract type, observers provide the only way for us to query the content of T_m.

If there is only one major type T_m in the module, we can call the creators of type T_m the creators of the module. So for modifiers and observers.

The word *term* has been used to define a sequence of operations in algebraic specification languages. In the following part, we will give the definition of a *constructive term* for this paper.

Definition 3. *A sequence of function invocations, starting from the invocation of a creator, followed by iterative invocations of modifiers through replacing the input variables of major type T_m with the previous invocation sequence, is called a* **constructive term** *of type T_m. The* **length of a constructive term** *is the number of functions invoked in the term. A constructive term is called a* **ground constructive term** *if all variables in the term are instantiated with concrete values.*

In RSL, a function may return more than one result. For the purpose of constructing terms, we have to invoke a function in a special way that project it to return only single value. In here, the value is of type T_m. For example, an invocation of function *output* shown in Figure 1 that only returns one value of type *Buff* can be expressed as:

$$\textbf{Let } (i, b_o) = \text{output}(b_i) \textbf{ in } b_o \textbf{ end}$$

A constructive term can be referred as a function with output of type T_m. Let $\pi(V_i)$ represent the invocation of function V_i involving a projection to a single return value. Then the general form of a constructive term is as follows:

$$cons = \pi(V_1^m(\ldots, \pi(V_2^m(\ldots, \ldots, \pi(V_n^m(\ldots, \pi(V_j^c)))))))$$

Definition 4. *An invocation of an observer function, through instantiating the input variables of major type T_m with a constructive term, is called an* **observable term**. *The* **length of an observable term** *is the number of functions invoked in the term. An observable term is called a* **ground observable term** *if all variables in the term are instantiated by concrete values.*

An observable term can be referred as a function with one return value whose type is other than type T_m. So the observable term *obs* will take the form :

$$obs = \pi(V_k^o(\ldots, \pi(V_1^m(\ldots, \pi(V_2^m(\ldots, \ldots, \pi(V_n^m(\ldots, \pi(V_j^c)))))))))$$

An axiom itself is a Boolean expression which, by definition, must evaluate to **true**. We denote an axiom as follows:

$$A = \langle U_l, Op, U_r, Pre \rangle$$

where U_l and U_r are value expressions representing the left-hand side and right-hand side of the axiom definition. Pre is the precondition of the axiom that is also a value expression. Op is an operator which defines the way how to compare U_l with U_r.

A value expression can be evaluated (or, synonymously, executed) to return a value of definite type. The type of return value for Pre in above axiom must be Boolean. If it is *true*, the precondition is satisfied and the evaluation of value expressions U_l and U_r will be conducted. The return values, they are of the same data type, are compared using Op to check whether the axiom holds. If the type of U_l and U_r is primitive, like built-in types, the comparison could be easily carried out. But for abstract data types, there are no straightforward methods to judge the relationship of two values automatically.

In our case, if the type of U_l and U_r is a major type T_m and the operator Op is equivalence, then we can apply an invocation of observer V^o to this axiom A to generate a variation of the axiom:

$$A^m = \langle \pi(V^o(U_l)), Op, \pi(V^o(U_r)), Pre \rangle$$

Because the data type of return value for $\pi(V^o(U_l))$ and $\pi(V^o(U_r)$ is no longer the abstract data type T_m, it is possible for us to check whether the axiom holds. This idea gives us the inspiration in the construction of test cases.

Consider the axiom a_2 in Figure 1. Since the right-hand side of a_2 is of major type, in order to derive test cases from this axiom, an observer, say *last_value*, is applied to a_2, transforming axiom a_2 into a variation a_2'

axiom
[a_2']
∀ i : **Int**, b : Buff •
last_value(input(i,b)) ≡ last_value(b)
pre i < 32 ∨ i > 126

3 Test Case Generation

In this section we present our approach to generate test cases by creating test calls, sequencing them, and solving the oracle problem. For simplicity, we require that the input specification should be well-formed, only one major type which is considered as abstract data type exist in the module and there is at least one creator, one modifier, and one observer in the module.

The whole process can be broken down into the following steps:

1. Analyse the specification, decide the major type and divide the functions into three categories: creator, modifier, observer. This step is easy to be conducted according to Definition 1 and Definition 2.

2. For each function, apply partition analysis and divide the input domain into subdomains. On each subdomain, instantiate the input variables except the major type's. Thus, we get a set of test calls for this function.
3. Decide an integer k and produce a collection of constructive terms and observable terms whose length is less than or equal to k by constructing from the test calls of different functions.
4. Use axioms as test oracle by replacing the variables of the axiom itself or a variation of the axiom with constructive terms and observable terms and execute the left-hand side and right-hand side of the axiom separately. The two results will be compared to decide whether the test passes or not.

Steps 2–4 will now be described in detail.

3.1 Partition Analysis

For a function $V = \langle \mathcal{I}, \mathcal{O}, \mathcal{P}, \mathcal{R} \rangle$, we collect the two parts of precondition and function body (or postcondition) together to form a large value expression $E = \mathcal{P} \wedge \mathcal{R}$. Then partition analysis is carried out on E.

We adopt the well-know strategy proposed by [6] to partition the expressions into their DNF. In [13], Meudec gives many coarse partitioning rules and refinement rules for VDM-SL specification. Most of these rules and methods are used in our work after amendment and extension.

After applying partition analysis, a partition $P = \{D_1, \ldots, D_n\}$ for function V is produced. Each subdomain D_i is represented by a set of predicates that give the constraint over input variables. We use $C_{D_i} = p_1 \wedge \ldots \wedge p_m$, where p_j $(1 \leq j \leq m)$ is a predicate, to denote this constraint.

Consider a finite list of values $DT = \{i_1, \ldots, i_n\}$ where $n \geq 1$, with respective types $T = (T_1, \ldots, T_n)$ on subdomains D_i, while DT satisfies the constraint C_{D_i}. We say that DT is a solution to constraint C_{D_i} and $V(DT)$ is a test call of V. DT is also referred as a *test data* of V. If no DT can be found for C_{D_i}, then we say subdomain D_i is infeasible for V.

The job of instantiating at this stage is only for the input variables which are not of major type. As an abstract data type, the major type could not be instantiated directly in most cases. The approach to instantiate a major type will be discussed in the next step.

A simple "generate and test" strategy, where a solution candidate is first generated, then tested against the constraint for consistency, is adopted in our work, because the predicates are quite simple in most situations. The constraint $C_{D_i} = p_1 \wedge \ldots \wedge p_m$ is first reordered by the number of variables contained in each predicate p_j so that the predicate with less variables will appear in front of the constraint. Then, the variables in predicate p_1 will be instantiated to ensure p_1 holds. The variables values instantiated in prior predicate will be checked whether they meet the requirement of the current predicate, if not, or there are no values for the variables of this predicate, new values will be calculated for variables. This process continues until the last predicate has been coped. Then, the set of obtained values will be evaluated against the whole constraint.

If the evaluation is not successful, all values will be abandoned and the whole process starts again. This procedure repeats until a set of values which satisfy the constraint is found or the number of repetition exceeds a given limit. In this case, it is assumed that this constraint is infeasible. As a future extension the connection with a constraint solver as in [3] is envisaged.

Faults may either affect the behavior within a subdomain (computation fault) or affect the boundaries of the subdomains (domain faults) [9]. Computation faults are detected by choosing one or more test calls from each subdomain. Domain faults are detected by testing around subdomain boundaries. So in our approach, if possible, we construct several test calls from each subdomain D_i, one or more are around the boundaries and one is within the subdomain by random selection.

For example, if we conduct partition analysis on function *input* in Figure 1, 7 subdomains will be generated. After resolving the constraints using our strategy and instantiate values for each subdomain, we get 13 test calls from 5 feasible subdomains. Note that input variable b is of major type and will be handled at the next step.

$$\left\{\begin{array}{c} i \geq 97 \ \wedge \ i \leq 122 \\ i < 97 \ \wedge \ i > 122 \\ i \geq 32 \ \wedge \ i < 97 \\ i > 122 \ \wedge \ i \leq 126 \\ i < 32 \wedge i > 126 \\ i < 32 \\ i > 126 \end{array}\right\} \Rightarrow \left\{\begin{array}{ccc} \text{input(97,b);} & \text{input(101,b);} & \text{input(122,b)} \\ & \varnothing & \\ \text{input(32,b);} & \text{input(49,b);} & \text{input(96,b)} \\ \text{input(123,b);} & \text{input(125,b);} & \text{input(126,b)} \\ & \varnothing & \\ & \text{input(31,b);} \quad \text{input(-1032,b)} & \\ & \text{input(127,b);} \quad \text{input(1782,b)} & \end{array}\right\}.$$

3.2 Term Construction

Test calls for every function are now available. Input variables in each test call have been replaced by concrete values except the ones of major type. Following the categories of their functions, test calls can also be divided into three groups : *creator*, *modifier* and *observer*.

The idea of using a sequence of invocations of functions, so called terms, rather than test cases for individual functions, in the testing process was discussed in many papers [7, 4, 8]. An invocation to function V that only returns a single value of major type T_m is denoted by $\pi^m(V)$. The invocation to function V that returns a single value other than of major type is denoted by $\pi^o(V)$.

Let $\{\pi^m(V_i^c)\}, \{\pi^m(V_j^m)\}, \{\pi^o(V_l^o)\}$ represent sets of invocations of test calls whose functions correspond to creator, modifier and observer, respectively. In order to build constructive terms, we start from a $\pi^m(V_i^c)$ to construct the term, then we select a $\pi^m(V_j^m)$ and replace the input variables of major type with the prior built term. This process continues until the length of the term reaches the positive integer k. There are no un-instantiated variables, either of a major type or of other data types, existing in the terms. So these constructive terms are also ground constructive terms. Selecting a $\pi^o(V_l^o)$ and replacing the input variables of major type with a constructive term, an observable term is

generated. For observable terms, the situation is the same, they are also ground observable terms.

For the purpose of illustration suppose the positive integer $k = 4$. From the set of test calls built in the above step, a group of ground constructive terms with length less than or equal to k is induced through replacing the input variables of major type with the prior term.

1. $empty$
2. $input(101, empty)$
3. $input(49, input(101, empty))$
4. **Let** $(i_o, b_o) = \text{output}(input(49, input(101, empty)))$ **in** b_o **end**

In the same way, a group of ground observable terms and their data types is shown as :

1. $has_value(empty)$: **Bool**
2. $count(input(101, empty))$: **Int**
3. $first_value(input(49, input(101, empty)))$: **Int**

3.3 Build Test Case

Let $\{CT_i\}$ $(1 \leq i \leq n)$ be the set of constructive terms, and $\{OT_j\}$ $(1 \leq j \leq m)$ be the set of observable terms, where $n, m \geq 1$. An axiom in the specification is denoted as $A = \langle U_l, Op, U_r, Pre \rangle$. We derive test cases from the axiom using the algorithm shown in Figure 2.

In this algorithm, it is clear that if the return value of U_l and U_r of A is of the major type T_m and the Op is equivalence $(\equiv)$, a variation of axiom $A^m = \langle \pi(V^o(U_l)), Op, \pi(V^o(U_r)), Pre \rangle$, instead of A itself, is actually used in the process of generating test cases.

After the last step, a finite set of test cases is generated from the specification. Each test case is denoted as a tuple $Tc = \langle Tc_l, Op, Tc_r, Pre \rangle$ which includes two value expressions Tc_l, Tc_r and their relationship Op under precondition Pre. The test cases not only provide the test data to be executed, but also serve as test oracles.

For instance, consider to derive a test case from axiom a_2 of the specification given in Figure 1. It should be noted that the return value of left-hand side or right-hand side of the axiom is of major type, thus, a variation of the axiom, a_2', which can be found at the end of Section 2, is used in deriving test cases.

Note that axiom a_2' has a precondition $(i < 32 \vee i > 126)$ that contains a variable i which still is not instantiated. It is quite easy to select an integer i, for instance $i = 17$, which satisfies the condition $(i < 32 \vee i > 126)$. By replacing variable b in the axiom a_2' with a constructive term: $input(49, input(101, empty))$, we can generate a test case from the axiom:

$$
\begin{aligned}
Tc = \langle\ & \text{last_value}(\text{input}(17, input(49, input(101, empty)))), \\
& =, \\
& \text{last_value}(input(49, input(101, empty))), \\
& 17 < 32 \vee 17 > 126\ \rangle
\end{aligned}
$$

To apply the test case $Tc = \langle Tc_l, Op, Tc_r, Pre \rangle$ to an implementation, we should map each function in Tc_l, Tc_r and Pre to a method in the implementation program. As a result, this mapping generates three method sequences M_l, M_r and M_{pre} in the program corresponding to Tc_l, Tc_r and Pre, respectively. For a complete implementation, this mapping exists and can be indicated manually by the implementation designer or can be derived automatically from the specification. After executing M_l, M_r and M_{pre} in the program and obtaining results O_l, O_r and O_{pre}, if O_{pre} is true, we compare O_l with O_r with relation Op. If it is satisfied, the test is passed, otherwise a implementation error is revealed by test case $Tc = \langle Tc_l, Op, Tc_r, Pre \rangle$.

GenerateTestCase
Begin
  Select axiom $A = \langle U_l, Op, U_r, Pre \rangle \in \mathcal{A}$
  $Tc_l := U_l$; $Tc_r := U_r$; $Pre := Pre$
  Select $CT_i \in \{CT_i\}$
  For each variable of major type in Tc_l, Tc_r, Pre
    $Tc_l := Tc_l$ replace the variable with CT_i
    $Tc_r := Tc_r$ replace the variable with CT_i
    $Pre := Pre$ replace the variable with CT_i
  End For
  For each variable of other data types in Tc_l, Tc_r, Pre
    If type of the variable is T
      Select $OT_j \in \{OT_j\}$ of data type T
      $Tc_l := Tc_l$ replace the variable with OT_i
      $Tc_r := Tc_r$ replace the variable with OT_i
      $Pre := Pre$ replace the variable with OT_i
    End If
  End For
  If result of Tc_l is of major type **and** Op is $\equiv$
    Select $\pi^o(V_k^o) \in \{\pi^o(V_k^o)\}$
    $Tc_l := \pi^o(V_k^o)$ replace major type input variable with Tc_l
    $Tc_r := \pi^o(V_k^o)$ replace major type input variable with Tc_r
  End If
  Return $T_c = \langle Tc_l, Op, Tc_r, Pre \rangle$
End

Fig. 2. Algorithm to derive test cases from axioms

3.4 Discussion

We have described every step of our approach to derive test cases from RSL specification and use these test cases as test oracles. However, there are still things remain unclear : determine the positive integer k and which individual function test calls should be selected in constructing terms. These are equal to the problems of how many test cases should be created for the testing process and how the coverage of the test cases should be taken.

Practically, we adopt the following criteria:

- The above positive integer k may be determined by first asking the user give an acceptable value according to his understanding that major type implemented in the program, such as the maximum sizes of arrays, or the boundary values of variables. If the user does not wish to give the value, the k will be default set to the number of functions in the specification plus 1. This default k provides the possibility that every function could appear in longest constructive terms;
- Each test call of creator or modifier should appear in the constructive terms at least once. Each test call of observers should appear in the observable terms at least once;
- Each constructive term and observable term should be tested at least once;
- Each axiom should be used to generate test cases at least once if the axiom is capable to derive test cases.

4 Related Work

In the world of formal specification languages, model-based and algebraic specifications are frequently regarded as separate and so are their testing techniques. For testing of model-based specifications, as in VDM and Z, partition analysis is a standard method that the work of Dick and Faivre [6] was a major contribution to. This method has been adapted and ameliorated in many other research works [13, 2]. There are two major steps in the method.

1. Partition an operation's input domain by reducing the input expression to disjunctive normal form (DNF) and then derive test case from each DNF.
2. Construct a Finite Sate Automaton (FSA) from the specification by analysis of the state space and operations. The FSA can be used to sequence the test cases.

For partition analysis, it is quite difficult to solve the problem of test sequencing because of the characteristics of model-based specification. In the Dick and Faivre paper, only generation of DNF (Step 1) was automated. Currently, there is still no proposal which automates the full process [3]. At Step 2, there is possible state explosion and the non-discovery problem which makes it difficult to determine all the FSA states and transitions.

On the other hand, there are also many works on deriving test cases from algebraic specifications. Testing code against algebraic specification consists of showing that the final system satisfies the axioms in the specification[8].

In the ASTOOT approach [7], the concept of equivalent terms which are valid sequences of invocations of operations has been presented and the idea of using pairs of equivalent terms, rather than individual operations, as test cases was adopted. The equivalent terms are generated by term rewriting of the axioms of the specification. A test case consists of pairs of terms and a flag indicating the relationship between the terms (equivalent or not). A strategy of

"approximate check" for object observational equivalent was proposed, known as the **EQN** method.

Motivated by the work of ASTOOT, Chen et al [4] used fundamental pairs, which are pairs of equivalent ground terms formed by replacing all the variable on both sides of an axiom by normal forms, as test cases. They also gave a new algorithm referred as "relevant observable context approach" for determining the observational equivalence of two objects.

Using algebraic specifications, it is easy to create test data and to solve the oracle problem. But the algebraic style is often criticized as rather restrictive and cumbersome to use in practice. Some important features of programs, for example non-termination and higher-order functions, are difficult to model in algebras; equations are often not expressive enough to conveniently capture properties which one may want to state as requirements.

Our test approach is based on RSL, which can be considered as a combination of an algebraic specification language and a model-based specification language, that allows the development of an implementation to be based on the model-based specification while providing the additional capability of algebraic specification for validating the specification. So our approach can hopefully inherit the above advantages from both algebraic and model-based testing techniques, and overcome the problems within those techniques.

Concerning the experience in the above papers, we take Dick and Faivre's partition analysis technique to construct disjunctive normal form (DNF) and derive the test calls from these DNF and we do not need to build a Finite State Automaton (FSA) in our approach. We adapt the ideas of ASTOOT [7] and Chen et al [4] that sequences of operations and axioms are used in building test cases to our approach. However, there are still few distinctions between their approaches and ours:

1. The approaches of ASTOOT and Chen impose many restrictions on the axioms that can be used to produce pairs of equivalent terms. For example, the axiom must be an equational axiom, the variables which occur in one side of the axiom should also appear on the other side and the return value of left hand side or right hand side of the axiom will be the main class[4]. These restrictions mean only a few axioms can be used to derive test cases with. In contrast, our approach puts no restrictions on axioms. Almost every axiom is able to be used to generate test cases.
2. The concept of observational equivalence of two abstract objects is not adopted in our approach and so we do not have any complicated algorithms, neither coarse nor accurate, to determine if objects are observational equivalence. In our approach, the return values of test cases are always of simple data types which are easy to compare. If the return value of the left hand side or right hand side of an axiom is of abstract data type, a variety of the original axiom, which has the simple return data type, will be used to generate test cases.

5 The Tool

The tool that implements the automatic generation test cases from RSL specifications is based on the approach described in the above sections. It has been written using the Gentle Compiler Construction System, a modern toolkit for compiler writers and implementors of domain specific languages, and amounts to a little more than ten thousand lines of Gentle code. The tool itself is integrated in the "rsltc" RAISE tools which provide type checking, pretty-printing, generation of confidence conditions, showing module dependencies, and translation to Standard ML and to C++.

The tool hooks onto the back of the RSL type checker to extract representations of RSL specifications in the form of abstract syntax trees and Gentle environment variables as the tool's input and produces as output a separate file containing the test cases, or includes the test cases into the original RSL file. That means the tasks of reading RSL files, analysing the structures and checking their syntax are performed by the RSL type checker. We can also use the C++ translator to translate the RSL specification and its test cases into a C++ program and execute the program immediately in order to evaluate the effect of the test cases.

Test case declarations are a new extension to RSL to support interpretation and translation. A test case declaration starts with the keyword **test_case** and followed by one or more test case definitions. Using the specification in Figure 1 as input, the test case generation tool produces as output a RSL **test_case** declaration. Two of those test cases are presented as follows:

test_case

[t1_left] count(**let** (j, m) = output(input(first_value(input(49, input(101, empty))), input(49, input(101, empty)))) **in** m **end**),
[t1_right] count(input(49, input(101, empty))),
[t1_result] count(**let** (j, m) = output(input(first_value(input(49, input(49, input(101, empty))),input(49, input(101, empty)))) **in** m **end**) $\leq$ count(input(49, input(101, empty))),
[t2_left] last_value(input(17, input(49, input(101, empty)))),
[t2_right] last_value(input(49, input(101, empty))),
[t2_pre] $17 < 32 \vee 17 > 126$,
[t2_result] last_value(input(17, input(49, input(101, empty)))) = last_value(input(49, input(101, empty)))

Appending the RSL test_case declaration generated by the tool to the original specification in Figure 1, a new version of the specification is created. The C++ translator is applied to translate the new specification into a C++ file. After compiling the C++ file and executing it, we get the final testing results. The testing results of the above two test cases are shown as follows:

```
[t1_left] 2
[t1_right] 2
[t1_result] true
[t2_left] 49
[t2_right] 49
[t2_pre] true
[t2_result] true
```

The tool can be seen at the web site: http://www.gzas.org/rslweb.

6 Conclusion

In this paper, we have presented an approach to generate test cases from RSL specifications. In contrast to other's work, our approach combines the techniques for testing algebraic specifications and model-based specifications. This approach can be fully automatic and we design a tool to derive test cases from RSL specifications using our approach. The overall aim of our work is to make RSL being actually used in industrial software development through building a series of tools that could be helpful to this process. Our approach, in order to be implemented as the test cases generating tool, must be based on techniques which should be very effective, reliable and practical.

In order to make the application of our approach to non-trivial specification practical, there are many issues remaining unsolved. While we have focused so far on module-testing, there are also many interesting questions pertaining to how to extend our approach to system level. In RSL, a system is a collection of modules organized in a hierarchy through class extending or instantiating children modules as objects in parents [12]. We shall also consider to introduce mutation testing technique [1] to our approach to examine whether our test cases have enough coverage to find the errors in a mutant specification. In the future, we also hope to do some research work on how to apply coalgebra theory, which is a relatively new research field, to the test cases generation of RSL specifications. Ultimately, we hope to use the results of these work to expand and improve our tool.

References

1. Bernhard K. Aichernig. Contract-based mutation testing in the refinement calculus. In *REFINE 2002, the British Computer Society - Formal Aspects of Computing refinement workshop*. Elsevier, July 2002.
2. Salimeh Behnia and Hélène Waeselynck. Test Criteria Definition for B Models. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings, Volume I*, volume 1709 of *Lecturen Notes in Computer Science*, pages 509–529. Springer, 1999.

3. Fabien Peureux Bruno Legeard and Mark Utting. Automated Boundary Tesing from Z and B. In *FME 2002:Formal Methods Getting IT Right, International Symposium of Formal Methods Europe*, pages 21–40. Springer-Verlag, July 2002.
4. Huo Yan Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. In Black and White: An Integrated Approach to Class-Level Testing of Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, July 1998.
5. John Derrick and Eerke Boiten. Testing Refinements of State-based Formal Specifications. *Software Testing, Verification and Reliability*, (9):27–50, July 1999.
6. Jeremy Dick and Alain Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*. Springer-Verlag, April 1993.
7. Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, January 1994.
8. Marie-Claude Gaudel and Perry R. James. Testing Algebraic Data Types and Processes: A Unifying Theory. *Formal Aspects of Computing*, 10(5 & 6):436–451, 1998.
9. Rob Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditional slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, 2002.
10. Robert M. Hierons. Testing from a Z Specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.
11. Merlin Hughes and David Stotts. Daistish: Systematic Algebraic Testing for OO Programs in the Presence of Side-effects. In *ISSTA'96*, 1996.
12. Tomasz Janowski Hung Dang Van, Chris George and Richard Moore. *Specification Case Studies in RAISE*. FACIT series. Springer, 2002.
13. Christophe Meudec. *Automatic Generation of Software Test Cases From Formal Specifications*. PhD thesis, The Queen's University of Belfast, May 1998.
14. Ccile Peraire, Stphane Barbey, and Didier Buchs. Test Selection for Object-Oriented Software Based on Formal Specifications.
15. A. Pretschner, O. Slotosch, H. otzbeyer, E. Aiglstorfer, and S. Kriebel. Model Based Testing for Real: The Inhouse Card Case Study, 2001.
16. The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.

Verifying OWL and ORL Ontologies in PVS

Jin Song Dong, Yuzhang Feng*, and Yuan Fang Li

School of Computing, National University of Singapore
{dongjs, fengyz, liyf}@comp.nus.edu.sg

Abstract. The Semantic Web vision is being realized to reach the full potential of the Web. Semantic data modeling is the foundation of the Semantic Web. The Web Ontology Language (OWL) and OWL Rules Language (ORL) provides basic machinery to the semantic mark-up for data. However, there is limited tool support for OWL and no tool support currently for ORL. In this paper, we propose to model OWL and ORL language semantics in PVS specification language so that OWL and ORL ontologies can be transformed and verified in the Prototype Verification System (PVS). PVS user-defined proof strategies are also developed to automate the proof process.

Keywords: PVS, Semantic Web, OWL, ORL, reasoning.

1 Introduction

Unlike conventional web as we have now, the Semantic Web (SW) [2] is a platform for inter-machine data and information exchange, filtering, integration, etc., across organizational boundaries without human supervision. It extends the current web and reaches its full potential by making it truly ubiquitous and ready for the machines. The Web Ontology Language (OWL) [7], a Recommendation by World Wide Web Consortium (W3C), defines the basic vocabulary for describing data on the web and is a layer on which Web Services can be developed. In a way, modeling of data using OWL is an important part of requirements engineering for Semantic Web.

In order for intelligent software agents to automatedly process data on the web, ontology languages such as DAML+OIL and (part of) OWL were originally designed to be decidable [19, 22]. However, the trade-off is the limited expressiveness, which forbids some very desirable properties to be specified. To partially overcome this limitation of OWL, the OWL Rules Language (ORL) [12] has recently been proposed by Horrocks & Patel-Schneider.

Reasoning tool support for OWL is limited at the moment. Moreover, currently there is no tool support for ORL. SW reasoning tools such as FaCT [11] and RACER [10] have been developed to be fully automated; hence they cannot support ORL without major modification. However, as it can be foreseen

* Author for correspondence: fengyz@comp.nus.edu.sg.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 265–279, 2005.
© Springer-Verlag Berlin Heidelberg 2005

that ORL be integrated into the ontology languages hierarchy, the correctness of OWL and ORL ontologies is crucial to establishing *trust* in Semantic Web.

SW can be regarded as an emerging area from the knowledge representation and the web communities. The software engineering community can also play an important role in the SW development. Software verification techniques can be applied to check SW ontology related properties. We believe SW will be a new research and application domain for software engineering, especially for software verification techniques. In this paper, we propose to develop reasoning environment in PVS for OWL and ORL.

The rest of the paper is organized as follows. We briefly introduce the Semantic Web, ontology languages, tools and PVS in Section 2. In Section 3, we present PVS semantics for OWL with ORL axioms. In Section 4, we concisely discuss transformation from ORL to PVS. Reasoning support for OWL and ORL using PVS theorem prover is presented through a few case studies in Section 5. Section 6 presents related works, summarizes our contribution and discusses possible future works.

2 Overview

2.1 Semantic Web, Languages and Tools

Semantic Web. Although the traditional World Wide Web (WWW) was originally designed for machine processing, it ends up to be consumed only by human, i.e., web contents are only visually marked-up for humans to read. To reach the its full potential, it is necessary to make the web a platform for intelligent software agents to interact with each other to accomplish complex tasks without human supervision. To achieve this goal, data on the web must be given structured and precise meaning so that software agents can process data cooperatively and autonomously. The Semantic Web [2] was proposed by Tim Berners-Lee as the next generation of the web and it is now a W3C activity in its second phase.

Ontology Languages. Data in SW are represented by ontologies, which define their concepts and relationships. Ontology languages provide vocabularies for expressing ontologies.

Built on top of XML, the Resource Description Framework (RDF) [14] is a model of metadata defining a mechanism for describing resources without assumptions about a particular application domain. RDF describes web resources in a simple triplet format: $\langle subject\ predicate\ object \rangle$, where *subject* is the resource of interest, *predicate* is one the properties of this resource and *object* states the value of this property. RDF Schema [4] provides facilities to describe RDF data. RDF Schema allows structured and semi-structured data to be mixed together, which makes them hard for machines to process.

The syntactic ambiguity and relatively limited expressiveness of RDF Schema is partially overcome by the DARPA Agent Markup Language (DAML) [19], which is built on top of RDF Schema and based on description logics. DAML pooled effort with the Ontology Inference Layer project [5] to produce the ontol-

ogy language DAML+OIL. It provides a richer set of language primitives to describe classes and properties than RDF Schema and allows only structured data.

In 2004, a new ontology language based on DAML+OIL, the Web Ontology Language (OWL) [21] became the W3C Recommendation. It consists of three sublanguages: OWL Lite, DL & Full, with increasing expressiveness. These languages are designed for user groups with different requirements. OWL Lite & DL are decidable but Full is generally not. The undecidability of OWL Full comes from relaxing certain constraints from OWL DL. For example, OWL Full does not enforce the mutual exclusiveness between classes, properties, data values and individuals.

Although the design of OWL has taken into consideration of the different expressiveness needs of different user groups, it is still not expressive enough. Some very desirable properties cannot be expressed even in OWL Full. An important reason for this is that although the language provides a relatively rich set of language primitives for describing classes, it does not provide as many primitives for describing properties. For example, it does not support property composition. In the light of this weakness, Horrocks and Patel-Schneider [12] proposed an extension to OWL, the OWL Rules Language (ORL), in a syntactically and semantically coherent manner. ORL incorporates Horn clause rules into OWL and makes rules part of axioms that can be used to express more complex classes and properties.

The major extensions of ORL are the inclusion of Horn clause rules and variable declarations. The rules are in the form of $\texttt{antecedent} \rightarrow \texttt{consequent}$, where both antecedent and consequent are conjunctions of atoms: class membership, property membership, individual (in)equalities. Informally, a rule means that if the antecedent holds, then the consequent must also hold. A simple example rule shown below states that if $?b$ is a parent of $?a$ and $?c$ is a brother of $?b$, then $?c$ is an uncle of $?a$.

$$\texttt{parent}(?a, ?b) \wedge \texttt{brother}(?b, ?c) \rightarrow \texttt{uncle}(?a, ?c)$$

Ontology Tools. Various ontology tools have been built to support the development of the SW, such as ontology design, creation, management, merging, maintenance, publishing, reasoning, etc. In the rest of this section, we will briefly introduce a few reasoning tools.

FaCT (**Fa**st **C**lassification of **T**erminologies) [11] is a description logics classifier developed at University of Manchester. FaCT supports automated concept-level reasoning (concept subsumption and satisfiability testing), but not instance-level reasoning. Currently FaCT supports DAML+OIL and OWL.

RACER (**R**enamed **A**Box and **C**oncept **E**xpression **R**easoner) [10] is a reasoner for the description logic $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D})^-$ [9]. It has a much richer set of functionalities compared to that of FaCT, including ontology creation, query, retrieval and evaluation, knowledge base conversion to DAML+OIL/OWL, etc.

2.2 PVS

The Prototype Verification System (PVS) is an integrated environment for the development of formal specifications written in the PVS specification language

[17]. It supports a wide range of activities in specification development: creation, documentation, type-checking, theorem-proving, etc. The distinguishing feature of PVS is its synergistic integration of an very expressive specification language and powerful theorem-proving capabilities.

PVS provides an expressive specification language that augments classical higher-order logic with a sophisticated type system with predicate subtypes and dependent types, and with parameterized theories and a mechanism for defining abstract data types such as lists and trees.

PVS specifications are organized into *theories*, which define data types, axioms, theorems and conjectures that can be reused by other theories.

PVS has a powerful interactive theorem prover/proof checker [16]. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic condition rewriting, simplification, etc. User-defined proof strategies can be used to enhance the automation in the proof checker.

The proof goal in PVS is represented as a sequent which consists of a list of formulas called antecedents and a list of formulas called consequents. The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents. Either or both of the antecedents and consequents may be empty. An empty antecedent is equivalent to true, and an empty consequent is equivalent to false, so if both are empty the sequent is false. Every proof in PVS starts with a single consequent. It can be seen that the structure of sequents in PVS very much resembles that of the rules in ORL except that in ORL the conjunction of antecedents implies the conjunction of consequents. But as pointed out in [12] that an ORL rule of multiple consequents can be easily transformed into multiple rules each with a single consequent. Therefore we believe PVS is a natural reasoner for ORL.

3 PVS Semantics for OWL and ORL

In order to use PVS to verify and reason ontologies with ORL axioms, it is necessary to define the PVS semantics for OWL & ORL. This semantic model forms the reasoning environment for verification using PVS theorem prover. In this section, we present a PVS specification for a subset of OWL Full language primitives and the newly proposed ORL. The complete model can be found online[1].

3.1 PVS Semantics for OWL Constructs

Basic Concepts

Everything in Semantic Web is a *Resource*. So we model it by defining a non-empty type in PVS.

```
RESOURCE: TYPE+
```

[1] `http://www-appn.comp.nus.edu.sg/~rpfm/ORL2PVS`

In OWL Full the universe of individuals consists of all resources. Thus we define *Individual* to be a type equivalent to *Resource*.

```
INDIVIDUAL: TYPE+ = RESOURCE
```

Each class in OWL is a resource, which has a number of individuals associated with it: the instances of this class. So we model *Class* as a subtype of *Resource* and define a function *instances* that maps a class to a set of individuals.

```
CLASS: TYPE+ FROM RESOURCE
instances: [CLASS -> set[INDIVIDUAL]]
```

A property relates resources to resources. So we model *Property* as a predicate over a tuple of two resources.

```
PROPERTY: TYPE = pred[[RESOURCE,RESOURCE]]
```

Class Relationships

The property *subClassOf* is defined as a boolean function from two classes. For a class `c1` to be the sub-class of class `c2`, the instances of `c1` must be a subset of the instances of `c2`.

```
subClassOf?(c1,c2:CLASS): bool =
(
    subset?(instances(c1),instances(c2))
)
```

Other class relationship properties such as *disjointWith* and *equivalentClass* are similarly defined.

Class and Property

The property *allValuesFrom* attempts to establish a maximal set of individuals as a class. It defines a class `c1` of all individuals `i1` for which it holds that if the pair `(i1,i2)` is in the property `p` implies that `i2` is an instance of class `c2`. So we model it as a function from a property `p` and a class `c2` to a class `c1` and specify its meaning as an axiom as follows.

```
allValuesFrom: [PROPERTY, CLASS -> CLASS]
allValuesFrom_ax: AXIOM FORALL (c1,c2:CLASS),(p:PROPERTY):
    (allValuesFrom(p,c2) = c1 IMPLIES FORALL (i1:INDIVIDUAL):
        member(i1,instances(c1)) IFF FORALL (i2:INDIVIDUAL):
            (p(i1,i2) IMPLIES member(i2,instances(c2)))))
```

Property Relationships

The property *subPropertyOf* states that a property $p1$ is a sub-property of property $p2$ if and only if all pairs $(i1, i2)$ in $p1$ are also in $p2$. Therefore it is modeled as a boolean function of two properties.

```
subPropertyOf?(p1,p2:PROPERTY): bool =
(
    FORALL (i1,i2:INDIVIDUAL):  (p1(i1,i2) IMPLIES p2(i1,i2))
)
```

3.2 ORL Extension

In ORL [12], a rule consists of an antecedent and a consequent, each of which consists of a (possibly empty) set of atoms. Atoms can be of the form $C(x)$, $P(x, y)$, $sameAs(x, y)$ or $differentFrom(x, y)$, where C is an OWL class description, P is an OWL property, and x, y are either variables, OWL individuals or OWL data values. Informally, an atom $C(x)$ holds if x is an instance of the class description C, an atom $P(x, y)$ holds if x is related to y by property P, an atom $sameAs(x, y)$ holds if x is interpreted as the same object as y, and an atom $differentFrom(x, y)$ holds if x and y are interpreted as different objects. A rule may be read as meaning that if the antecedent holds (is "true"), then the consequent must also hold.

An ORL rule will be modeled as a PVS rewrite rule, e.g., a universally quantified predicate of the form

$$a_1 \wedge a_2 \wedge ... \wedge a_m \Rightarrow c_1 \wedge c_2 \wedge ... \wedge c_n$$

where a_i and c_j are one of the four forms of atoms.

3.3 Proof Support for PVS

To make the proving process of PVS more automated, a set of rewrite rules and theorems is also defined. They aim to hide certain amount of underlying model from the verification and reasoning and to achieve abstraction and automation. Usually these rules relate several classes & properties by defining the effect of using them in a particular way. One simple example is the `subClassOf_transitive` theorem. It states that if a class `c1` is a sub-class of a class `c2` and `c2` is a sub-class of a class `c3`, then `c1` is a sub-class of `c3`.

```
subClassOf_transitive: THEOREM FORALL (c1,c2,c3:CLASS):
    subClassOf?(c1,c2) AND subClassOf?(c2,c3) IMPLIES
       subClassOf?(c1,c3)
```

The following theorem, *member_subClassOf* states that an instance of a particular class is also an instance of all the super classes of this class.

```
member_subClassOf: THEOREM
    FORALL (i:INDIVIDUAL),(c1,c2:CLASS):
        member(i, instances(c1)) AND subClassOf?(c1,c2)
            IMPLIES member(i, instances(c2))
```

4 Transforming ORL to PVS

As ORL is an extension to OWL with the inclusion of rules, we perform the transformation in two steps. We transform OWL constructs into PVS specifications first, followed by the transformation of ORL rules.

We have developed a tool in Java to automatically transform OWL ontologies into PVS specifications. For example, the following ontology fragment defines a class *Person* and specifies some of its properties. The transformed PVS fragment is shown at the right.

```
<owl:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <rdfs:subClassOf><owl:Restriction>
      <owl:onProperty
         rdf:resource="#hasParent"/>
      <owl:allValuesFrom
         rdf:resource="#Person"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction owl :cardinality="1">
      <owl:onProperty
         rdf:resource="#hasFather"/>
    </owl:Restriction></rdfs:subClassOf>
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class prefab="#Man"/>
    <owl:Class prefab="#Woman"/>
  </owl:unionOf></owl:Class>
```

```
Person: CLASS
Person_union_ax:
    AXIOM Person=unionOf((:Man,Woman:))
Person_subClassOf_ax_1:
    AXIOM subClassOf?(Person,Animal)
Person_subClassOf_ax_2: AXIOM subClassOf?
    (Person,allValuesFrom(hasParent,Person))
Person_subClassOf_ax_3: AXIOM subClassOf?
    (Person,cardinality(hasFather,1))
```

In order to facilitate reasoning about numbers, data type properties are transformed into predicates and functional data type properties are transformed into functions. The advantage of doing this will become clearer when we discuss reasoning in Section 5. For example, transformation of the datatype property *age* is given below the OWL fragment:

```
<owl:DatatypeProperty rdf:ID="age">
  <rdf:type rdf:resource="http://www.w3.org/
    2002/07/owl#FunctionalProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/
    2000/10/XMLSchema#nonNegativeInteger"/>
</owl:DatatypeProperty>

age: [INDIVIDUAL -> Nat]
```

The tool we developed is also capable of transforming instance ontologies into PVS specifications. For example, the following shows an OWL instance ontology fragment and the corresponding PVS specification.

```
<Description rdf:ID="Ian">
  <rdf:type>
    <owl:Class rdf:ID="Person"/>
  </rdf:type>
  <shoe size>14</shoe size>
  <age>37</age>
</Description>
```

```
Ian: INDIVIDUAL
Ian_Person_ax:
    AXIOM member(Ian,instanceOf(Person))
Ian_shoesize_14_ax: AXIOM shoe_size(Ian)=14
Ian_age_37_ax: AXIOM age(Ian)=37
```

Transformation of ORL rules is straightforward. Each rule is transformed into an axiom, which is a universally quantified Horn clause with each of the atoms transformed into a predicate. For example,

```
<owlr:Rule rdf:ID="Rule1">
  <owlr:antecedent>
    <owlr:individualPropertyAtom  polypro="hasParent">
      <owlr:Variable tournament="x1" />
      <owlr:Variable tournament="x2" />
    </owlr:individualPropertyAtom>
    <owlr:individualPropertyAtom  polypro="hasBrother">
      <owlr:Variable tournament="x2" />
      <owlr:Variable tournament="x3" />
    </owlr:individualPropertyAtom>
  </owlr:antecedent>
  <owlr:consequent>
```

```
    <owlr:individualPropertyAtom  polypro="hasUncle">
      <owlr:Variable tournament="x1" />
      <owlr:Variable tournament="x3" />
    </owlr:individualPropertyAtom>
  </owlr:consequent>
</owlr:Rule>
```

is transformed into

```
Rule1_ax: AXIOM FORALL (x1,x2,x3: RESOURCE)
    hasParent(x1,x2) AND hasBrother(x2,x3)
        IMPLIES hasUncle(x1,x3)
```

5 Ontology Reasoning Using PVS

In this section, we demonstrate how PVS can be used to check ontology-related properties and to reason beyond the modeling power of OWL & ORL. It is presented in two parts. Firstly, standard SW reasoning are performed. In the second part, we show how PVS can reason ORL and more complex properties that even ORL cannot express.

5.1 Standard SW Reasoning

Standard SW reasoning includes three categories, namely inconsistency checking, subsumption reasoning and instantiation reasoning. The following subsections illustrate each category with an example.

Inconsistency Checking. Ensuring the consistency of ontologies is an important task in various stages of ontology development, as inconsistent ontologies may lead agents to reason erroneously and make wrong conclusions.

To be precise, knowledge base consistency amounts to verifying whether every concept in the knowledge base admits at least one individual [15].

The following is an example of inconsistency checking in the animal ontology. After transforming the ontology into a PVS specification, we identified the following closely related classes, properties and their axioms.

```
Animal,Vegetarian,Cow,MadCow,Food,Meat,Vegetable:CLASS
eats:O_PROPERTY
Vegetarian_subClassOf_ax_1: AXIOM subClassOf?(Vegetarian,Animal)
Vegetarian_allValuesFrom_ax_1: AXIOM Vegetarian=allValuesFrom(eats,Vegetable)
Cow_subClassOf_ax_1: AXIOM subClassOf?(Cow,Vegetarian)
MadCow_subClassOf_ax_1: AXIOM subClassOf?(MadCow,Cow)
MadCow_subClassOf_ax_2: AXIOM subClassOf?(MadCow,someValuesFrom(eats,Meat))
Meat_subClassOf_ax_1: AXIOM subClassOf?(Meat,Food)
Vegetable_subClassOf_ax_1: AXIOM subClassOf?(Vegetable,Food)
Vegetable_disjointWith_ax_1: AXIOM disjointWith?(Vegetable,Meat)
```

We suspect that there is an inconsistency in the class of *MadCow*. To prove that, we assert the following theorem, which means that the class of *MadCow* does not admit any individual.

```
MadCow_inconsistent: THEOREM
    (EXISTS (i:INDIVIDUAL):member(i, instances(MadCow))) IMPLIES FALSE
```

After applying `(lemma)` to supply PVS with known facts (axioms), applying `(skolem!)` to remove quantifiers and instructing PVS to understand the subclass relationship between *MadCow* and *Vegetarian*, we need to prove `member(i!1,instances(Vegetarian))`, that $i!1$ is a member of *Vegetarian*, which can be proved by the theorem *member_subClassOf* introduced in Section 3.3.

By expanding the definition of *Vegetarian* and exploiting the fact that *MadCow* is a subclass of an anonymous class that *eats Meat*, we can finish up the proof using a `(grind)`, which is a catch-all strategy that is frequently used to automatically complete a proof branch or to apply all the obvious simplifications.

Subsumption Reasoning. The task of subsumption reasoning is to infer that an OWL class is a sub-class of another. This could be accomplished in PVS fairly automatically. One of the simplest ways is by using the fact that `subClassOf?` is a transitive property, which can be easily proved by PVS.

There are other ways of proving subsumption relationships. One of them is by inter-class relationships such as `intersectionOf` and `UnionOf`. For example, we have the following transformed ontology fragment and we want to prove that the class *TallMan* is a subclass of *Person* using theorem *TallMan_subClassOf_Person* defined on the right:

```
TallMan_intersection_ax: AXIOM
  TallMan=intersectionOf((:TallThing,Man:))
Person_union_ax: AXIOM
  Person=unionOf((:Man,Woman:))
```

```
TallMan_subClassOf_Person: THEOREM
  subClassOf?(TallMan,Person)
```

The main steps of this proof are to prove separately `subClassOf?(TallMan,Man)` and `subClassOf?(Man,Person)`. Then the simple subsumption reasoning can finish proving the theorem. The above two goals can be proved by the application of two user defined theorems relating *intersectionOf* and *unionOf* to *subClassOf*, respectively.

Instantiation Reasoning. Instantiation reasoning asserts that one resource is or is not an instance of a class. Some SW reasoning tools such as FaCT are designed to only support concept-level reasoning. Hence reasoning at the instance-level cannot be performed by these tools. We demonstrate through an example that PVS supports instance-level reasoning.

In the example ontology, we defined an individual called *Santa*, who can move by both walking and flying, by the following axioms.

```
Santa_moves_walk_ax: AXIOM moves(Santa,walk)
Santa_moves_fly_ax: AXIOM moves(Santa,fly)
```

We want to prove that *Santa* is not an instance of the class *Person*. By stating the facts that all instances of the *Person* class can move only by walk, that the individual *Santa* can fly, and that *walk* and *fly* are disjoint, we can finish the proof with a `(grind)` command.

Table 1. The Model of Scheduling Tasks

```
:Agent a owl:Class.
:Task a owl:Class.
:TimePoint a owl:Class.
:Data a owl:Class.
:relatesTo a owl:TransitiveProperty;
  rdfs:domain Task;
  rdfs:range Task;
:assignedTo a owl:ObjectProperty;
  rdfs:domain Task;
  rdfs:range Agent.
:starts a owl:ObjectProperty;
  rdfs:domain Task;
  rdfs:range TimePoint.
:ends a owl:ObjectProperty;
  rdfs:domain Task;
  rdfs:range TimePoint.
:precedes a owl:TransitiveProperty;
  rdfs:domain TimePoint;
  rdfs:range TimePoint.
:overlaps a owl:ObjectProperty;
  rdfs:domain Task;
  rdfs:range Task.
:uses a owl:ObjectProperty;
  rdfs:domain Task;
  rdfs:range Data.
```

```
:a1 a Agent.
:a2 a Agent.
:t1 a Task;
  :starts :tp1;
  :ends :tp3;
  :assignedTo :a1.
:t2 a Task;
  :starts :tp2;
  :ends :tp4;
  :uses :d2;
  :assignedTo :a2.
:t3 a Task;
  :starts :tp4;
  :ends :tp5;
  :relatesTo :t1;
  :uses :d1;
  :assignedTo :a2.
```

```
:tp1 a TimePoint;
  :precedes :tp2.
:tp2 a TimePoint;
  :precedes :tp3.
:tp3 a TimePoint;
  :precedes :tp4.
:tp4 a TimePoint;
  :precedes :tp5.
:tp5 a TimePoint.
:d1 a Data.
:d2 a Data.
```

5.2 Checking ORL and Beyond

The above examples demonstrate PVS's power of performing consistency, subsumption and instantiation reasoning about OWL ontologies with certain degree of automation. Now we shall illustrate that PVS can reason about ORL and more complex properties that even ORL cannot capture.

ORL Reasoning. As stated earlier, one important reason of OWL expressive limitation is that while the language contains a rich set of class constructors, very little can be said about properties. Even simple composition of two properties is impossible to represent. It is for this reason that ORL is proposed. Here we demonstrate how PVS can act as a reasoner to support ORL.

We illustrate our idea with an example ontology about scheduling agents for different tasks, which is represented in n3 [3] syntax below in Table 1. The main reasoning task is, given a schedule and a set of constraints, to determine whether the schedule violates the constraints. Informally, there is a set of tasks and a set of agents. Any task can be assigned to any agent. There is also a set of discrete time points and a set of data. A time point may precede another. Each task starts and ends at a particular time point and may possibly use a piece of data. A task could relate to another task. Some tasks may overlap with some other task(s).

Four rules capture the requirements of the system. The first one states that an agent cannot be assigned to two overlapping tasks. The transformed PVS theorem is given on the right.

$$Task(t1) \wedge Task(t2) \wedge Agent(a1) \wedge Agent(a2) \wedge assignedTo(t1, a1) \wedge assignedTo(t2, a2) \wedge overlaps(t1, t2) \rightarrow differentFrom(a1, a2)$$

```
rule_1: AXIOM FORALL(t1,t2,a1,a2 : RESOURCE):
  member(t1,instances(Task)) AND
  member(t2,instances(Task)) AND
  member(a1,instances(Agent)) AND
  member(a2,instances(Agent)) AND
  assignedTo(t1,a1) AND assignedTo(t2,a2)
  AND overlaps(t1,t2)
    IMPLIES
  differentFrom?(a1,a2)
```

Since ORL rules transformation to PVS is straightforward, as we previously mentioned in Section 4, we will omit the PVS version of the following rules.

The second rule requires that related tasks must be assigned to the same agent.

$$Task(t1) \wedge Task(t2) \wedge Agent(a1) \wedge Agent(a2) \wedge assignedTo(t1, a1) \wedge assignedTo(t2, a2) \wedge relatesTo(t1, t2) \rightarrow sameAs?(a1, a2)$$

The third rule requires that any two overlapping tasks cannot use the same piece of data.

$$Task(t1) \wedge Task(t2) \wedge Data(d1) \wedge Data(d2) \wedge uses(t1, d1) \wedge uses(t2, d2) \wedge overlaps(t1, t2) \rightarrow differentFrom?(d1, d2)$$

The last rule defines when two tasks are overlapping - when one task that starts earlier ends after the other task starts.

$$Task(t1) \wedge Task(t2) \wedge TimePoint(tp1) \wedge TimePoint(tp2) \wedge TimePoint(tp3) \wedge TimePoint(tp4) \wedge starts(t1, tp1) \wedge ends(t1, tp2) \wedge starts(t2, tp3) \wedge ends(t2, tp4) \wedge precedes(tp1, tp3) \wedge precedes(tp3, tp2) \rightarrow overlaps(t1, t2)$$

To prove that the schedule violates some of the constraints, we simply prove the following PVS theorem: `violateConstraint: theorem FALSE`.

A proof strategy is intended to capture patterns of inference steps. A defined proof rule is a strategy that is applied in a single atomic step so that only the final effect of the strategy is visible and the intermediate steps are hidden from the user. We define a number of proof strategies, such as `(installTimePoint)`, `(installData)`, `(installAgent)`, etc., each of which introduces all the axioms one by one of a particular class. The following strategy introduces to PVS all facts related to all the time points.

```
(defstep installTimePoint ()
  (then
    (lemma "tp1_instanceOf_ax")
    (lemma "tp1_precedes_ax")
    (lemma "tp2_instanceOf_ax")
    (lemma "tp2_precedes_ax")
```

```
    (lemma "tp3_instanceOf_ax")
    (lemma "tp3_precedes_ax")
    (lemma "tp4_instanceOf_ax")
    (lemma "tp4_precedes_ax")
    (lemma "tp5_instanceOf_ax")
  )
  "Installing all axioms of TimePoint"
  "Installing all axioms of TimePoint"
)
```

Then we also define a strategy which finds and installs the transitive closure of the property `precedes`, i.e., the relative temporal order of all pairs of time points, as follows. This is needed for determining instances of the `overlaps` property later.

```
(defstep installAllPrecedes ()
  (then
    (lemma "precedes_transitive_ax")
    (rewrite "transitiveProperty?")
    (try (forward-chain -1) (installAllPrecedes) (delete -1))
  )
  "Finding and installing all precedes property instances"
  "Finding and installing all precedes property instances"
)
```

Basically this strategy repeatedly forward-chains the `precedes_transitive_ax` axiom until there is no more effect. Similarly, we find all instances of the property `relatesTo` by using the strategy `installAllRelatesTo` (not shown here).

Now we apply the rules. First, we apply the fourth rule to discover all instances of the property `overlaps` by using the strategy `installAllOverlaps` below.

```
(defstep installAllOverlaps ()
  (then
    (lemma "rule_4")
    (try (forward-chain -1) (installAllOverlaps) (delete -1))
  )
  "Finding and installing all overlaps property instances"
  "Finding and installing all overlaps property instances"
)
```

Then we can apply the other three rules one by one by using strategies similarly.

We apply the `(grind)` command, which proves the theorem. It means that the schedule cannot satisfy the conjunction of all constraints. A closer look at the ontology discovers that tasks $t1$ and $t2$ are related and yet overlapping. This reasoning technique becomes more important when the ontology contains more classes and more complicated properties.

Reasoning Beyond ORL. One example that OWL & ORL cannot deal with is the concrete domains: it can only make assertions about linear (in)equalities of cardinalities of property instances over integer. PVS, on the other hand, can perform basic arithmetic operations and comparisons, which we believe could improve the proof power beyond SW.

We illustrate the idea with the same schedule example. In the previous section, we model time as discrete time points and their temporal relationship as

an abstract *precedes* property. Now we can use the natural number domain to model time. Correspondingly, the *starts* and *ends* properties would have to be refined into functions from *Task* to natural number. Then the *overlaps* property could be refined as follows.

```
overlaps_ax: theorem
  FORALL (t1,t2:INDIVIDUAL):
    member(t1,instances(Task)) AND
    member(t2,instances(Task)) AND
    starts(t1) < starts(t2) AND starts(t2) < ends(t1)
      IMPLIES
    overlaps(t1,t2)
```

The above is just a simple example property that ORL cannot specify. If more constraints are to be put into the ontology, such as deadline for the whole schedule (which requires addition over numbers) or axioms other than $C(x)$, $P(x, y)$, $sameAs(x, y)$ and $differentFrom(x, y)$ are to be put into rules, more interesting properties would arise, which are also inexpressible in ORL.

6 Conclusion

Ensuring the correctness of shared ontologies is an important task in ontology development as inconsistent ontologies may lead agents to draw erroneous conclusions. In our previous works [8, 20], we have attempted to use a combination of SW and formal methods tools to reason about DAML+OIL/RDF ontologies. We used Alloy Analyzer (AA) [13], Z/EVES [18], RACER and OilEd [1] in combination to check for properties of interest. Some properties are beyond the modeling power of DAML+OIL. In this approach, the various tools were used in a complementary way such that a balance of automation and expressiveness is achieved. Moreover, the source of ontological errors can be traced in AA.

There are a few drawbacks to this approach. Firstly, AA does not scale up very well and secondly, Z/EVES works interactively, as PVS theorem prover does.

One part of the future works is to enhance the proof support for OWL and ORL. The PVS reasoner will be more effective if the semantics include not only essential functions but also sufficient supporting lemmas and theorems that makes proof of trivial goals more automated.

PVS is a generic theorem prover. As a result, it lacks complete automation. Hence, another part of the future works is to reduce user interactions as much as possible so that the reasoning procedure can be more efficient. This can be achieved by developing more advanced proof strategies.

Model-checking capabilities used for automatically verifying temporal properties of finite-state systems have recently been integrated into PVS. Hence PVS could be used to model and reason behaviors of SW services such as DAML-S [6].

In conclusion, we presented the PVS semantics for the Semantic Web ontology language OWL and its proposed extension ORL, the transformation process from OWL ontologies to PVS specifications, and our approach of using PVS theorem prover to reason ontology-related properties, sometimes beyond the modeling

capabilities of ORL. Some of the advanced features such as proof strategy are incorporated in our approach.

ORL is a newly proposed ontology language. To our knowledge, so far there is no existing reasoning system that can support ORL prior to this work. The high expressiveness of PVS language, its highly tunable proof strategies and similarity between PVS sequents and ORL rules make PVS a natural reasoner for ORL.

Acknowledgements

This work is supported by the DIRP Grant *"Formal Design Methods and DAML"* from Defense Science & Technology Agency (DSTA) Singapore. We would also like to thank Hai Wang, who directed our attention to ORL and provided valuable comments on an earlier draft of this paper.

References

1. Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a reasonable ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, number 2174 in Lecture Notes in Computer Science, pages 396–408, Vienna, September 2001. Springer-Verlag.
2. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):35–43, 2001.
3. Tim Berners-Lee. Notation 3 – Ideas about Web architecture. http://www.w3.org/DesignIssues/Notation3.html, 1998.
4. D. Brickley and R.V. Guha (editors). Resource description framework (rdf) schema specification 1.0. http://www.w3.org/TR/rdf-schema/, February 2004.
5. J. Broekstra, M. Klein, S. Decker, D. Fensel, and I. Horrocks. Adding formal semantics to the web: building on top of rdf schema. In *ECDL Workshop on the Semantic Web: Models, Architectures and Management*, 2000.
6. M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml service. http://www.daml.org/services/daml-s/2001/05/.
7. M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein (editors). OWL Web Ontology Language 1.0 Reference. http://www.w3.org/TR/owl-ref/, 2002.
8. J. S. Dong, C. H. Lee, Y. F. Li, and H. Wang. A combined approach to checking web ontologies. In *Proceedings of 13th World Wide Web Conference (WWW'04)*, pages 714–722, New York, USA, May 2004.
9. Volker Haarslev and Ralf Möller. Practical Reasoning in Racer with a Concrete Domain for Linear Inequations. In Ian Horrocks and Sergio Tessaris, editors, *Proceedings of the International Workshop on Description Logics (DL-2002)*, Toulouse, France, April 2002. CEUR-WS.
10. Volker Haarslev and Ralf Möller. *RACER User's Guide and Reference Manual: Version 1.7.6*, December 2002.
11. I. Horrocks. The FaCT system. *Tableaux'98, LNCS*, 1397:307–312, 1998.

12. Ian Horrocks and Peter F. Patel-Schneider. A proposal for an owl rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731, New York, USA, May 2004. ACM. http://www.cs.man.ac.uk/~horrocks/DAML/Rules/.
13. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *The 22nd International Conference on Software Engineering (ICSE'00)*, pages 730–733, Limerick, Ireland, June 2000. ACM Press.
14. F. Manola and E. Miller (editors). RDF Primer. http://www.w3.org/TR/rdf-primer/, February 2004.
15. Daniele Nardi and Ronald J. Brachman. An introduction to description logics. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors, *The description logic handbook: theory, implementation, and applications*, pages 1–40. Cambridge University Press, 2003.
16. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
17. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, December 2001.
18. M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lect. Notes in Comput. Sci.*, pages 72–85. Springer-Verlag, 1997.
19. F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks (editors). Reference description of the DAML+OIL ontology markup language. Contributors: T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L. A. Stein, ..., March, 2001.
20. Hai Wang. *Semantic Web and Formal Design Methods*. PhD thesis, National University of Singapore, 2004.
21. World Wide Web Consortium (W3C). OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/, March 2003.
22. World Wide Web Consortium (W3C). Web Ontology Language (OWL) Use Cases and Requirements. http://www.w3.org/TR/webont-req/, March 2003.

Symbolic and Parametric Model Checking of Discrete-Time Markov Chains

Conrado Daws*

Nijmegen Institute for Computing and Information Sciences,
University of Nijmegen, The Netherlands
daws@cs.ru.nl

Abstract. We present a language-theoretic approach to symbolic model checking of PCTL over discrete-time Markov chains. The probability with which a path formula is satisfied is represented by a regular expression. A recursive evaluation of the regular expression yields an exact rational value when transition probabilities are rational, and rational functions when some probabilities are left unspecified as parameters of the system. This allows for parametric model checking by evaluating the regular expression for different parameter values, for instance, to study the influence of a lossy channel in the overall reliability of a randomized protocol.

1 Introduction

In recent years, the need to formally reason about probabilistic behaviour, exhibited, for instance, by randomized algorithms, or communication protocols and computer networks with unreliable or unpredictable behaviour, has triggered research in the area of formal methods for the specification and verification of probabilistic systems. The general approach has consisted in extending those models, logics and techniques, which have proved successful in the non-probabilistic setting, with probabilities . In particular, this has lead to the theory of probabilistic model checking [8, 5] of PCTL [14, 1] over discrete probabilistic systems, and, in the last few years, to tools implementing it [17, 21].

Discrete probabilistic systems are typically modelled by an extension of transition systems with discrete probability distributions. In this model, a set of outgoing distributions on the set of states is associated with every state. Each such distribution gives the probability with which the source state can reach some target state in one step. Models with at most one distribution per state are said to be fully probabilistic, and usually referred to as a *discrete-time Markov chains* (DTMC). Models with both nondeterministic and probabilistic choice are usually referred to as a *Markov Decision Processes* (MDP).

The logic PCTL is a version of CTL where the existential and universal quantification over paths in CTL are replaced with a *probabilistic operator* $\mathcal{P}_{\sim\lambda}(\cdot)$,

* This work was originally carried out at the Formal Methods and Tools group of the University of Twente, supported by the European Community Project IST-2001-35304 AMETIST.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 280–294, 2005.
© Springer-Verlag Berlin Heidelberg 2005

where $\sim \in \{\leq, <, >, \geq\}$, and $\lambda \in [0, 1]$ is the *probability threshold*, affording the specification of properties such as "a leader will eventually be elected with probability 1" or "the chance of shutdown occurring is at most 0.001".

Probabilistic model-checking of PCTL over discrete probabilistic systems is based on the computation in every state of the probability measure of the set of paths satisfying a (path) formula. These probabilities are computed numerically by solving a system of linear equations in the case of DTMCs [14], and by solving a linear optimization problem in the case of MDPs [5].

We present a new, language-theoretic, approach for probabilistic model checking of DTMCs. Within our approach, transition probabilities are considered *letters* of an alphabet of a finite state automaton. The probability measure of a set of paths satisfying a formula is computed symbolically as a *regular expression* on that alphabet, with the standard algorithms to obtain a regular expression from a finiste state automaton. The regular expression is then evaluated to its *exact rational value* when transition probabilities are rational. Moreover, the symbolic representation of probability measures as regular expressions allows us to leave transition probabilities unspecified as formal parameters. In this case, the evaluation of a regular expression is a quotient between two polynomials on the parameters, with rational coefficients.

In this way, we can perform *parametric model checking*, e.g., check whether a formula holds for different values of the parameters, for instance, to study the influence of a lossy channel on the reliabilty of a protocol, or to obtain algebraically the value of a parameter such that the system satisfies some property. However, parametric model-checking is applicable only for formulas without nested probabilistic operators, but this does not represent a strong restriction in practice because such formulas are not needed to specify properties of interest.

The remainder of the article is organized as follows. Section 2 is a short introduction to the theory behind probabilistic model checking of PCTL over discrete-time Markov chains. Section 3 introduces our technique for symbolic model-checking of DTMCs, and we extend it to the parametric case in Section 4. We illustrate its application with two small case studies in Section 5. Finally, Section 6 concludes our presentation with a discussion of related and future work.

2 Probabilistic Model Checking

We start with a short introduction to model checking of PCTL formulas for discrete-time Markov chains. Throughout this section, we consider a given set of atomic propositions AP.

2.1 Discrete Time Markov Chains

A *discrete-time Markov chain* is a tuple $\mathcal{D} = (\mathcal{S}, s_0, \mathbf{P}, L)$ where

- $\mathcal{S}$ is a finite set of states
- $s_0 \in \mathcal{S}$ is an initial state

- $L : \mathcal{S} \mapsto 2^{\mathsf{AP}}$ is a labelling function which gives the atomic propositions that are true in a state.
- $\mathbf{P} : \mathcal{S} \times \mathcal{S} \mapsto [0,1] \cap \mathbb{Q}$ is a probability matrix with rational values such that for all $s \in \mathcal{S}$, $\sum_{t \in \mathcal{S}} \mathbf{P}(s,t) = 1$.

The function $\mathbf{P}(s,\cdot)$ is the distribution on $\mathcal{S}$ for state s. Notice that states with no outgoing distribution can be considered by adding a self-loop with probability 1, without changing the transient and limiting probabilities of the system. The matrix entry $\mathbf{P}(s,t)$ gives the probability of making a transition from s to t. The probability of following a finite path $s_0 s_1 \ldots s_n$ is $\mathbf{P}(s_0,s_1) \cdot \mathbf{P}(s_1,s_2) \cdot \ldots \cdot \mathbf{P}(s_{n-1},s_n)$. These probabilities for finite paths give rise to a unique probability measure $\Pr_s$ on the set $Path_s$ of infinite paths starting in state s, defined on the sets of paths having a finite common prefixe, such that $\Pr_s(\{\omega | \omega = s s_1 \ldots s_n.\omega'\}) = \mathbf{P}(s,s_1) \cdot \mathbf{P}(s_1,s_2) \cdot \ldots \cdot \mathbf{P}(s_{n-1},s_n)$ [19].

2.2 The Logic PCTL

The logic PCTL [14, 5] is a version of CTL where the existential and universal quantification over paths in CTL are replaced by a *probabilistic operator* $\mathcal{P}_{\sim\lambda}(\cdot)$, with $\sim \in \{\leq, <, >, \geq\}$ and $\lambda \in [0,1]$ rational is the *probability threshold*, that can be applied to a path formula. The formal syntax of PCTL formulas over AP is given by the following grammar:

$$\phi ::= \mathit{true} \mid a \in \mathsf{AP} \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{P}_{\sim\lambda}(\alpha)$$
$$\alpha ::= \mathsf{X}\phi \mid \phi \mathsf{U} \phi$$

2.3 Semantics and Model Checking

The semantics of PCTL is the same as that of CTL for the fragment where they both coincide. The semantics of the probabilistic operator is:

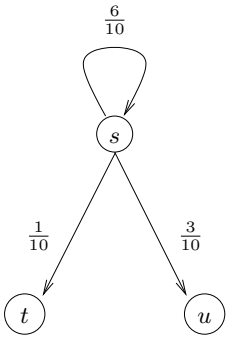
$$s \models \mathcal{P}_{\sim\lambda}(\alpha) \;\text{ iff }\; \Pr_s(\{\omega \in \mathit{Path}_s \mid \omega \models \alpha\}) \sim \lambda$$

meaning that the probability measure of the set of paths satisfying α is calculated and compared to the threshold λ, yielding true or false.

The model checking algorithm proceeds in the same way as for CTL, by induction on ϕ. The only difference is the evaluation of the probabilistic operator appearing in sub-formulas of the type $\mathcal{P}_{\sim\lambda}(\mathsf{X}\phi)$ and $\mathcal{P}_{\sim\lambda}(\phi_1 \mathsf{U} \phi_2)$. The example below shows the standard approach based on numerical solutions of a linear equation system. Section 3 presents our symbolic algorithm based on regular expressions.

2.4 A Simple Example

Let's consider the DTMC of Fig. 1 (left). The initial state s has a probabilistic branching to t with probability $\frac{1}{10}$, to u with probability $\frac{3}{10}$ and to itself with probability $\frac{6}{10}$. The probability with which t can be reached from s, denoted



$$x_s = \frac{6}{10}x_s + \frac{1}{10}x_t + \frac{3}{10}x_u$$
$$x_t = 1$$
$$x_u = 0$$

Fig. 1. A simple DTMC and the corresponding linear equation system to compute the probabilities for $true\mathsf{U}t$ with solution $x_s = \frac{1}{4}, x_t = 1, x_u = 0$

x_s, is the probability measure of the set of paths starting in s and satisfying the formula $\alpha = true\mathsf{U}t$. Its numerical value is obtained as the unique solution of the linear equation system of Fig. 1 (right), which is $x_s = \frac{1}{4}$. It follows that $s \models \mathcal{P}_{\leq \frac{1}{4}}(\alpha)$ and $s \models \mathcal{P}_{\geq \frac{1}{4}}(\alpha)$.

Tools like PRISM [21] and RAPTURE [17] find the solution of the linear equation system using iterative methods (e.g. Jacobi, Gauss-Seidel), that numerically approximate the solution. It must be noticed that since these methods do not compute the exact solution, those tools might yield the wrong result when the solution is equal, or close enough, to the threshold of the formula, like in this example. Our symbolic approach does not suffer from this, but the same goal could be achieved using direct methods on rational numbers with arbitrary precision.

3 Symbolic Model-Checking of DTMCs

This section presents a language-theoretic approach to model checking of DTMCs. It is based on deriving from a DTMC a finite state automaton recognizing a language over an alphabet consisting of the *strictly positive* transition probabilities appearing in the matrix $\mathbf{P}$. The initial state of the FSA is the state on which the formula is to be checked, whilst the sets of final states and the transition function depend on the path formula under consideration. A path formula yields a regular expression that is evaluated recursively to the *exact rational value* of the probability measure of the set of paths satisfying it.

3.1 Derived FSA

We derive from a DTMC $\mathcal{D} = (\mathcal{S}, s_0, \mathbf{P}, L)$ a finite state automaton $\mathcal{A} = (\mathcal{S}, \Sigma, \delta, \mathcal{S}_f)$ such that:

- $\mathcal{S}$ is the same set of states of $\mathcal{D}$
- $\mathcal{S}_f \subseteq \mathcal{S}$ is a subset of final states

Table 1. Evaluation of regular expressions as rational numbers

$\mathrm{val}(p/q) = \frac{p}{q}$	$\mathrm{val}(x\|y) = \mathrm{val}(x) + \mathrm{val}(y)$
$\mathrm{val}(x^*) = \frac{1}{1-\mathrm{val}(x)}$	$\mathrm{val}(x.y) = \mathrm{val}(x) \times \mathrm{val}(y)$

- $\Sigma = \{(p/q) \mid \exists i, j \in \mathcal{S} \,.\, \mathbf{P}(i,j) = \frac{p}{q} > 0\}$ is the alphabet, consisting of the strictly positive entries of the probability matrix.
- $\delta : \mathcal{S} \times \Sigma \mapsto 2^{\mathcal{S}}$ is a transition function derived from $\mathbf{P}$ which associates with every pair of states and letters, a set of states such that if $\delta(s,a) = Q$ then for every $q \in Q$, $\mathbf{P}(s,q) = a$.

3.2 Evaluation of Regular Expressions

The set $\mathcal{R}(\Sigma)$ of regular expressions over the alphabet Σ, is the set of expressions containing the elements of Σ, and closed by union ($|$), concatenation ($.$) and star ($*$). These expressions can be evaluated to a rational value, by replacing union by addition ($+$), concatenation by multiplication ($\times$) and star by the limit of a geometric series. Formally, the evaluation function $\mathrm{val} : \mathcal{R}(\Sigma) \mapsto \mathbb{Q}$ is defined inductively by the rules of Table 1[1].

The regular language $\mathcal{L}(\mathcal{A}, s_i)$ recognized by $\mathcal{A}$ with initial state $s_i \in \mathcal{S}$, corresponds to the (possibly infinite) set Ω of finite paths from s_i to some final state in $\mathcal{S}_f$, following only transitions allowed by δ. Among all the regular expressions corresponding to this language, we consider a regular expression r computed with the *inductive* or the *state-elimination* algorithms of [15], without simplifying expressions of the type $a|a$ [2].

The following proposition states that the evaluation of r is the probability measure in s_i of the set of paths with prefixes in Ω.

Proposition 1. *Let r be a regular expression computed for $\mathcal{L}(\mathcal{A}, s_i)$. Then,*

$$\mathrm{val}(r) = \Pr_{s_i}(\{\omega \in Path_{s_i} \mid \exists\, k \geq 0 \,.\, \omega(k) \in \mathcal{S}_f, \textit{ and } \\ \forall\, l < k, \exists a \in \Sigma \,.\, \delta(\omega(l), a) \ni \omega(l+1)\})$$

[1] Notice that the evaluation of x^* is not defined when x evaluates to 1, but this does not happen in the regular expressions we obtain because the final states of the FSAs we consider have no outgoing transition, thus, every cycle must be exited. The same remark applies for the parametric case.

[2] To be precise we should talk about regular expressions with multiplicities [4]. However, the regular expressions computed by the standard algorithms mentioned above without simplification, preserve the multiplicities of words, and, thus, our results hold.

3.3 Model-Checking

Let $\mathcal{D} = (\mathcal{S}, s_0, \mathbf{P}, L)$ be a DTMC and $\mathcal{P}_{\sim\lambda}(\alpha)$ a PCTL formula. We charaterize the set of paths satisfying α as a regular expression on Σ. For the next operator the regular expression can be obtained directly from $\mathbf{P}$ and α. For an until formula, we derive from $\mathcal{D}$ and α a finite automaton $\mathcal{A}_\alpha$ generating the probability measure of paths satisfying α. The set of states satisfying a state formula ϕ is denoted by $[\![\phi]\!]$.

Next Formulas. Let $\alpha = \mathsf{X}\phi$ be a next formula. A regular expression corresponding to the set of paths satisfying α is $|_j(p/q)_j$ such that $s_j \in [\![\phi]\!]$ and $\mathbf{P}(s_i, s_j) = \frac{p}{q}$.

Until Formulas. Let $\alpha = \phi_1 \mathsf{U} \phi_2$ be an until formula. The derived finite automaton is such that the final states are those satisfying ϕ_2, and the transition function is restricted to those states satisfying $\phi_1 \wedge \neg\phi_2$. Formally:

- $\mathcal{S}_f = [\![\phi_2]\!]$
- $\delta(s,a) = \begin{cases} \emptyset & \text{if } s \notin [\![\phi_1]\!] \text{ or } s \in [\![\phi_2]\!] \\ \{t \mid \mathbf{P}(s,t) = a\} \cap ([\![\phi_1]\!] \cup [\![\phi_2]\!]) & \text{otherwise} \end{cases}$

Model-Checking. Let $\mathcal{A}_\alpha = (\mathcal{S}, \Sigma, \delta, \mathcal{S}_f)$ be the finite automaton derived from $\mathcal{D}$ and α. Then, the following proposition states that the model checking problem can be solved for a state s_i by evaluating a regular expression equivalent to the language recognized by $\mathcal{A}$ with initial state s_i.

Proposition 2. *Let r be a regular expression computed for $\mathcal{L}(\mathcal{A}_{\alpha,s_i})$. Then,*

$$s_i \models \mathcal{P}_{\sim\lambda}(\alpha) \textit{ iff } \mathrm{val}(r) \sim \lambda$$

In order to model-check recursively formulas with nested probabilistic operators, we need to establish the validity of every probabilistic subformula in each state. In this case, the inductive algorithm for computing regular expressions which gives for every state a regular expression corresponding to the language it accepts, should be preferred. However, for efficiency reasons, the state-elimination algorithm is more appropriate to model-check simple formulas without nested probabilistic operators, like those usually considered in practice.

3.4 A Simple Symbolic Example

Let's consider the DTMC of Figure 1, and the path formula $\alpha_1 = \mathit{true}\mathsf{U}t$ to be evaluated in state s. We derive the finite automaton depicted in Figure 2 (left) with alphabet $\Sigma = \{1/10, 3/10, 6/10\}$, initial state $s_i = s$, final states $\mathcal{S}_f = \{t\}$ and a transition function defined by $\delta(s, 6/10) = \{s\}$, $\delta(s, 1/10) = \{t\}$ and $\delta(s, 3/10) = \{u\}$.

The language recognized by this automaton corresponds to the set of paths reaching t from s. It can be described by the regular expression $r = (6/10)^*.(1/10)$

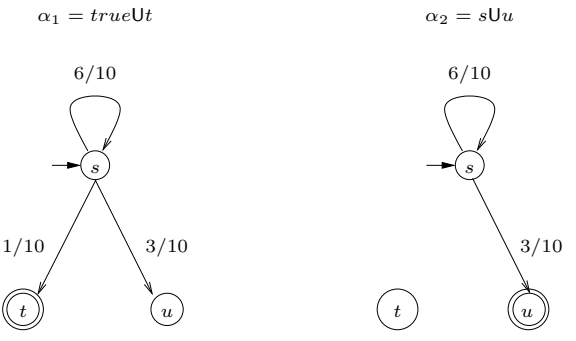


Fig. 2. Finite automata for the verification of α_1 and α_2 in s

which is evaluated to $\mathrm{val}(r) = \frac{1}{1-\frac{6}{10}} \times \frac{1}{10} = \frac{1}{4}$. It follows that $s \models \mathcal{P}_{\geq \frac{1}{4}}(\alpha_1)$ and $s \models \mathcal{P}_{\leq \frac{1}{4}}(\alpha_1)$, thus avoiding the problem arising with numerical computations. Figure 2 (right) also depicts the finite automaton derived for the evaluation of $\alpha_2 = s \mathsf{U} u$ in state s.

4 Model Checking Parametric DTMCs

Since regular expressions are computed formally, that is, probabilities are considered just symbols prior to evaluation, it is natural to extend our model checking technique to the case where probabilities are given as formal parameters. This makes possible to consider parametric models where some transition probabilities are left unspecified. The regular expression is in this case evaluated to a rational function, i.e., a quotient between two polynomials on the parameters, which can be manipulated algebraically for parametric analysis.

4.1 Parametric DTMCs

Let X be a set of formal parameters. A parametric DTMC is a DTMC where we extend the probability matrix to take values also in X. The formal parameters must satisfy the linear system corresponding to the stochastic condition of the probability matrix, i.e., for all $s \in \mathcal{S}$, $\sum_{t \in \mathcal{S}} \mathbf{P}(s,t) = 1$, and they must be *strictly positive* reflecting the fact that a transition between two states is present in the derived finite automaton only if it corresponds to a strictly positive probability.

A parametric DTMC gives rise to a family of DTMCs by instantiating the formal parameters to a value with an instantiation function $\kappa : \mathbb{Q}_+ \cup X \mapsto [0,1]$ such that for all $q \in \mathbb{Q}_+$, $\kappa(q) = q$, for all $x \in X$, $\kappa(x) > 0$, and for all $s \in \mathcal{S}$, $\sum_{t \in \mathcal{S}} \kappa(\mathbf{P}(s,t)) = 1$. For a parametric DTMC $\mathcal{D}_X$, and an instantiation function κ, $\kappa(\mathcal{D}_X)$ denotes the DTMC such that the probability matrix is obtained by instantiating the formal parameters.

Table 2. Evaluation of regular expressions as rational functions

$$\mathrm{val}(p/q) = \frac{p}{q} \qquad \mathrm{val}(r|s) = \frac{P_r Q_s + Q_r P_s}{Q_r Q_s}$$
$$\mathrm{val}(x \in X) = x \qquad \mathrm{val}(r.s) = \frac{P_r P_s}{Q_r Q_s} \qquad \mathrm{val}(r^*) = \frac{Q_r}{Q_r - P_r}$$

4.2 Evaluation of the Regular Expression

The finite state automaton for a parameterized DTMC and a path formula is derived as in the non-parametric case. The regular expression is also evaluated recursively. In this case, the operators on regular expressions, union, concatenation and star, are replaced by the corresponding addition, multiplication and inversion for *rational functions*, that is, quotients $\frac{P(X)}{Q(X)}$ between two polynomials on X.

The evaluation function $\mathrm{val} : \mathcal{R}(\Sigma) \mapsto \langle\mathbb{Q}\rangle X \times \langle\mathbb{Q}\rangle X$ associates with a regular expression r, a pair (P_r, Q_r) of polynomials on X with coefficients in $\mathbb{Q}$, noted $\frac{P_r}{Q_r}$, defined by induction on the regular expression following the rules in Table 2.

Let $\mathcal{D}_X$ be a parametric DTMC, $\mathcal{A}$ the derived FSA, and r a regular expression for its language computed with the inductive or the state-elimination algorithms. The following proposition states that the evaluation of r for any instantiation of the parameters κ, noted $\kappa(\mathrm{val}(r))$, is the probability measure in state s_i for $\kappa(\mathcal{D}_X)$, of the set of paths from s_i to some state in $\mathcal{S}_f$ following only transitions allowed by δ.

Proposition 3. *Let r be a regular expression computed for $\mathcal{L}(\mathcal{A}, s_i)$. Then,*

$$\kappa(\mathrm{val}(r)) = \mathrm{Pr}_{s_i,\kappa(\mathcal{D}_X)}(\{\omega \in Path_{s_i} \mid \exists\, k \geq 0\,.\,\omega(k) \in \mathcal{S}_f, \textit{ and } \forall\, l < k, \exists a \in \Sigma\,.\,\delta(\omega(l), a) \ni \omega(l+1)\})$$

4.3 Model Checking Simple PCTL Formulas

Let $\mathcal{A}_\alpha = (\mathcal{S}, \Sigma, \delta, \mathcal{S}_f)$ be the finite automaton derived from $\mathcal{D}_X$ and a path formula α that does not contain nested probabilistic operators. The following proposition states that model-checking such path formulas for a state s_i in $\kappa(\mathcal{D}_X)$ consists in evaluating a regular expression equivalent to the language recognized by $\mathcal{A}_\alpha$ with initial state s_i, for the instantiation κ.

Proposition 4. *Let r be a regular expression computed for $\mathcal{L}(\mathcal{A}_{\alpha,s_i})$. Then,*

$$s_i \models_{\kappa(\mathcal{D}_X)} \mathcal{P}_{\sim\lambda}(\alpha) \textit{ iff } \kappa(\mathrm{val}(r)) \sim \lambda$$

Thus, by evaluating the corresponding regular expression, we obtain an algebraic expression of the probability measure of the sets of paths satisfying a path formula, as a rational function on the parameters. We can use the result to

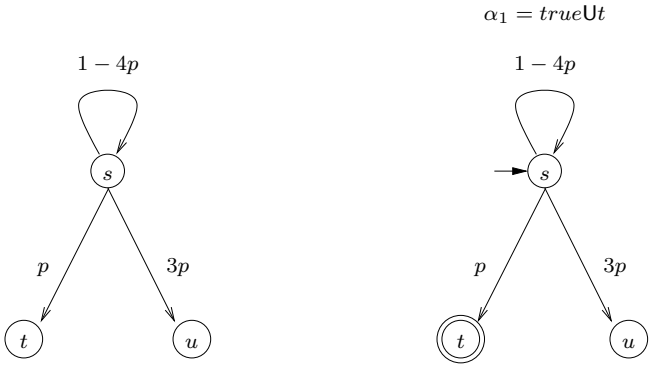


Fig. 3. Parametric DTMC and FSA for the verification of $true\mathsf{U}t$ in s

check whether the system satisfies a formula for different values of the parameters, without having to model check the system every time. Moreover, we can manipulate the algebraic expression in order to synthesize the values of certain parameters such that a formula is satisfied.

Parametric model-checking is however restricted to formulas without nested probabilistic operators, because a recursive evaluation of a formula is not possible in general, since the set of states satisfying a probabilistic formula is a parameterized set. This is not a strong restriction in our opinion, since in practice general formulas are not necessary to specify properties of interest. Moreover, such formulas are also problematic when using iterative numerical methods because of the propagation of the numerical error inherent to these methods.

4.4 A Simple Parametric Example

Now let's consider that the transition probabilities of the DTMC of Figure 1 are not completely specified, and that we have the parametric DTMC depicted in Figure 3 (left), such that $\mathbf{P}(s,t) = p$, $\mathbf{P}(s,u) = 3p$ and $\mathbf{P}(s,s) = 1-4p$, for $0 < p < \frac{1}{4}$.

The finite state automaton derived for the verification of α_1 is depicted in Figure 3 (right). The regular expression for the language it accepts is $r = (1-4p)^*.p$, which is evaluated to $\mathrm{val}(r) = \frac{1}{1-(1-4p)} \times p = \frac{1}{4}$. That is, state s satisfies both $s \models \mathcal{P}_{\geq \frac{1}{4}}(\alpha_1)$ and $s \models \mathcal{P}_{\leq \frac{1}{4}}(\alpha_1)$ for any valid value of p. Notice that the evaluation of the regular expression is not defined for $p = 0$ but it is for $p = \frac{1}{4}$, hence we could relax the requirement that $\mathbf{P}(s,s)$ be strictly positive. Intuitively, this corresponds to removing the self-loop in s, which does not disconnect the graph.

5 Application

We apply our formal model checking approach to two small examples. We generate the regular expressions for the derived finite automata using the state-

elimination algorithm implemented in JFLAP[13, 18] and a simple script to evalute them.

5.1 Simulating a Dice with a Coin

We consider a probabilistic program due to Knuth and Yao [20], which models a fair dice using only fair coins, that has already been analyzed using a probability theory [16] for the theorem prover HOL [11, 12], and the probabilistic symbolic model checker PRISM [22].

The DTMC of Figure 4 generates a uniform distribution on $\{1, \ldots, 6\}$ from a source of independent, unbiased, random bits, which can be seen as a model of a dice using a fair coin. Starting at state 0, the coin is tossed. Whenever heads appears, the system takes the upper branch and when tails appears, the lower branch. This continues until the value of the dice is decided.

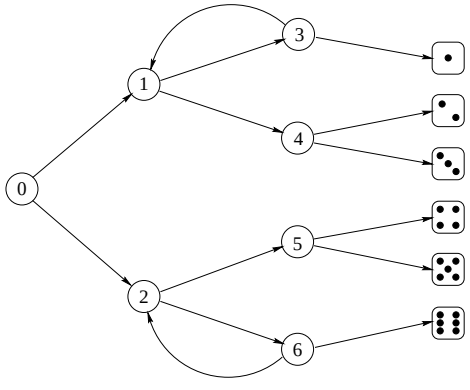


Fig. 4. Simulating a dice tossing a coin: upper branches correspond to tail, and lower branches to head

The properties of interest of this example are that it terminates with probability 1, and that it generates the uniform distribution. Let $\boxed{i}$ be an atomic proposition characterizing the state where the value i was obtained. Let $\alpha_0 = true \mathsf{U} \bigvee_{i=1}^{i=6} \boxed{\mathrm{i}}$ and $\alpha_i = true \mathsf{U} \boxed{\mathrm{i}}$ for $i = 1 \ldots 6$. Then, the initial state s_0 must satisfy the following PCTL formulas, for $i = 1 \ldots 6$:

$$\mathcal{P}_{\geq 1}(\alpha_0), \quad \mathcal{P}_{\leq \frac{1}{6}}(\alpha_i), \quad \mathcal{P}_{\geq \frac{1}{6}}(\alpha_i)$$

Table 3 shows the results from applying our model checking approach to the dice model. For each path formula α_i, the second column gives the regular expression[3] corresponding to $\mathcal{L}(\mathcal{A}_{\alpha_i})$ and the third column gives its evaluation. The

[3] Although the language is the same for every α_i, JFLAP can return different regular expressions.

Table 3. Regular expressions and evaluations for model checking the dice example

α	$r = \mathcal{L}(\mathcal{A}_\alpha)$	val(r)
$\alpha_1, \alpha_2, \alpha_3, \alpha_6$	$(1/2).((1/2).(1/2))^*.(1/2).(1/2)$	$\frac{1}{6}$
α_4, α_5	$((1/2).(1/2)\|(1/2).(1/2).((1/2).(1/2))^*.(1/2).(1/2)).(1/2)$	$\frac{1}{6}$

Table 4. Regular expressions for parametric analysis of the dice example

α	$r = \mathcal{L}(\mathcal{A}_\alpha)$
α_1	$(1/2).(h_1.h_3)^*.h_1.(1-h_3)$
α_2, α_3	$(1/2).(h_1.h_3)^*.(1-h_1).(1/2)$
α_4, α_5	$(1/2).(1-t_2).(1/2)\|(1/2).t_2.(t_6.t_2)^*.t_6.(1-t_2).(1/2)$
α_6	$(1/2).t_2.(t_6.t_2)^*.(1-t_6)$

regular expression corresponding to $\mathcal{A}_{\alpha_0}$ is the union of the regular expressions for $\mathcal{A}_{\alpha_i}$, thus it is evaluated to 1. It follows that s_0 satisfies all above formulas.

Now we show how to do parametric analysis on the dice model when we allow the use of biased coins. Let $0 < h_i < 1$ and $t_i = 1 - h_i$ be the probabilities of getting head or tail in state s_i. In order to obtain the uniform distribution, we must have $h_0 = h_4 = h_5 = \frac{1}{2}$ for symmetry reasons, hence, only states s_1, s_2, s_3 and s_6 can use biased coins. Table 4 shows the regular expressions obtained using the formal parameters h_i and t_i.

We will prove that the uniform distribution can not be obtained with a single biased coin. First, we must have $\text{val}(r_{\alpha_1}) = \text{val}(r_{\alpha_2})$. This means that $h_1(1 - h_3) = (1 - h_1)/2$ and, hence, $h_1 = 1/(3 - 2h_3)$. Thus, if s_1 and s_3 must use the same coin, we should also have $h_1 = h_3$ or $h_1 = 1 - h_3$. Both cases yield a second degree equation, with a unique solution in $]0, 1[$, $h_1 = h_3 = \frac{1}{2}$.

5.2 The IPv4 Zeroconf Protocol

We consider a simple probabilistic model of part of the IPv4 Zeroconf protocol, designed for the self-configuration of IP network interfaces. This part, modelled by the DTMC of figure 5, taken from [6], deals with the collision-avoiding mechanism of the protocol. When a fresh host joins the network, which we assume to have a fixed configuration, it selects uniformly a random IP address among the $K = 65024$ possible addresses. If there are m hosts in the network the probability of a collision is $q = m/K$.

The host can select a new IP address with probability $1 - q$ and join the network. Otherwise, it tries to detect the collision by asking the other hosts whether they are using this address, and then waits for an answer. However, the new host might not receive an answer in time with probability p, in which case it repeats the question. If the answer is received in time, with probability $1 - p$, then the new host can restart selecting a new address again. The protocol requires that n questions must be asked if no answer arrived. If after n tries the host didn't get an answer, then it will erroneously consider its IP as new.

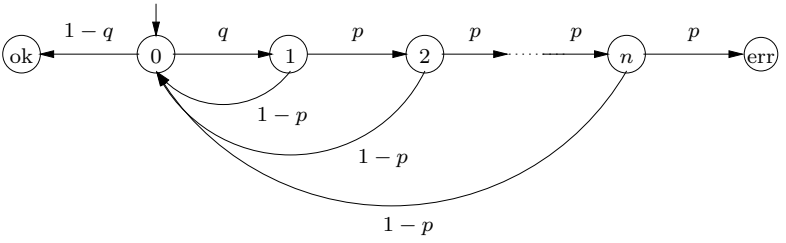


Fig. 5. Model of the zeroconf protocol

$p \setminus q$	0.1	0.2	0.4	0.7	0.8	0.9
0.1	✓	✓	✓	✓	✓	✓
0.2	✓	✓	✓	✓		
0.3	✓	✓				
0.5						
0.7						

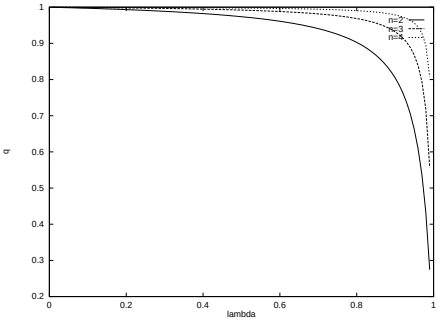


Fig. 6. Parametric analysis of the Zeroconf protocol

We are interested in the probability with which a correct new address will be selected, that is the probability P_{ok} for reaching s_{ok} from s_0. In order to compute it, we consider the language recognized by the automaton with initial state s_0 and final state s_{ok}. The regular expression for it, is $r_{ok} = (q.(1 - p)(1|p|p.p| \ldots |p^n))^*.(1 - q)$. By evaluating this regular expression, and after a simple algebraic simplification, we obtain an analytic expression of P_{ok}:

$$P_{ok} = \frac{1-q}{1-q(1-p)\sum_{k=0}^{n} p^k} = \frac{1-q}{1-q(1-p)\frac{1-p^{n+1}}{1-p}} = \frac{1-q}{1-q(1-p^{n+1})}$$

We want the system to ensure that the new host will get a valid address with probability at least λ, i.e., that it satisfies $s_0 \models \mathcal{P}_{\geq\lambda}(true \mathsf{U} ok)$. This is equivalent to $P_{ok} \geq \lambda$. The table below (left) shows the results of parametric model checking for $\lambda = 0.999$, $n = 4$ and different values of parameters p and q. The graph below (right) plots the maximal value of q ensuring that the property holds, for $p = 0.3$ and $n = 2$, 3 and 4, in function of the probability threshold λ.

6 Conclusions

We presented a new language-theoretic approach to *symbolic probabilistic model checking* of PCTL over DTMCs. It is based on representing the probability

measure of the set of paths satisfying a path formula as a regular expression, computed with the state elimination or inductive algorithms, for the language recognized by a finite-state automaton derived from a DTMC and a PCTL formula, where the alphabet is the set of strictly positive transition probabilities of the DTMC. When these are rational, the probability measure is evaluated to its exact rational value, whereas when transition probabilities are left unspecified as parameters, it yields a rational function on them, which can be used for parametric model checking of the system.

Although the symbolic approach cannot compete with advanced numerical methods in terms of efficiency, we believe that it has some important advantages. Besides allowing for parametric analysis as illustrated in the examples, our approach could be used to generate *"counter-examples"* violating a property, an important feature lacking in probabilistic model checking. For instance, any subterm of a regular expression whose evaluation is bigger than a threshold can be viewed as a counter-example for a property stating that the probability must be less than this threshold.

The only related result on symbolic model checking for parameterized DTMCs we are aware of is [1]. Their method consists in computing strongly connected components and then reduce a Markov chain to a DAG corresponding to its transient behaviour. Unfortunately, no algorithm is provided, and their description does not give any insight into how to obtain the probability matrix of the DAG, a step not trivial in our view. This missing step could boil down to something similar to our method, but we believe the latter to be more intuitive, precise and clear from an algorithmic point of view. Moreover, the technique of [1] cannot deal with irreducible Markov chains, that is Markov chains which are a strongly connected component.

We plan to implement our approach to model-check PCTL formulas without nested probabilistic operators for both the parametric and the non-parametric case, using the state-elimination algorithm for computing regular expressions. In our opinion, formulas with nested probabilistic operators are never or hardly necessary to specify probabilistic properties of practical interest. Moreover, these formulas are also problematic when using iterative numerical methods, since the numerical error introduced in a probabilistic subformula can yield that a path formula is satisfied with probability 1 when it is actually satisfied with probability 0, or vice-versa.

The state-elimination algorithm is the language-theoretic counterpart of Gaussian elimination for solving linear equation systems. Full PCTL could be considered in the non-parametric case, using the inductive algorithm, but since it consists in filling-in a matrix with new entries, we can expect it to have serious limitations to cope with large systems. It will be important to compare our approach based on regular expressions with symbolic methods to solve systems of linear equations based on matrix inversion or Gaussian elimination, as implemented in several computer algebra systems. In order to cope with large systems, reduction techniques based on simulation [9, 10] can be applied, yielding a *sym-*

bolic upper or lower bound for probabilistic reachability properties on a reduced state space.

As future work, we are interested in extending the method to Markov decission processes (MDPs), that is, probabilistic systems with non-determinism, necessary to model compositionally complex systems. One can see an MDP as a parametric DTMC where the non-deterministic choice is replaced by a parametric probabilistic one, such that all but one of these parameters are equal to zero, and then apply our technique. However, this will require a number of evaluations exponential in the number of non-deterministic choices in the worst case. Heuristics to reduce this number, or an alternative approach, are thus necessary. A possible solution could be to consider high-level specification languages like process algebras with iteration [2], parallel composition and communication, and to device a linearization algorithm of such specifications with respect to language equivalence (process algebras with iteration are strictly more expressive with respect to bisimulation in the presence of parallel composition [3]) without building the corresponding automata, for instance using the concept of derivatives of regular expressions [7].

Acknowledgments. We are grateful to Joost-Pieter Katoen for his comments on an early version of this paper, Wan Fokkink for his encouragement, and Ryszard Janicki for helping clarify our results.

References

1. A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939 of *LNCS*, pages 155–165, Liege, Belgium, 1995. Springer Verlag.
2. J. Bergstra, W. Fokkink, and A. Ponse. Process algebra with recursive operations. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebrea*. Elsevier Science, 2001.
3. J. A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration and nesting. *The Computer Journal*, 37(4):243–258, 1994.
4. J. Berstel and C. Reutenauer. *Rational Series and Their Languages*. EATCS Monographs in Computer Science. Springer-Verlag, 1988.
5. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*, pages 499–513. Springer-Verlag, 1995.
6. H. Bohnenkamp, P. van der Stok, H. Hermanns, and F. Vaandrager. Cost-optimization of the IPv4 zeroconf protocol. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 531–540. IEEE Computer Society, June 2003.
7. J. Brzozowsky. Derivatives of regular expressions. *Journal of ACM*, 11(4), 1964.
8. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

9. P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. In L. de Alfaro and S. Gilmore, editors, *Proceedings of Process Algebra and Probabilistic Methods. Performance Modeling and Verification. Joint International Workshop, PAPM-PROBMIV 2001,* Aachen, Germany, volume 2165 of *Lecture Notes in Computer Science*, pages 29–56. Springer-Verlag, 2001.
10. P. D'Argenio, B. Jeannet, H. Jensen, and K. Larsen. Reduction and refinement strategies for probabilistic analysis. In H. Hermanns and R. Segala, editors, *Proceedings of Process Algebra and Probabilistic Methods. Performance Modeling and Verification. Joint International Workshop, PAPM-PROBMIV 2002,* Copenhagen, Denmark, volume 2399 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
11. M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988.
12. M. J. C. Gordon and T. F. Melham. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.
13. Gramond and Rodger. Using JFLAP to interact with theorems in automata theory. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 31, 1999.
14. H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
15. J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman, Reading, Massachusetts, 2000.
16. J. Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.
17. B. Jeannet, P. D'Argenio, and K. Larsen. RAPTURE: A tool for verifying Markov Decision Processes. In I. Cerna, editor, *Tools Day'02,* Brno, Czech Republic, Technical Report. Faculty of Informatics, Masaryk University Brno, 2002.
18. JFLAP (java formal languages and automata package) web page. `http://www.cs.duke.edu/~rodger/tools/jflap/`.
19. J. Kemeny, J. Snell, and A. Knapp. *Denumerable Markov Chains*. Graduate Texts in Mathematics. Springer, 2nd edition, 1976.
20. D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, New York, 1976.
21. M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In J. B. T. Field, P. Harrison and U. Harder, editors, *Proc. Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.
22. PRISM web page. `http://www.cs.bham.ac.uk/~dxp/prism/`.

Verifying Linear Duration Constraints of Timed Automata

Pham Hong Thai and Dang Van Hung

United Nations University,
International Institute for Software Technology,
P. O. Box 3058, Macau
{dvh, pht}@iist.unu.edu

Abstract. This paper aims at developing a technique for checking if a timed automaton satisfies a linear duration constraint on the automaton states. The constraints are represented in the form of linear duration invariants - a simple class of chop-free Duration Calculus (DC) formulas. We prove that linear duration invariants of timed automata are discretisable, and reduce checking if a timed automaton satisfies a linear duration invariant to checking if the integer timed region graph of the original automaton satisfies the same linear duration invariant. The latter can be done with exhaustive search on graphs. In comparison to the techniques in the literature, our method is more powerful: it works for the standard semantics of DC and the class of the closed timed automata while the others cannot be applied.

1 Introduction

Constraints on the durations of system states form a class of important properties of real-time systems. They can be formalised by a class of simple chop-free Duration Calculus formulas of the form $A \leq \ell \leq B \Rightarrow \sum_{s \in S} c_s \int s \leq M$. This class was first introduced with the name *linear duration invariants* and investigated in [14]. The duration of a state s is a mapping from time intervals to reals and is denoted by $\int s$. $\int s$, when applied to an observation time interval $[b, e]$ is the accumulated time for the presence of state s over $[b, e]$; and the term ℓ when applied to an observation time interval $[b, e]$ returns the length $e - b$ of the interval. A linear duration invariant $A \leq \ell \leq B \Rightarrow \sum_{s \in S} c_s \int s \leq M$ simply says that for any observation time interval $[b, e]$, if the length ℓ of the interval satisfies the constraint $A \leq \ell \leq B$ then the durations of the system states over that interval should satisfy the constraint $\sum_{s \in S} c_s \int s \leq M$. A desired property for a simple gas burner "for any observation interval that is longer than 60 seconds, the ratio between the duration of the state *leak* and the length of the interval should not be more than 5%" is represented as a linear duration invariant $\ell \geq 60 \Rightarrow \int leak \leq 5\% * (\int leak + \int nonleak)$ (here we have used the equation $\int leak + \int nonleak = \ell$). A system safety saying that an unsafe state s should not occur, can also be represented by a linear duration invariant

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 295–309, 2005.
© Springer-Verlag Berlin Heidelberg 2005

as $\ell \geq 0 \Rightarrow \int s \leq 0$. The relative fairness of two processes p_1 and p_2 can be represented by two linear duration invariants $\ell \geq 0 \Rightarrow \int p_1.run - \int p_2.run \leq 1$ and $\ell \geq 0 \Rightarrow \int p_2.run - \int p_1.run \leq 1$. This says that the running time of processes p_1 and p_2 are almost the same for any observation interval.

Since timed automata are good models of real-time systems, and since linear duration invariants are important properties of real-time systems, it is interesting if verifying a linear duration invariant of a timed automata can be done automatically. In fact, this problem has attracted a great deal of attention during last decade, since the introduction of Duration Calculus in [13]. Many algorithms have been proposed in the literatures, but all of them have high complexity and do not work for the general case. Some restrictions are needed either on timed automata, or on linear duration invariants or on the meaning of the satisfaction of linear duration invariants by automata in order for those algorithms to apply.

For example, in [5], a solution for checking a LDP of a timed automaton LDI is given using mixed integer and linear programming techniques. The authors have to put restrictions on both linear duration invariants and the meaning of satisfaction: the premise of LDIs should be true, i.e. there is no constraint on the length of the observation intervals, the coefficients in LDIs should be integral, and the observation intervals should start at time 0. In [14], a nice solution to the problem is given using linear programming (techniques) only, but the authors had to restrict themselves on the *real-time* automata, i.e. timed automata with one clock which is reset by every transition. This solution is generalised in [7] and in [9] to a wider subclass of timed automata, but still cannot be used for the whole class of timed automata, and the restriction on the meaning of satisfaction still applies. In [3] the authors considered checking LDI for timed automata with observation intervals started at time 0 only, which is a restriction on the meaning of satisfaction. In general, these algorithms are based on symbolic representation of the behavior of the systems by extended time regular expressions, and hence, reduce the problem to a number of linear programming problems. In practice, the number of linear programming problems which have to be solved is very large, so the time complexity of these algorithms is very high.

For reducing the complexity of the problem, some other papers use a different approach. The authors of these papers consider those properties which are discretisable, i.e. they are satisfied by all the behaviours of a timed automata if and only if they are satisfied by all integral behaviors only. That means, we can check such kind of properties of a timed automaton by exploring the integral region graph of the automaton as in [12]. This technique is combined with linear programming technique in [8] for checking some other classes of discretisable properties. In these papers, the authors also had to enforce some restrictions on linear duration invariants and on observation intervals.

In this paper, we study if we can remove the restrictions mentioned as above and develop a general technique to solve the problem for the general case. The idea on discretisability of LDP in [12] is the motivation for the discretisability of LDIs in this paper. We prove that LDI is also a discretisable property for timed automata. However, the discretisability of LDIs is used in this paper in a

manner that is different from the one in [12]. In the following, we call a LDI having the premise equivalent to "true" (i.e. the premise can be removed) a linear duration property (LDP). In [12] the discretisability of LDP is used to reduce a region graph to an integral region graph, but in this paper the discretisability of LDI is used to approximate a real-time interval by integral-time interval as well.

Our results are summarised as follows. We first define the different semantics for the satisfaction of a LDI by a timed automaton. We do this by introducing the different classes of Duration Calculus models defined by a timed automaton $\mathcal{A}$: $\mathcal{M}_0(\mathcal{A})$ is the set of DC models generated by $\mathcal{A}$ with the observation intervals of the form $[0, t]$, where t is a non negative real; $\mathcal{M}(\mathcal{A})$ is the set of DC models generated by $\mathcal{A}$ with no restriction on the observation intervals; $\mathcal{M}_{uv}(\mathcal{A})$ is the set of DC models generated by $\mathcal{A}$ with the observation intervals of the form $[t_p, t_q]$, where t_p and t_q are the times the automaton enters states p and q, respectively, in the corresponding behaviour; and $\mathcal{M}_I(\mathcal{A})$ is the set of DC models corresponding to the integral behaviours $\mathcal{A}$ with the integral observation intervals. Then, we prove that given a LDI D, $\mathcal{M}(\mathcal{A}) \models D$ if and only if $\mathcal{M}_I(\mathcal{A}) \models D$. Based on these results we reduce the problem of checking $\mathcal{M}(\mathcal{A}) \models P$ to the one of checking $p \models P$ for all paths p in the region graph of $\mathcal{A}$, and we reduce the problem of checking $\mathcal{M}(\mathcal{A}) \models D$ to the one of checking $p \models D$ for all path p in the integral region graph of $\mathcal{A}$. The resulting problem can be solved by standard exhaustive search techniques.

The paper is organized as follows. In the next section we recall some basic notions of timed automata and Duration Calculus formulas. In Section 3 we prove the discretisability of LDIs for timed automata. In Section 4, we propose an algorithm for checking a LDI of a timed automata by searching on the weighted graph constructed from the integral region graph of the automaton. Finally, Section 5 is the conclusion of this paper.

2 Preliminaries

In this section, we recall some notions that will play the basic role in defining the problem in this paper. They are timed automata, region graphs, and Duration Calculus formulas in the form of Linear Duration Invariants (LDI).

2.1 Timed Automata

Timed automata was introduced in [1, 2] as formal models for real-time systems. Here we only give a brief description for timed automata and their behavior. Readers are referred to [1] for their more details. As usual, we denote by $\mathbf{R}^+$ and $\mathbf{N}$ the sets of nonnegative real numbers and natural numbers, respectively.

For a finite set of clock variables X, let $\Phi(X)$ be the set of clock constraints on X, which are conjunctions of the formulas of the form $x \leq c$ or $c \leq x$, where $x \in X$ and $c \in \mathbf{N}$. A timed automaton is a finite state machine equipped with the set of clock variables X, and is defined as follows.

Definition 1. *A timed automaton $\mathcal{A}$ is a tuple $\langle L, s_0, \Sigma, X, E, I\rangle$, where*

- *L is a finite set of locations,*
- *$s_0 \in L$ is an initial location,*
- *Σ is a finite set of symbols (action names),*
- *X is a finite set of clocks,*
- *I is a mapping that assigns to each location $s \in L$ a clock constraint $I(s) \in \Phi(X)$ which is called invariant of location s. Intuitively, the timed automaton only stays at s when the values of the clocks satisfy the invariant $I(s)$.*
- *$E \subseteq L \times \Phi(X) \times \Sigma \times 2^X \times L$ is a set of switches. A switch $\langle s, \varphi, a, \lambda, s'\rangle$ represents a transition from location s to location s' with symbol a, where φ is a clock constraint over X that specifies the enabling condition of the switch, and $\lambda \subseteq X$ gives the set of clocks to be reset with this switch.*

For convenient to our method, here we consider the class of the closed timed automata, i.e. timed automata that do not include clock constraints of the form $x < c$ or $c < x$.

A clock interpretation ν for the set of clocks X is a mapping that assigns a nonnegative real value to each clock. For $\delta \in \mathbf{R}$, let $\nu + \delta$ denote the clock interpretation which maps every clock $x \in X$ to the value $\nu(x) + \delta$. For $\lambda \subseteq X$, let $\nu[\lambda := 0]$ denote the clock interpretation which assigns 0 to each $x \in \lambda$ and agrees with ν over the rest of the clocks.

A state of automaton $\mathcal{A}$ is a pair (s, ν) where s is a location of $\mathcal{A}$ and ν is a clock interpretation which satisfies invariant $I(s)$. State (s_0, ν_0) is the initial state where s_0 is the initial location of $\mathcal{A}$ and ν_0 is the clock interpretation for which $\nu_0(x) = 0$ for all clocks x.

A transition of $\mathcal{A}$ can:

- change state by letting time elapse: For a state (s, ν) and a real-valued time increment $\delta \geq 0$, $(s, \nu) \xrightarrow{\delta} (s, \nu + \delta)$ if for all $0 \leq \delta' \leq \delta$, $\nu + \delta'$ satisfies invariant $I(s)$.
- change state by taking a location switch : For a state (s, ν) and a switch $\langle s, \varphi, a, \lambda, s'\rangle$ such that ν satisfies φ and $\nu[\lambda := 0]$ satisfies $I(s')$ then $(s, \nu) \xrightarrow{a} (s', \nu[\lambda := 0])$.

A time elapsing transition and a following location switching transition can be combined into one transition and denoted by $(s, \nu) \xrightarrow{\delta, a} (s', \nu')$. That means the system stays at location s with the current clock interpretation ν, after δ time units, the clock interpretation $\nu + \delta$ satisfies the enabling condition (time constraint) φ of switch $e = \langle s, \varphi, a, \lambda, s'\rangle$, and the system transits to location s' by taking e with label a and resets the clocks in the λ to 0, and the new state of the system is (s', ν').

In this paper we consider only *nonZeno* behaviours of automata ([1, 2]), that is those behaviours for which in any finite time interval there is only a finite number of transition occurrences.

Example 1. The timed automaton in Figure 1, taken from [4], have two clocks x and y. The set of locations is $\{s_0, s_1\}$, where invariants are $I(s_0) = (y \leq 5)$ and $I(s_1) = (x \leq 8 \wedge y \leq 10)$. The set of switches is

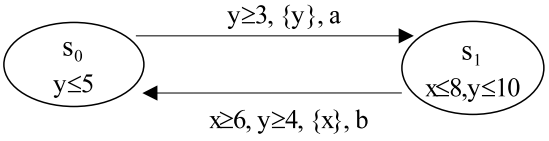


Fig. 1. A timed automaton

$\{\langle s_0, y \geq 3, a, \{y\}, s_1\rangle, \langle s_1, x \geq 6 \wedge y \geq 4, b, \{x\}, s_0\rangle\}$. The clock y is reset to 0 each time the automaton transits from s_0 to s_1 and the clock x is reset to 0 when the automaton transits from s_1 to s_0.

Definition 2. *Let $\mathcal{A}$ be a timed automaton.*

1. *A* run *or an* execution *r of $\mathcal{A}$ is an infinite sequence of state transitions: $(s_0, \nu_0) \overset{\delta_1, a_1}{\rightarrow} (s_1, \nu_1) \overset{\delta_2, a_2}{\rightarrow} \ldots$, where (s_0, ν_0) is an initial state of $\mathcal{A}$.*
2. *A* behavior *ρ correspondents to above run r, is the infinite sequence of timed locations*

$$\rho : (s_0, t_0)(s_1, t_1) \ldots (s_m, t_m) \ldots$$

that satisfies the following conditions
- *$t_0 = 0$.*
- *for any $T \in \mathbf{R}$, there is some $i \geq 0$ such that $t_i \geq T$.*
- *t_i is the moment that system enters to s_i, for all $i \geq 0$. That means, $\delta_i = t_i - t_{i-1}$ and $\mathcal{A}$ stays in state s_{i-1} for $t_i - t_{i-1}$ time units and then transits to s_i in the run r.*

Note that in this paper, a behavior of a timed automaton is a sequence of time stamped locations instead of a sequence of timed stamped switches as in other papers. However the semantics of timed automata is not changed. This way of representation of behaviours shows the DC models generated by them more explicitly. A run or a behavior is said to be *integral* iff for all $i \geq 0$, the values of clock variables in ν_i, the time delay δ_i, and time stamps t_i are integral.

2.2 Linear Duration Invariants and Duration Properties

Models in Duration Calculus. Duration Calculus (DC) was introduced by Zhou Chaochen et al. in [13] as a logic to reason about the state duration of real-time systems. A comprehensive introduction to *DC* can be found in the recent monograph [15]. In DC, a state is viewed as a boolean-valued function of the continuous time that has the value true (denoted by 1) at time t iff the state is present at t. Otherwise it takes the value 0. An interpretation $\mathcal{I}$ of the system is an assignment that assigns to each system state s a boolean-valued function $\mathcal{I}_s$. A DC model consists an interpretation $\mathcal{I}$ and a time interval $[b, e]$.

It represents an observation of the behavior of the system states in an interval of time $[b, e]$. Given an interpretation $\mathcal{I}$, the duration of a state s over time interval $[b, e]$ is defined as $\int_b^e \mathcal{I}_s(t)dt$, which is exactly the accumulated present time of s in the interval $[b, e]$ by the interpretation $\mathcal{I}$.

In this paper we consider the set of DC models that express all the observations of the behaviours of a timed automaton. Each behavior $\rho = (s_0, t_0)(s_1, t_1)(s_2, t_2)\dots$ of timed automaton $\mathcal{A}$ defines uniquely an interpretation $\mathcal{I}$ in DC by: for any $s \in L$, $\mathcal{I}_s(t) = 1$ iff $\exists i \bullet (s_i = s \wedge t \in [t_i, t_{i+1}))$. We also denote such $\mathcal{I}$ by $(\overline{s}, \overline{t})$ in which $\overline{s} = (s_0, s_1, \dots)$ and $\overline{t} = (t_0, t_1, \dots)$ express a sequence of s_i and t_i from behaviour ρ. Hence, $(\overline{s}, \overline{t}, [b, e])$ is a DC model representing the observation of ρ in the interval $[b, e]$, which is an observation of the timed automaton $\mathcal{A}$ over interval $[b, e]$. We also call $(\overline{s}, \overline{t}, [b, e])$ a DC model of $\mathcal{A}$.

Let $\mathcal{M}(\mathcal{A})$ denote set of DC models of $\mathcal{A}$. To cope with the different meanings of the satisfaction of a DC formula by $\mathcal{A}$ as said in the introduction of the paper, we introduced the following classes of DC models of $\mathcal{A}$:

$$\begin{aligned}
\mathcal{M}_0(\mathcal{A}) &= \{\sigma \mid \sigma = (\overline{s}, \overline{t}, [0, T]) \in \mathcal{M}(\mathcal{A}),\ T \geq 0\},\\
\mathcal{M}_{uv}(\mathcal{A}) &= \{\sigma \mid \sigma = (\overline{s}, \overline{t}, [t_u, t_v]) \in \mathcal{M}(\mathcal{A}),\ t_u, t_v \text{ occur in } \overline{t} \text{ and } t_u \leq t_v\},\\
\mathcal{M}_I(\mathcal{A}) &= \{\sigma \mid \sigma = (\overline{s}, \overline{t}, [b, e]) \in \mathcal{M}(\mathcal{A}) \text{ and } t_i, b, e \in \mathbf{N}, \forall i \geq 0\}.
\end{aligned}$$

In the other word, $\mathcal{M}_0(\mathcal{A})$ is the set of models representing the observations starting from 0 and ending at any time point. $\mathcal{M}_{uv}(\mathcal{A})$ consists of models that representing the observations starting and ending at those time points at which the automaton transits to a location, i.e. the observations between two location switching transitions. A DC model of $\mathcal{A}$ in $\mathcal{M}_I(\mathcal{A})$ represents an observation of an integral behavior of $\mathcal{A}$ (i.e behavior in which transitions take place only at integer time) from an integer time point to an integer time point.

Linear Duration Properties and Linear Duration Invariants. Given a timed automaton $\mathcal{A} = \langle L, s_0, \Sigma, X, E, I\rangle$. A linear duration invariant over L is a DC formula of the form

$$\mathcal{D} : A \leq \ell \leq B \Rightarrow \sum_{s \in L} c_s \int s \leq M.$$

where c_s, A, B and M are real numbers, $A \leq B$ (B may be ∞). In $\mathcal{D}$ DC term $\int s$ is a duration term denoting the duration of location s, and ℓ is a DC term denoting the interval length. LDI $\mathcal{D}$ evaluates over a DC model $(\mathcal{I}, [b, e])$ as tt and denoted by $(\mathcal{I}, [b, e]) \models \mathcal{D}$ iff $A \leq e - b \leq B \Rightarrow \sum_{s \in L} c_s \int_b^e \mathcal{I}_s(t)dt \leq M$ evaluates to true (in the predicate calculus). Here we define the satisfaction of $\mathcal{D}$ by $\mathcal{A}$ directly on the behaviours of $\mathcal{A}$ as follows.

Definition 3. *Let $\mathcal{D}$ be a LDI as above. For each $\sigma = (\overline{s}, \overline{t}, [b, e]) \in \mathcal{M}(\mathcal{A})$ we define $l(\sigma)$ and $\theta(\sigma)$ as*

$$l(\sigma) = e - b \text{ and } \theta(\sigma) = \sum_{s \in L} c_s P_s$$

where P_s is the accumulated time for the presence of location s in the interval $[b, e]$ and is calculated as follows. Let u and v be the indexes in $\bar{t}$ such that $t_{u-1} < b \leq t_u$ and $t_v \leq e < t_{v+1}$. For $s \neq s_v$ and $s \neq s_{u-1}$, $P_s = \sum_{u \leq j \leq v-1 \wedge s_j = s}(t_{j+1} - t_j)$. $P_{s_{u-1}} = \sum_{u \leq j \leq v-1 \wedge s_j = s_{u-1}}(t_{j+1} - t_j) + (t_u - b)$, and $P_{s_v} = \sum_{u \leq j \leq v-1 \wedge s_j = s_v}(t_{j+1} - t_j) + (e - t_v)$.

Hence, $\theta(\sigma)$ evaluates over model $\sigma = (\bar{s}, \bar{t}, [b, e])$ as

$$\theta(\sigma) = c_{s_{u-1}}(t_u - b) + \sum_{u \leq j \leq v-1 \wedge s_j = s} c_s(t_{j+1} - t_j) + c_{s_v}(e - t_v) \tag{1}$$

By expanding the sum and letting t_i's be common factors, we have

$$\theta(\sigma) = \sum_{i=u}^{v} a_i t_i + c_{s_v} e - c_{s_{u-1}} b \tag{2}$$

where a_i's are real numbers that are derivable from c_s's.

Definition 4. *Given a LDI $\mathcal{D}$.*

- *A DC model $\sigma = (\mathcal{I}, [b, e]) \in \mathcal{M}(\mathcal{A})$ is said to satisfy $\mathcal{D}$, denoted by $\sigma \models \mathcal{D}$, iff $A \leq l(\sigma) \leq B$ implies $\theta(\sigma) \leq M$.*
- *Timed automaton $\mathcal{A}$ is said to satisfy $\mathcal{D}$, denoted by $\mathcal{A} \models \mathcal{D}$, iff $\sigma \models \mathcal{D}$, for all $\sigma \in \mathcal{M}(\mathcal{A})$.*

When a LDI $\mathcal{D}$ has the premise equivalent to true, i.e. equivalent to $0 \leq \ell \leq \infty$, we say that $\mathcal{D}$ is a linear duration property (LDP) ([12]). So, a LDP is a special LDI which do not have premise. Hence, checking a LDP is normally simpler than checking a LDI.

Similarly, for any class $\mathcal{M}_x(\mathcal{A})$, $x \in \{uv, I, 0\}$, we define $\mathcal{M}_x(\mathcal{A}) \models \mathcal{D}$ iff $\sigma \models \mathcal{D}$ for all $\sigma \in \mathcal{M}_x(\mathcal{A})$.

The model-checking problem in this paper is formulated as: given a timed automaton $\mathcal{A} = \langle L, s_0, \Sigma, X, E, I \rangle$, given a LDI $\mathcal{D}$ over L; find an algorithm to decide whether $\mathcal{A} \models D$.

3 Discretisability of Linear Duration Invariants with Respect to Timed Automata

3.1 ϵ-Digitising

The concept of ϵ-digitising was first introduced in [6]. We recall here the definition of ϵ-digitisation given by them.

Definition 5. *Given a positive real x and ϵ, $(0 \leq \epsilon < 1)$. Let x_ϵ be an integer defined as*

$$x_\epsilon = \begin{cases} \lfloor x \rfloor & \text{if fraction of } x \text{ is less than or equal to } \epsilon \\ \lceil x \rceil & \text{otherwise} \end{cases}$$

The number x_ϵ is called ϵ-digitisation of x.

Some properties of ϵ-digitisation needed in the development of our techniques are listed in the following lemmas. Proving of these lemmas is easy so reader be referred [10].

Lemma 1. *Given two integer numbers $a \leq b$, given two nonnegative real numbers $t_i \geq t_j$. Then for all $\epsilon \in [0,1)$ we have*

$$a \leq t_i - t_j \leq b \Leftrightarrow a \leq t_{i\epsilon} - t_{j\epsilon} \leq b\,.$$

As a consequence of the lemma, if $t_i \geq t_j$ then $t_{i\epsilon} \geq t_{j\epsilon}$ for all $\epsilon \in [0,1)$ (apply the lemma with $a = 0$). This means that under the ϵ-digitisation, the order of a sorted sequence of real numbers is unchanged.

Lemma 2. *Let $\{a_i\}$, $\{t_i\}$, $(i = 1..m)$ be two sequences of real numbers, where $t_i \geq 0$ for all $i = 1, \ldots, m$. Then we can always find a real number $\epsilon \in [0,1)$ such that*

$$\sum_{i=1}^{m} a_i t_i \leq \sum_{i=1}^{m} a_i t_{i\epsilon}$$

Lemma 3. *Let $\sigma = (\overline{s}, \overline{t}, [b,e]) \in \mathcal{M}(\mathcal{A})$ be a DC model of timed automaton $\mathcal{A}$. Let $\overline{s} = s_0, s_1, \ldots$; $\overline{t} = t_0, t_1 \ldots$ and $t_{u-1} < b \leq t_u$, $t_v \leq e < t_{v+1}$. Then for all $\epsilon \in [0,1)$, $\sigma_\epsilon = (\overline{s}, \overline{t}_\epsilon, [b_\epsilon, e_\epsilon])$ is an integral model of $\mathcal{A}$, i.e. $\sigma_\epsilon \in \mathcal{M}_I(\mathcal{A})$, where $\overline{t}_\epsilon = t_{0\epsilon}, t_{1\epsilon}, \ldots$.*

3.2 Discretisability of LDI

Definition 6. *Given a timed automaton $\mathcal{A}$ and a linear duration invariant $\mathcal{D}$. $\mathcal{D}$ is said to be* discretisable *with respect to $\mathcal{A}$ if $\mathcal{A} \models \mathcal{D}$ exactly when $\mathcal{M}_I(\mathcal{A}) \models \mathcal{D}$.*

Theorem 1. *Any linear duration invariant $\mathcal{D}$ which has the premise $A \leq \ell \leq B$ in which A and B are integral, is discretisable with respect to timed automaton $\mathcal{A}$ (here we consider ∞ as an integer by our convention).*

Proof. We have to prove that $\mathcal{M}(\mathcal{A}) \models \mathcal{D} \Leftrightarrow \mathcal{M}_I(\mathcal{A}) \models \mathcal{D}$.

The "only if" part is obvious because $\mathcal{M}_I(\mathcal{A}) \subseteq \mathcal{M}(\mathcal{A})$.

The "if" part is proved as follows. Let $\sigma \in \mathcal{M}(\mathcal{A})$ such that $\sigma \not\models \mathcal{D}$. We prove that there exists $\epsilon \in [0,1)$ such that $\sigma_\epsilon \not\models \mathcal{D}$, where σ_ϵ is the digitisation of σ w.r.t. ϵ.

Assume that $\sigma = (\overline{s}, \overline{t}, [b,e])$ with $\overline{s} = s_0, s_1, \ldots$ and $\overline{t} = t_0, t_1, \ldots$. Let indexes u and v be such that $t_{u-1} < b \leq t_u$, $t_v \leq e < t_{v+1}$. $\sigma \not\models \mathcal{D}$ implies that $A \leq e - b \leq B$ and $\theta(\sigma) > M$. By the definition of LDI, it follows from equation (2):

$$\theta(\sigma) = \sum_{i=u}^{v} a_i t_i + c_{s_v} e - c_{s_{u-1}} b > M$$

From Lemma 2 (note that coefficients a_i's in the lemma are any reals), $\exists \epsilon \in [0,1)$ such that $\sum_{i=u}^{v} a_i t_{i\epsilon} + c_{s_v} e_\epsilon - c_{s_{u-1}} b_\epsilon \geq \theta(\sigma) > M$. By Lemma 1 it follows from $A \leq e - b \leq B$ that $A \leq e_\epsilon - b_\epsilon \leq B$ (notice that A, B are integers). By Lemma 3 σ_ϵ is an integral DC model of $\mathcal{A}$, and $\theta(\sigma_\epsilon) = \sum_{i=u}^{v} a_i t_{i\epsilon} + c_{s_v} e_\epsilon - c_{s_{u-1}} b_\epsilon$. Hence, $\theta(\sigma_\epsilon) > M$.

Thus, we have obtained an integral model σ_ϵ which does not satisfy $\mathcal{D}$.

Now that the assumption that two integral bounds A and B in the premise of LDIs is not too restricted, and the result can be extended to the case that A and B are rationals using the well-known technique.

From this theorem, from now on we will consider only the integral DC models of timed automaton $\mathcal{A}$, i.e. models $\sigma = (\overline{s}, \overline{t}, [b, e]) \in \mathcal{M}_I(\mathcal{A})$.

4 Checking Linear Duration Invariants of Timed Automata with Graph Search

4.1 Integral Reachability Graph of Timed Automata

In this section, we shortly recall about integral region graph which is a part of region graph. Region graph was presented by Alur and Dill in ([1]) and has become well-known.

Let K_x be the largest constant compared with the clock $x \in X$ in the time constraints and the invariants of $\mathcal{A}$ and let $K = \max\{K_x \,|\, x \in X\} + 1$. An equivalence relation restricted into the set of all integral clock interpretations of $\mathcal{A}$ is defined as follows. Let ν_1, ν_2 be two integer clock interpretations. We say that ν_1 is equivalent to ν_2 and denoted by $\nu_1 \cong \nu_2$ iff for all $x \in X$ either $\nu_1(x) = \nu_2(x)$ or $\nu_1(x) \geq K_x + 1 \ \wedge \ \nu_2(x) \geq K_x + 1$. The equivalence class containing ν is denoted by $[\nu]$ and is called integral clock region. It is easy to see that number of integral clock regions is bounded by $(K+1)^k$ (k is number of clocks).

The equivalence relation $\cong$ is also extended to an equivalence relation on state space of timed automata. we call two states $q_1 = (s_1, \nu_1)$ and $q_2 = (s_2, \nu_2)$ of timed automaton $\mathcal{A}$ be region-equivalent (denoted by $q_1 \equiv q_2$) iff $\nu_1 \cong \nu_2$ and $s_1 = s_2$. The equivalence relation $\equiv$ partitions space of states of $\mathcal{A}$ into classes of states, each class is characterized by a couple of a location s and a clock region π and is denoted by $\langle s, \pi \rangle$. We also call $\langle s, \pi \rangle$ a region. It is obvious that the number of regions is bounded by $|L|(K+1)^k$.

A region $\langle s', [\nu'] \rangle$ is called be successor of $\langle s, [\nu] \rangle$ if $\exists d \geq 0$ and an transition $e = \langle s, \varphi, a, \lambda, s' \rangle$ such that $(s, \nu) \stackrel{d,a}{\rightarrow} (s', \nu')$. Then we write $\langle s, [\nu] \rangle \stackrel{d,a}{\rightarrow} \langle s', [\nu'] \rangle$

We can easily prove the following lemma.

Lemma 4. *If* $(s, \nu) \stackrel{d,a}{\rightarrow} (s', \nu')$ *then* $\langle s, [\nu] \rangle \stackrel{d,a}{\rightarrow} \langle s', [\nu'] \rangle$*, and reversely, if* $\langle s, \pi \rangle \stackrel{d,a}{\rightarrow} \langle s', \pi' \rangle$ *then for each* $\nu \in \pi$*, there exists* $\nu' \in \pi'$ *such that* $(s, \nu) \stackrel{d,a}{\rightarrow} (s', \nu')$.

From the lemma 4, the integral reachability graph $\mathcal{RG} = (\mathsf{V}, \mathsf{E})$ of the timed automaton $\mathcal{A}$ is built as follows. Each vertex $\mathsf{v} \in \mathsf{V}$ is a region $\langle s, \pi \rangle$. E is initialised to $\emptyset$, and V is initialised to $\{\langle s_0, \pi_0 \rangle\}$, where s_0 is initial location of $\mathcal{A}$ and π_0 is region with 0 as the value of all of clocks. Then, V is expanded as follows. If a vertex $\langle s, \pi \rangle \in \mathsf{V}$ has a successor $\langle s', \pi' \rangle$ then $\langle s', \pi' \rangle$ is added into V and $\mathsf{e} = (\langle s, \pi \rangle, \langle s', \pi' \rangle)$ is an edge in E. Besides, each edge e is labelled by an interval $[\mathsf{l}(\mathsf{e}), \mathsf{u}(\mathsf{e})]$, where $\mathsf{l}(\mathsf{e})$ and $\mathsf{u}(\mathsf{e})$ are the minimal and maximal integer time delay that automaton can stay at location s before it transits into location s'. $\mathsf{l}(\mathsf{e})$ and $\mathsf{u}(\mathsf{e})$ are defined as:

$$\mathsf{l}(\mathsf{e}) = \inf \left\{ d \geq 0 \mid d \in \mathbf{N},\ \langle s, \pi \rangle \overset{d,a}{\rightarrow} \langle s', \pi' \rangle \right\},$$
$$\mathsf{u}(\mathsf{e}) = \sup \left\{ d \geq 0 \mid d \in \mathbf{N},\ \langle s, \pi \rangle \overset{d,a}{\rightarrow} \langle s', \pi' \rangle \right\}.$$

From the definition of $\langle s, \pi \rangle$ and $\langle s', \pi' \rangle$, either $\mathsf{l}(\mathsf{e}) = \mathsf{u}(\mathsf{e})$ or $\mathsf{u}(\mathsf{e}) = \infty$. We will denote a labelled edge e by $(\mathsf{v}, \mathsf{v}', [\mathsf{l}, \mathsf{u}])$.

An detailed algorithm was also constructed in [12] and also in [10].

4.2 Relationship Between $\mathcal{M}_{uv}(\mathcal{A}) \cap \mathcal{M}_I(\mathcal{A})$ and Reachability Graph $\mathcal{RG}$ w.r.t. LDI $\mathcal{D}$

As mentioned above, in this section we consider only integral models. The restriction of $\mathcal{M}_{uv}(\mathcal{A})$ and $\mathcal{M}(\mathcal{A})$ on the integral DC models for $\mathcal{A}$ are $\mathcal{M}_{uv}(\mathcal{A})^I \mathrel{\widehat{=}} \mathcal{M}_{uv}(\mathcal{A}) \cap \mathcal{M}_I(\mathcal{A})$ and $\mathcal{M}_I(\mathcal{A})$, respectively.

Let $\mathcal{RG}$ be the reachability graph of $\mathcal{A}$.

Definition 7. *Let $\mathsf{p} = \mathsf{v}_1 \mathsf{v}_2 \ldots \mathsf{v}_m$ be a path in $\mathcal{RG}$, and let $\overline{d} = d_1, d_2, \ldots, d_{m-1}$ be a sequence of integers, where $d_i \in [\mathsf{l}(\mathsf{v}_i, \mathsf{v}_{i+1}), \mathsf{u}(\mathsf{v}_i, \mathsf{v}_{i+1})]$, for $i = 1..m{-}1$. The sequence $\wp = \mathsf{v}_1 d_1 \mathsf{v}_2 d_2 \ldots \mathsf{v}_{m-1} d_{m-1} \mathsf{v}_m$ (written as $\wp = (\mathsf{p}, \overline{d})$ for short) is called* **weighted interpretation** *of p.*

Definition 8

- *Let $\wp = (\mathsf{p}, \overline{d})$ be a weighted interpretation of path p. We define $l(\wp) \mathrel{\widehat{=}} \sum_{i=1}^{m-1} d_i$ and $\theta(\wp) \mathrel{\widehat{=}} \sum_{i=0}^{m-1} c_{\mathsf{v}_i} d_i$ and call them length and cost of $\wp$ respectively, where c_{v_i} is the coefficient c_{s_i} in formula $\mathcal{D}$ when s_i is the location of v_i.*
- *A weighted interpretation $\wp$ is said to satisfy $\mathcal{D}$, denoted by $\wp \models \mathcal{D}$, iff*

$$A \leq l(\wp) \leq B \Rightarrow \theta(\wp) \leq M$$

- *The graph $\mathcal{RG}$ is said to satisfy LDI $\mathcal{D}$ and is denoted by $\mathcal{RG} \models \mathcal{D}$ iff $\wp \models \mathcal{D}$ for all weighted interpretations $\wp$ of $\mathcal{RG}$.*

The following lemma plays a key role for our checking technique.

Lemma 5. *For any DC model $\sigma \in \mathcal{M}_{uv}(\mathcal{A})^I$, there exists a weighted interpretation $\wp$ of $\mathcal{RG}$ such that $l(\sigma) = l(\wp)$ and $\theta(\sigma) = \theta(\wp)$, and vice versa.*

Proof. Let $\sigma = (\overline{s}, \overline{t}, [t_u, t_v]) \in \mathcal{M}_{uv}(\mathcal{A})^I$. Then,

$$\begin{cases} \overline{s} = s_0 \dots s_u \dots s_v \dots , \\ \overline{t} = t_0 \dots t_u \dots t_v \dots , \\ l(\sigma) = t_v - t_u = \sum_{i=u}^{v-1} (t_{i+1} - t_i), \\ \theta(\sigma) = \sum_{i=1}^{m} c_{s_i} \sum_{\substack{j=u \\ s_j = s_i}}^{v-1} (t_{j+1} - t_j). \end{cases}$$

From the definition of model σ, σ corresponds to the sequence of transitions $(s_u, \nu_u) \overset{\delta_u, a_u}{\rightarrow} (s_{u+1}, \nu_{u+1}) \overset{\delta_{u+1}, a_{u+1}}{\rightarrow} \dots \overset{\delta_{v-1}, a_{v-1}}{\rightarrow} (s_v, \nu_v)$, where $\delta_i = t_{i+1} - t_i$, for all $i = u..v-1$. By Lemma 4 we have: $\langle s_u, [\nu_u] \rangle \overset{\delta_u, a_u}{\rightarrow} \langle s_{u+1}, [\nu_{u+1}] \rangle \overset{\delta_{u+1}, a_{u+1}}{\rightarrow} \dots \overset{\delta_{v-1}, a_{v-1}}{\rightarrow} \langle s_v, [\nu_v] \rangle$. Consequently, the weighted interpretation $\wp = (\mathsf{p}, \overline{d})$, where $\mathsf{p} = \langle s_u, [\nu_u] \rangle \langle s_{u+1}, [\nu_{u+1}] \rangle \dots \langle s_v, [\nu_v] \rangle$ and $\overline{d} = \delta_1, \delta_2, \dots, \delta_v$, satisfies the requirement of the lemma, i.e. $l(\wp) = l(\sigma)$, $\theta(\wp) = \theta(\sigma)$.

To prove the reverse direction, assume that $\wp = (\mathsf{p}, \overline{d})$ is a weighted interpretation of $\mathcal{RG}$, where $\mathsf{p} = \mathsf{v}_u \mathsf{v}_{u+1} \dots \mathsf{v}_v$, $\overline{d}$ is a sequence of integers $d_u, d_{u+1} \dots, d_v$, and $\mathsf{v}_i = \langle s, \pi_i \rangle$ for $i = u..v$. Due to the fact that $\mathcal{RG}$ is a reachability graph of $\mathcal{A}$, there exists a sequence of switches e_i $(i = u..v-1)$ such that $\langle s_u, \pi_u \rangle \overset{\delta_u, a_u}{\rightarrow} \langle s_{u+1}, \pi_{u+1} \rangle \overset{\delta_{u+1}, a_{u+1}}{\rightarrow} \dots \overset{\delta_{v-1}, a_{v-1}}{\rightarrow} \langle s_v, \pi_v \rangle$. By Lemma 4 we can find a model $\sigma \in \mathcal{M}_{uv}(\mathcal{A})$, i.e sequence of clock interpretations $\nu_i \in \pi_i$ such that $(s_u, \nu_u) \overset{\delta_u, a_u}{\rightarrow} (s_{u+1}, \nu_{u+1}) \overset{\delta_{u+1}, a_{u+1}}{\rightarrow} \dots \overset{\delta_{v-1}, a_{v-1}}{\rightarrow} (s_v, \nu_v)$. Hence, $l(\sigma) = l(\wp)$ and $\theta(\sigma) = \theta(\wp)$.

This lemma allows us, instead of checking $\mathcal{M}_{uv}(\mathcal{A}) \models \mathcal{D}$, to check $\mathcal{RG} \models \mathcal{D}$ which can be done by using popular searching techniques.

Removing Infinitive Edges

We now give some lemmas to simplify $\mathcal{RG}$ before doing search. Lemmas 6 and 7 say that the label $[\mathsf{l}, \infty)$ of an edge in $\mathcal{RG}$ either makes $\mathcal{RG}$ not satisfy $\mathcal{D}$ or can be replaced by a finite label $[\mathsf{l}, \mathsf{u}]$ without any change to the result of checking $\mathcal{RG} \models \mathcal{D}$. Recall that the premise of LDI $\mathcal{D}$ is $A \leq \ell \leq B$.

Lemma 6. *Assume that* $\mathsf{e} = (\mathsf{v}, \mathsf{v}', [\mathsf{l}, \infty))$ *is an infinite edge of region graph* $\mathcal{RG}$*. Then, if* $B = \infty$ *and* $c_{\mathsf{v}} > 0$ *then* $\mathcal{RG} \not\models \mathcal{D}$*.*

Lemma 7. *Assume that* $\mathsf{e} = (\mathsf{v}, \mathsf{v}', [\mathsf{l}, \infty))$ *is an infinite edge of* $\mathcal{RG}$*. Then label* $[\mathsf{l}, \infty)$ *can be replaced as follows without any change to the result of checking* $\mathcal{RG} \models \mathcal{D}$*.*

- *If* $B = \infty$ *and* $c_{\mathsf{v}} < 0$*, replace* $[\mathsf{l}, \infty)$ *by* $[\mathsf{l}, \mathsf{u}]$ *with* $\mathsf{u} = \max\{\mathsf{l}, A\}$*.*
- *If* $B < \infty$*, replace* $[\mathsf{l}, \infty)$ *by* $[\mathsf{l}, \mathsf{u}]$ *with* $\mathsf{u} = \max\{\mathsf{l}, B\}$*.*

The proof of the above lemmas is simple and is omitted here.

In summary, for checking $\mathcal{M}_{uv}(\mathcal{A}) \models \mathcal{D}$, we can apply the above lemmas first and either we discover $\mathcal{M}_{uv}(\mathcal{A}) \not\models \mathcal{D}$ early or we can convert the infinite edges of $\mathcal{RG}$ into finite ones. From now on, we assume that $\mathcal{RG}$ does not contain infinite edges.

4.3 Weighted Graph for Checking LDI

Similarly to checking LDP ([12]), we can also construct a weighted graph G from the reachability graph $\mathcal{RG}$ (not containing infinite edges) such that $\mathcal{RG} \models \mathcal{D}$ if and only if $\mathsf{G} \models \mathcal{D}$.

The weighted graph $\mathsf{G} = (\mathsf{V}, \mathsf{E}, \omega)$ is constructed from $\mathcal{RG} = (\mathsf{V}_R, \mathsf{E}_R)$ by the following procedure:

Step 1. $\mathsf{V} := \mathsf{V}_R$, $\mathsf{E} := \mathsf{E}_R$.
Step 2. For each edge $\mathsf{e} = ((\mathsf{v}_i, \mathsf{v}_j), [\mathsf{l}_{ij}, \mathsf{u}_{ij}]) \in E_R$,

1. $\mathsf{V} := \mathsf{V} \cup \left\{\mathsf{v}_{ij}^1, \mathsf{v}_{ij}^2, \ldots, \mathsf{v}_{ij}^{\mathsf{u}_{ij}-1}\right\}$ and $\omega(\mathsf{v}_{ij}^k) := c_{\mathsf{v}_i}$ for all $k = 0..\mathsf{u}_{ij} - 1$ (where $\mathsf{v}_{ij}^0 = \mathsf{v}_i$ and $\mathsf{v}_{ij}^{\mathsf{u}_{ij}} = \mathsf{v}_j$),
2. $\mathsf{E} := \mathsf{E} \setminus \{\mathsf{e}\}$,
3. $\mathsf{E} := \mathsf{E} \cup \left\{(\mathsf{v}_{ij}^k, \mathsf{v}_{ij}^{k+1}) \mid k = 0..\mathsf{u}_{ij} - 1\right\}$, and $\omega(\mathsf{v}_{ij}^k, \mathsf{v}_{ij}^{k+1}) := 1$ for all $k = 0..\mathsf{u}_{ij} - 1$,
4. $\mathsf{E} := \mathsf{E} \cup \left\{(\mathsf{v}_{ij}^k, \mathsf{v}_j) \mid k = \mathsf{l}_{ij}..\mathsf{u}_{ij} - 1\right\}$, and $\omega(\mathsf{v}_{ij}^k, \mathsf{v}_j) := 0$ for all $k = \mathsf{l}_{ij}..\mathsf{u}_{ij} - 1$.

Roughly speaking, G is built by "splitting" each edge $\mathsf{e} = (\mathsf{v}, \mathsf{v}', [\mathsf{l}, \mathsf{u}])$ of $\mathcal{RG}$ into u small edges with the length (weight) 1 by adding $\mathsf{u} - 1$ sub-vertices. All of these sub-vertices and v are assigned a weight as the coefficient c_s in LDI, where s is location of vertex v ($s \in \mathsf{v}$). On the other hand, from sub-vertices v^{l} to $\mathsf{v}^{\mathsf{u}-1}$ there are edges joining these sub-vertices to v' of the edge e with length 0. Hence, from v we can reach v' of the edge e through a path passing through only sub-vertices in G with the integer lengths between l and u. For the simplicity of presentation we call vertices v and v' of edge e mother vertices and call the sub-vertices in e child vertices. Besides, all the paths joining v and v' that go through only child vertices of e (in $\mathcal{RG}$) are also called paths belongs to e.

Figure 2 gives an example how to build graph G' from simple graph G with 2 edges.

In order to make use of G, we have to show that G is compatible to $\mathcal{RG}$ w.r.t checking LDI. First, we define length, cost and satisfaction of a path p in G w.r.t LDI $\mathcal{D}$.

Definition 9. *Let* $\mathsf{p} = \mathsf{v}_1\mathsf{v}_2 \ldots \mathsf{v}_m$ *be a path in* G. *The length* $l(\mathsf{p})$ *and the cost* $\theta(\mathsf{p})$ *of* p *are defined as*

$$l(\mathsf{p}) \mathrel{\widehat{=}} \sum_{i=1}^{m-1} \omega(\mathsf{v}_i, \mathsf{v}_{i+1}), \quad \theta(\mathsf{p}) \mathrel{\widehat{=}} \sum_{i=1}^{m-1} \omega(\mathsf{v}_i)\omega(\mathsf{v}_i, \mathsf{v}_{i+1}).$$

A path p *satisfies* $\mathcal{D}$ *iff* $A \le l(\mathsf{p}) \le B \Rightarrow \theta(\mathsf{p}) \le M$.

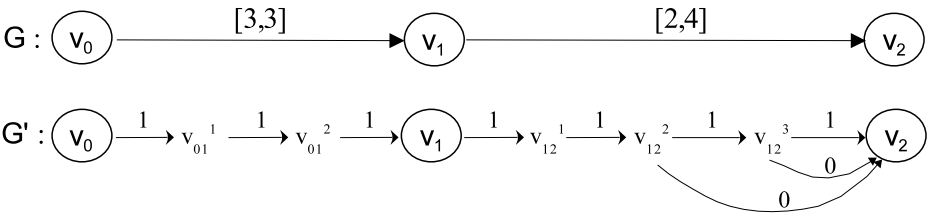


Fig. 2. "Discretising" graph of region graph

Lemma 8. *Each integral weighted interpretation* $\wp = (\mathsf{p}, \overline{d})$ *of* $\mathcal{RG}$ *corresponds to a path* p' *in* G *such that* $l(\wp) = l(\mathsf{p}')$, $\theta(\wp) = \theta(\mathsf{p}')$, *and reversely, each path* p' *in* G *corresponds to an integral weighted interpretation* $\wp = (\mathsf{p}, \overline{d})$ *of* $\mathcal{RG}$ *such that* $l(\wp) = l(\mathsf{p}')$, $\theta(\wp) = \theta(\mathsf{p}')$.

Proof. It is obvious from the definition of G that for each edge $(\mathsf{v}_i, \mathsf{v}_j, [\mathsf{l}, \mathsf{u}])$ of $\mathcal{RG}$ and an integer $d \in [\mathsf{l}, \mathsf{u}]$ there exists a path $\mathsf{p} = \mathsf{v}_i \mathsf{v}_{ij}^1 \ldots \mathsf{v}_{ij}^d \mathsf{v}_j$ of G such that $l(\mathsf{p}) = d$, $\theta(\mathsf{p}) = c_{\mathsf{v}_i} d$ and vice-versa. Hence, the lemma is correct.

From Lemma 8 we can conclude that if there exists an integral model $\sigma \in \mathcal{M}_{uv}(\mathcal{A})$ not satisfying LDI $\mathcal{D}$ then there exists a path that joins mother vertices of G and does not satisfy LDI and reversely. A similar result for any integral model of $\mathcal{A}$ and any path (joining two child vertices) of G is formulated by following lemma.

Lemma 9. *Given a timed automaton* $\mathcal{A}$, *a LDI* $\mathcal{D}$ *and weighted graph* G *as above. Then if there exists a path* $\mathsf{p} \in \mathcal{P}(\mathsf{G})$ *such that* $\mathsf{p} \not\models$ *LDI then there exists an integral model* $\sigma \in \mathcal{M}_I(\mathcal{A})$ *such that* $\sigma \not\models$ *LDI and vice-versa.*

Proof. See [10].

Theorem 2. *Checking the satisfaction of LDI* $\mathcal{D}$ *by timed automaton* $\mathcal{A}$ *is equivalent to checking the satisfaction of LDI* $\mathcal{D}$ *by the set of paths* $\mathcal{P}(\mathsf{G})$. *That is,* $\mathcal{A} \models \mathcal{D}$ *if and only if* $\mathcal{P}(\mathsf{G}) \models \mathcal{D}$.

This theorem follows immediately from Lemma 9 and the discretisability of LDI w.r.t timed automata $\mathcal{A}$.

4.4 Algorithm for Checking LDI

In this section we present the idea an algorithm for checking $\mathcal{A} \models \mathcal{D}$ based on traversing the weighted graph G. The algorithm uses procedure **Traverse**(vstart) and procedure **Checking-LDI**. The procedure **Traverse**(vstart) explores every path starting from fixed vertex vstart to see if it satisfies $\mathcal{D}$, and the procedure **Checking-LDI** calls procedure **Traverse**(vstart) for all vertices $\mathsf{vstart} \in \mathsf{V}$ for deciding satisfaction of $\mathcal{D}$ by $\mathcal{A}$.

The procedure **Traverse(**vstart**)** uses the backtracking technique to explore the graph. Starting from vertex vstart of G, the procedure constructs the current path p ($l(\mathsf{p})$ and $\theta(\mathsf{p})$ initialised to 0) while going along out-going edges to their destination vertices at which $l(\mathsf{p})$ and $\theta(\mathsf{p})$ are re-calculated and $A \leq l(\mathsf{p}) \leq B \Rightarrow \theta(\mathsf{p}) < M$ is verified. The procedure goes back when the current path p cannot be expanded (see the next paragraph how a path can be expanded) or the length of current path exceeds B, and terminates when either $A \leq l(\mathsf{p}) \leq B \Rightarrow \theta(\mathsf{p}) < M$ is violated or it goes back to the starting vertex vstart with no more out-going edge to go.

The current path p cannot be expanded when there is no new out-going edge to go when the number of repetitions of cycles has reached the limit. This applies only when $B = \infty$. When a positive cycle is discovered, i.e. a cycle p' (which is a sub-path of p) with $\theta(p') > 0$, the procedure returns $\mathcal{A} \not\models \mathcal{D}$. When there is no positive cycle in p, p cannot be expanded when the number of cycle repetitions has reached $k = \lceil \frac{A-c}{\triangle c} \rceil$, where c is the length of the shortest cycles in p. Any more repetition of a cycle will make $\theta(\mathsf{p})$ smaller. So, there is no need to check with expansion of p by more repetitions of cycles.

So, either there is a positive cycle in G, or eventually, either p will become not expandable, or $l(\mathsf{p}) > B$ will be reached for a path p starting from vstart. So, the procedure **Traverse(**vstart**)** will terminate eventually.

The detailed technical construction of the algorithm can be worked out easily, and is omitted here.

5 Conclusion

Exploring reachability graphs is one of popular methods for checking reachability property and some properties concerning time instants of real time systems. However, paths in the reachability graph do not preserve time durations of system locations, and hence, cannot be used for checking duration properties. By equipping edges of integral reachability graphs with the minimal and maximal bounds of state transitions, we are able to use this technique for checking duration properties. We have proposed an algorithm for checking LDI of timed automata using this technique. In this paper we have proved the discretisability of LDI, and proposed an algorithm based on this result to check if a timed automaton satisfies a LDI in the general semantics. Although the complexity of this algorithm is high, it can serve, at least, for showing that checking LDI of closed time automata is decidable. We do believe that checking is feasible for some specific LDI and abstract timed automata.

References

1. R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, pp. 183–235, 1994.
2. R. Alur. Timed Automata. Proceedings of 11^{th} *International Conference on Computer-Aided Verification,* LNCS 1633, pp. 8–22, Springer-Verlag, 1999.

3. Victor A. Braberman and Dang Van Hung. On Checking Timed Automata for Linear Duration Invariants. Technical Report 135, UNU/IIST, P.O.Box 3058, Macau, February 1998. Proceedings of *the 19th Real-Time Systems Symposium RTSS'98*, December 2–4, 1998, Madrid, Spain, IEEE Computer Society Press 1998, pp. 264–273.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
5. Y. Kesten, A. Pnueli, J Sifakis, and S. Yovine. Integration Graphs: A Class of Decidable Hybrid Systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 179–208. Springer Verlag, 1994.
6. Thomas A. Henzinger, Zohar Manna and Amir Pneuli. Towards Refining Temporal Specifications into Hybrid Systems, Hybrid Systems I, LNCS 736,Springer-Verlag, 1993.
7. Li Xuan Dong and Dang Van Hung. Checking Linear Duration Invariants by Linear Programming. Research Report 70, UNU/IIST, P.O.Box 3058, Macau, May 1996. Published in Joxan Jaffar and Roland H. C. Yap (Eds.), *Concurrency and Parallelism, Programming, Networking, and Security* LNCS 1179, Springer-Verlag, Dec 1996, pp. 321–332.
8. Li Yong and Dang Van Hung. Checking Temporal Duration Properties of Timed Automata. Technical Report 214, UNU/IIST, P.O.Box 3058, Macau, October 2001. Published in *Journal of Computer Science and Technology*, Vol. 17, No. 6, Nov. 2002. pp. 689 – 698.
9. Pham Hong Thai and Dang Van Hung. Checking a Regular Class of Duration Calculus models for Linear Duration Invariants. Technical Report 118, UNU/IIST, P.O.Box 3058, Macau, July 1997. Proceedings of the *International Symposium on Software Engineering for Parallel and Distributed Systems* (PDSE'98), Kyoto, Japan, 20 – 21, April 1998. Bernd Kramer, Naoshi Uchihira, Peter Croll and Stefano Russo (Eds). IEEE Press 1998, pp. 61 – 71.
10. Pham Hong Thai and Dang Van Hung. Verifying Linear Duration Constraints of Timed Automata. Technical Report 306, UNU/IIST, P.O.Box 3058, Macau, June 2004.
11. S. Tripakis, S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. *Formal Methods in System Design*, 18, 25–68, 2001. Kluwer Academic Publishers, Boston.
12. Zhao Jianhua and Dang Van Hung. Checking Timed Automata for Some Discretisable Duration Properties. Technical Report 145, UNU/IIST, P.O.Box 3058, Macau, August 1998. Published in *Journal of Computer Science and Technology*, Volume 15, Number 5, September 2000, pp. 423–429.
13. Zhou Chaochen , C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
14. Zhou Chaochen, Zhang Jingzhong, Yang Lu, and Li Xiaoshan. Linear Duration Invariants. Research Report 11, UNU/IIST, P.O.Box 3058, Macau, July 1993. Published in: *Formal Techniques in Real-Time and Fault-Tolerant systems*, LNCS 863, 1994.
15. Zhou Chaochen and Hansen M. R., *Duration Calculus. A Formal Approach to Real-Time Systems*, Springer, 2004.

Idempotent Relations in Isabelle/HOL

Florian Kammüller and J.W. Sanders

Technische Universität Berlin,
Institute for Software Engineering and Theoretical Computer Science
and University of Oxford, Computing Laboratory

Abstract. A characterization of idempotent relations is presented first as a paper-style proof, then by its formalization in Isabelle/HOL. The novel characterization gives rise to the construction of idempotent relations by an abstract algorithm. This algorithm is rigorously developed inside Isabelle/HOL using primitive recursive function definitions. Whilst the characterisation and algorithm appear to be new, we regard this as an interesting demonstration of the interplay between mathematical reasoning and program development, in particular using Isabelle/HOL.

1 Introduction

Idempotents are important in various areas of Mathematics. An abstract structure is frequently able to be represented in a particular concrete form in which each abstract element is represented as a transformation on some underlying set and the abstract operation is represented as sequential (or functional) composition of transformations. In some cases the idempotents (i.e. the elements e satisfying $ee = e$, where we follow convention and write the algebraic operation as juxtaposition) form the basis for that representation.

Here are two examples. In Analysis, a linear function on a vector space is idempotent under sequential composition only if it is a projection onto a subspace of the vector space. That is the foundation of spectral resolution, in which a (say normal) linear operator is represented in terms of projections [DS58]. In Algebra, much of the structure theory of (abstract) semigroups [CP61] and some of the theory of ideals in ring theory [CP61] depends on identifying the idempotents.

In Computer Science, a transaction consists of a sequence $t = t_1, \ldots, t_m$ of actions. Given a second transaction $u = u_1, \ldots, u_n$ the idea is to perform both transactions efficiently, but as if each were executed atomically. Efficiency is achieved by interleaving the actions of t with those of u; but correctness is preserved only if the desired interleaving is obtained from the sequential composition $t; u$ by interchanging those actions that commute: $t_i; u_j \sqsubseteq u_j; t_i$. If, in doing so, two identical actions become adjacent then one of them can be deleted if it is idempotent: $t_i; t_i = t_i$. Though not common, it is important to take advantage of such simplification whenever possible.

Linear operators, functions and actions are all special cases of relations. Determining whether or not a given relation is idempotent is routine: it has the

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 310–324, 2005.
© Springer-Verlag Berlin Heidelberg 2005

same computational complexity as that of Boolean-matrix squaring (i.e. multiplication). However the generation of all idempotent relations on a finite set appears to be more difficult. It is not even clear, for example, how many idempotent relations there are. One of the motivations for finding an algorithm to generate them has thus been to use it as a basis for counting them. The naive approach of generate-and-test has the following complexity. (a) The complexity of squaring a Boolean matrix is at worst $O(n^{lg7})$ by Strassen's algorithm. (b) On a set of size n there are 2^{n^2} relations and so the generate-and-test algorithm takes the product of that and the size in (a). An abstract algorithm that does better is the topic of this paper.

This paper first introduces a theorem and a proof characterizing idempotent relations which we believe to be original. Next, this theorem is formalized and proved in the interactive theorem prover Isabelle [Pau94] using its instantiation to Higher Order Logic (HOL). Besides confirming that the paper proof is correct, the formalization represents an interesting application of Isabelle/HOL, particularly as the mechanical proof involves some reasoning about finite sets, which is usually quite tricky. Finally, we devise an algorithm that computes idempotent relations. This algorithm is defined inside Isabelle/HOL using primitive recursive functions. This final aspect is, from an engineering point of view, probably the most interesting aspect of this work: starting from a theoretical characterization an effective procedure is derived. Thereby, we show that Isabelle/HOL, can be used as a program development framework in which correct programs can be derived — and even be tested on the fly, since the primitive recursive functions can be translated one to one into ML.

The paper is organized as follows: the next section introduces the theoretical characterization (with proof). After that, Section 3 presents the mechanization of the theorem in Isabelle. Section 4 then starts explaining the actual content of the theoretical characterization by introducing some representative examples, leading on to an informal description of a construction. Section 4.2 is used for the presentation of the algorithm based on primitive recursive functions. Section 4.3 then indicates the properties that have to be shown to prove the correctness of the algorithm. Finally, in Section 5 we draw some conclusions.

2 A Characterization of Finite Idempotent Relations

We characterize idempotent relations under the assumption of finiteness, and show by example that we cannot do better.

A relation r on a given set is idempotent iff $r \circ r = r$ where $\circ$ is relational composition, i.e.

$$r \circ s = \{(x, y).\exists z.\ (x, z) \in s \wedge (z, y) \in r\}\,.$$

Where convenient (particular in pictures) we use the notation $r; r$ for relational composition. For simplicity, let $r(x)$ stand for the relational image $r.(\!|\{x\}|\!)$ (where $r.(\!|A|\!) = \{y.\exists x \in A.(x, y) \in r\}$) and r^2 stand for $r \circ r$. The characterization is

based on fixpoints of the relation, i.e. elements x of the domain with $x \in r(x)$, that is $(x, x) \in r$.

The characterization is given by the following theorem.

Theorem 1. *Let r be a finite relation. Then*

$$\text{idempotent } r \quad \equiv \quad \forall x \in \mathbf{dom}\ r.\ \begin{pmatrix} r(x) = \bigcup y \in r(y) \cap r(x).\ r(y) \\ \forall y_a \in r(x).\ r(y_a) \subseteq r(x) \end{pmatrix}.$$

Clearly, idempotence implies transitivity $r \circ r \subseteq r$. The second conjunct is equivalent to r being transitive, but we prefer to write it this way to emphasize the relationship between the ranges of single elements. The first conjunct describes $r \subseteq r \circ r$ and is the major clue to the construction to follow.

We prove the theorem using the following lemmata.

Lemma 1. *Let r be idempotent. Then*

$$x \in r(x) \Rightarrow r(x) = \bigcup y \in r(y) \cap r(x).\ r(y).$$

Proof: Transitivity gives us $\forall y \in r(x).\ r(y) \subseteq r(x)$. Clearly,

$$\bigcup y \in r(y) \cap r(x).\ r(y) \subseteq \bigcup y \in r(x).\ r(y)$$

and by transitivity of $\subseteq$ the left-hand side is a subset of $r(x)$. Since $x \in r(x)$, $r(x) \subseteq \bigcup y \in r(y) \cap r(x).\ r(y)$, whereby we have equality. □

Lemma 2. *Let r be finite and idempotent. Then*

$$x \in \mathbf{dom}\ r, x \notin r(x) \Rightarrow \forall z \in r(x).\ \exists\ y \in r(y) \cap r(x).\ z \in r(y).$$

Proof: We prove that if the assumption and the negation of the conclusion hold then r is not finite. Assume for contradiction

$$\exists\ z \in r(x).\ \neg \exists\ y \in r(y) \cap r(x).\ z \in r(y)$$

Since $(x, x) \notin r$, we need $y_0 \neq x$ for (x, z) to be in r^2 in order to have $(x, y_0), (y_0, z) \in r$ and thereby $(x, z) \in r$. Now, $y_0 \neq z$ otherwise we had a $y = z$ with $z \in r(z)$ contradicting the assumption. Summarizing, $y_0 \neq x$ and $y_0 \neq z$. However, now $y_0 \in r(x)$ and $y_0 \notin r(y_0)$. By repetition of the argument, we need a y_1 with $(x, y_1), (y_1, y_0) \in r$ with $y_1 \notin \{x, z, y_0\}$, and so forth — ultimately leading to an infinite sequence of $y_i \in r(x)$, establishing that r is not finite. □

Proof of Theorem 1. Now, we are prepared for the proof of the theorem.
Correctness (⇒): Let r be idempotent and $x \in \mathbf{dom}\ r$ be arbitrary. If $x \in r(x)$, just apply Lemma 1. If $x \notin r(x)$, Lemma 2 gives

$$\bigcup z \in r(x) \subseteq \bigcup y \in r(y) \cap r(x).\ r(y)\,.$$

Since r is idempotent, it is also transitive. Hence, the right-hand side $\subseteq r(x)$. Since the left-hand side is equal to $r(x)$ we have also for $x \notin r(x)$ that

$$r(x) = \bigcup y \in r(y) \cap r(x).\ r(y)\,.$$

Completeness ($\Leftarrow$): Let $r(x) = \bigcup y \in r(y) \cap r(x).\ r(y)$. For any $(x, y) \in r$, $y \in r(x)$. By assumption there is y' with $y \in r(y')$ for some $y' \in r(y') \cap r(x)$, i.e. $(x, y') \in r$ and $(y', y') \in r$. Since $y \in r(y')$, also $(y', y) \in r$, hence $(x, y) \in r^2$.

As the second conjunct of the characterization corresponds to transitivity, we have on the other hand that if $(x, y) \in r^2$ then $(x, y) \in r$. □

From the theorem it follows immediately that if a finite idempotent relation is nonempty then it has a fixpoint.

Corollary 1. *If $r \neq \varnothing$ is finite and idempotent then $\exists\ x.\ x \in r(x)$.*

By contraposition this implies that if there is no fixpoint, the relation must be infinite.

Corollary 2. *If r is idempotent, $r \neq \varnothing$ and $\neg\exists\ x.\ x \in r(x)$ then r is infinite.*

An illustrative example is the relation $<$ on rational numbers.

Example 1. The relation $<: \mathbb{Q} \times \mathbb{Q}$ is idempotent and infinite.

The relation $<$ is obviously transitive, and for any x and y with $x < y$ there is an element between x and y.

3 Mechanical Proof

In this section and the following we introduce some Isabelle/HOL formalizations. Since this tool supports mathematical syntax, the only peculiarities to mention from the start are: `[| P; R |] ⟹ S` is a meta-level implication and can be read as $(P \wedge Q) \Rightarrow S$. In contrast `P ⟶ Q` is the implication of the object logic HOL. Other operators will be explained when necessary. Formalizations in Isabelle start by defining a theory that contains types, constant declarations, and definitions. The theory for idempotent relations contains just one definition for idempotence.

```
idempotent :: (α × α) set => bool
"idempotent r == (r ∘ r = r)"
```

Theorem 1 is then proved in the scope of that theory. The representation of the theorem in Isabelle is almost like the paper-style theorem. However, we have had to resolve the self-reference in the binder of the union as otherwise the binding would not have worked. The relational image of a singleton set is denoted `r"{x}`.

```
finite r ⟹ idempotent r =
  ∀ x ∈ Domain r. r"{x} = ⋃ y: {z. z ∈ r"{z} ∩ r"{x}}. r"{y} ∧
                          ∀ ya ∈ r"{x}. r"{ya} ⊆ r"{x}
```

3.1 Proof of Lemma 1

The proof of Lemma 1 is very simple in Isabelle. Using a lemma that infers transitivity from idempotence it is just one application of the elimination rule for transitivity. The rest is done automatically using the tactic `auto`.

```
[| idempotent r; x: r"{x} |] ⟹
          r"{x} = ⋃ y ∈ {z. z ∈ r"{z} ∩ r"{x}}. r"{y}
```

3.2 Proof of Lemma 2

This part of the proof of Theorem 1 is the difficult bit. What is done on paper rather casually and informally by sketching a repetitive process in which yet another element y_i is needed and then concluding that the set $r(x)$ cannot be finite, is harder on the logical level. The repetitive process is first proved as a lemma. Applying this lemma in an induction the existence of an infinite sequence is proved. Some further theorems that generalize from the existence of this particular sequence then provide the possibility to infer infinity from there. These theorems can then be chained together to construct the contradiction to the assumed finiteness.

Core Lemma. The core lemma describes that under the assumptions of Lemma 2 it is possible to infer a new element `y` that is in relation `r` to all others so far, but is not equal to any of the former ones.

```
[| idempotent r; x ∈ Domain r; x ∉ r"{x}; z ∈ r"{x};
  ¬ (∃ y. y ∈ r"{y} ∧ y ∈ r"{x} ∧ z ∈ r"{y}); z = s 0;
  ∀ j. j ≤ n ⟶ s j ∈ r"{x} ∧ ∀ i. i < j ⟶ (s j,s i)∈ r ∧ s j ≠ s i
|] ⟹ ∃ y. y ∈ r"{x} ∧ ∀ j. j ≤ n ⟶ (y, s j)∈ r ∧ y ≠ s j
```

Similar to the paper style proof it uses the properties of idempotence to infer that new "middle" element and furthermore transitivity to establish the invariant that it is related to all previous ones. We use here a variable `s` that formalizes a sequence over natural numbers used in the following to produce the infinite sequence.

A Chain of Lemmata. The first step of the proof leading to the conclusion that there is an infinite sequence, is an induction that shows that under the given assumptions of Lemma 2, there is such a sequence `s`.

```
[| idempotent r; x ∈ Domain r; x ∉ r"{x}; z ∈ r"{x};
   ¬ (∃ y. y ∈ r"{y} ∧ y ∈ r"{x} ∧ z ∈ r"{y})
|] ⟹ ∀ n. ∃ s:: nat => α . z = s 0 ∧
          (∀ j. j ≤ n ⟶ (s j) ∈ r"{x} ∧
          (∀ i. i < j ⟶ (s j, s i) ∈ r ∧ (s j) ≠ (s i)))
```

This proof is an induction over natural numbers. In the induction step the core lemma is applied to produce the new element of the sequence `s` having the appropriate properties.

The conclusion of the previous step can be weakened.

```
    ∀ n. ∃ s:: nat => α . z = s 0 ∧ (∀ j. j ≤ n ⟶ (s j) ∈ r"{x} ∧
      (∀ i. i < j ⟶ (s j, s i) ∈ r ∧ s j ≠ s i))
⟹ ∀ n. ∃ s. ∀ j. j ≤ n ⟶ s j ∈ r"{x} ∧ (∀ i. i < j ⟶ s j ≠ s i)
```

The weaker set of properties of the sequence is sufficient to infer the existence of a set whose cardinality is always growing and whose elements are all subsets of a set `p` — which will ultimately be `r"{x}` in our case.

```
[| ∀ n. ∃ s:: nat => α .
 (∀ j. j ≤ n ⟶ (s j) ∈ p ∧ (∀ i. i < j ⟶ (s j) ≠ (s i))) |]
⟹ ∀ n. ∃ S. card S = Suc n ∧ S ⊆ p
```

Finally, the set derived in the previous step can be used to infer that the set `p` is infinite.

```
∀ n. ∃ S. card S = Suc n ∧ S ⊆ p ⟹  ¬ finite p
```

The variable `p` of type set can be instantiated to `r"{x}`. Thereby, chaining up all these lemmata, we can put together the proof of Lemma 2 by producing a contradiction with the assumed finiteness of the relation.

```
[| finite r; idempotent r |] ⟹
  ∀ x ∈ Domain r. x ∉ r"{x} ⟶
    (∀ z ∈ r"{x}. ∃ y. y ∈ r"{y} ∧ y ∈ r"{x} ∧ z ∈ r"{y})
```

It may seem a bit odd that we have to derive first that the sets `S` we are constructing for contradiction have a cardinality. However, as infinity is just the negation of finiteness, the only way to construct a contradiction is to arrive at a property that a finite set has, i.e. a finite cardinality, and that clearly cannot be assumed for the sequence.

The proof of Lemma 2 is, like some proofs in lattice theory [BKS01, DP02], rather intricate. It would be much easier if a sequence could be constructed on the outside of the universal quantification over n, i.e. $\exists s.\forall n \ldots$. However, this is not possible in our case. We have to show that such a sequence exists for each n. Fortunately, as the core lemma can be identified and applied inside the induction this sequence can be prolonged in each step and by identifying the commonality of the sequences — that they are all in some set `p` — we can construct the sequence of sets represented by the existentially quantified `S`.

Proof of the Theorem. The proof of the correctness, i.e.

```
[| finite r; idempotent r |] ⟹
  (∀ x ∈ Domain r. r"{x} = (⋃ y: {z. z: r"{z} ∩ r"{x}}. r"{y}) ∧
                           (∀ ya ∈ r"{x}. r"{ya} ⊆ r"{x}))
```

simply puts together Lemma 1 and Lemma 2 with transitivity.

For completeness note that we can infer the property $r \subseteq r \circ r$ from the first conjunct of the characterization alone.

```
(∀ x ∈ Domain r. r"{x} = (⋃ y: {z. z ∈ r"{z} ∩ r"{x}}. r"{y}))
⟹ r ⊆ r ∘ r
```

The other conjunct is equivalent to transitivity so we can prove the other inclusion almost automatically. Finally, we put the two parts together to finish the proof.

4 Algorithm

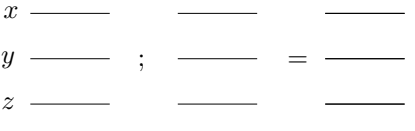
In this section we will develop an abstract algorithm that builds all idempotent relations over a given domain. Before introducing the construction algorithm, we illustrate how the characterization may be seen as a recipe for constructing idempotent relations. After that, in Section 4.2 we introduce the primitive recursive definitions of the algorithm. Finally in Section 4.3 we outline the major steps in proving the correctness and discuss the solution in Section 4.4.

4.1 Constructing Idempotent Relations

The main idea of the construction is to start from relations that are constituted by *fixpoints*, i.e. pairs of the form (x, x) (see Section 2), and so are trivially idempotent. The next step adds more relations based on the latter by extending the ranges of the single points according to Theorem 1. Finally, from those — again idempotent relations — we get all idempotents by considering admissible domain extensions.

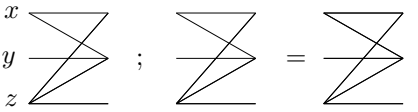
To understand the characteristics of idempotent relations we consider some representative examples to illustrate the meaning of the characterization in Theorem 1. We consider small examples depicting the relations graphically as lines connecting points picking out typical cases that illustrate the scope of idempotence and prepare the ground to find an effective procedure to construct them.

Fixpoints. The first and most simple example is that of a relation that consists of only fixpoints. For the three elements x, y, z the relation $r = \{(x, x), (y, y), (z, z)\}$ is depicted together with a graphical illustration of its idempotence:



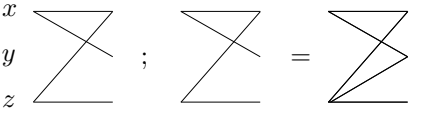
The points that are connected by the relation are all fixpoints. Where in extension it is hard to decide whether a relation is idempotent, graphically it reduces to following up all paths from the left of the left graph to the right of the right graph.

Range Extensions. Starting from a pure fixpoint relation, a relation that is constructed by extending the ranges of the fixpoints is in most cases also idempotent. Consider the following example, where starting from the same fixpoints the ranges of x and z are extended by y and $\{x, y\}$ respectively.

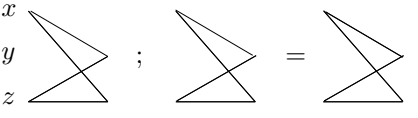


The composition gives the same element, hence this relation is idempotent. This extension conforms to the first conjunct of Theorem 1 for the case of fixpoints, i.e. Lemma 1: the ranges of x and z are defined by just their own ranges.

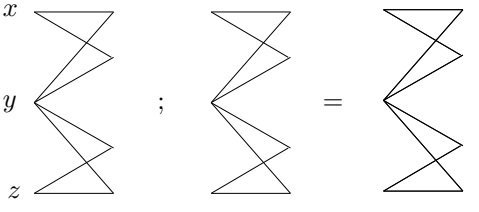
However, even though in most cases the range extension of fixpoints is arbitrary, one has to respect the second conjunct of Theorem 1, transitivity. In the following example we have just the fixpoints x and z, but the range of z contains y, while $r(y) \not\subseteq r(z)$! Hence, the relation on the left below is **not** idempotent (whilst the result on the right is).



Hangers-on. As a final step relations that are constructed from fixpoints and admissible range extensions (such extensions that respect transitivity) can be further extended by adding non-fixpoint elements on the domain side. These additional domain extensions of the relations can be considered *hangers-on*: they hang on to the fixpoint elements, thereby using their "idempotence". This is an instance of the first conjunct of Theorem 1 (more precisely Lemma 2): for elements of the domain of the relations with $x \notin r(x)$, i.e. non-fixpoints, there must exist fixpoints whose ranges constitute the range of these non-fixpoints. An example is given by the following relation where the non fixpoint x hangs on to the fixpoint z having exactly the same range as z.

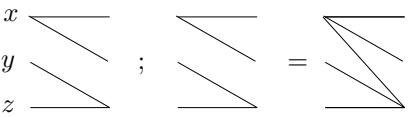


Hangers-on need to copy the entire range of the fixpoints they hang-on to. However, they can have more than one fixpoint they hang on to. The following example with five points, shows how the hanger-on y can actually accumulate the ranges of both fixpoints x and z.



The previous example illustrates why in the characterization of Theorem 1 we need the union over all fixpoints for the case of hangers-on.

For domain extensions we also have to respect transitivity, as illustrated by the following example.



Although each of the components of the graphs in this relation respects the rules found so far, the combination is not idempotent. The way to avoid this when building domain extensions is to consider only such sets of hangers-on that are not contained in any range of a fixpoint (here hanger-on y is contained in the range of fixpoint x).

Informal Algorithm. To use the intuition given above to develop an algorithm, we proceed informally as follows:

- For a given list of fixpoints l build range extensions for each of the fixpoints.
  - The fixpoint is always contained in the range extension. The range extensions are initially all possible subsets of the intended range R of the relation resulting in a list of relations that are lists of pairs (l_i, s_i) of fixpoint and range extension, where $l_i \in s_i$.
  - Check the resulting lists of pairs (l_i, s_i) of fixpoint and range extension for transitivity and delete the nontransitive relations.
- Build from the list of range extended relations all combinations d_i of possible domain extensions. A domain extension describes which elements hang-on to a fixpoint, i.e. share its range.
  - A domain extension is any subset of the domain of the prospective relation that does not contain fixpoints from l and that is not in the range s_i of any fixpoint.
  - The elements in the domain extension all have the same range as the fixpoints they are hanging-on to. So it suffices to record the element together with the fixpoints.
- Finally, the derived list of elements of the form

$$[(l_1, s_1, d_1), \ldots (l_n, s_n, d_n)]$$

  represents a relation that can be recovered from this list representation as

$$r = \{(x, y).\exists\ i.\ (x = l_i \vee x \in d_i) \wedge y \in s_i\}.$$

4.2 Primitive Recursive Construction Functions

Instead of defining a nested loop that builds the entire set of all idempotent relations for given domain, range, and fixpoints, we implement the algorithm in a functional manner using primitive recursive functions directly as they are provided in Isabelle/HOL.

Initially, we need at various points in the algorithm a function that produces all possible subsets of a set. As we decided to implement the algorithm using lists, the function `sublists` builds sublists of a list rather than sets.

```
sublists :: "α list => α list list"
```

Assuming that the recursion returns all possible sublists of a list `l` the step of the function builds the possible sublists of list `c # l` by concatenating all sublists of `l` with all sublists of `l` where the element `c` has been put in first. The recursion finishes by returning the list with the empty sublist.

```
primrec
  sublists_empty: "sublists_fn [] = [[]]"
  sublists_step:  "sublists (c # l) =
                     (let ll = sublists l in ll @ (map(λ x. c # x) ll))"
```

This is an example of a primitive recursive definition as indicated by the keyword `primrec`. We omit the keyword in the following when it is clear from context.

Next we consider the function that given the prospective range of the relation and a list of fixpoints builds the range extensions `range_exs R l`.

```
range_exs :: "[α list, α list] => (α × α list)list list"
```

This function is rather complex. For each fixpoint `li` it first builds all sublists of the range `R` omitting `li` to avoid repetitions as `li` has to be in the range extension — and is inserted into each range extension afterwards.

```
range_exs_empty: "range_exs R [] = []"
range_exs_step: "range_exs R (li # ll)  =
      (let sl = (sublists [x:R. x ≠ li])
       in (if (ll = []) then  map (λ x. [(li,li # x)]) sl
           else combine_re (map (λ x. (li, li#x)) sl) (range_exs R ll)))"
```

The expression `[x:R. x ≠ li]` is a filter denoting the list of elements in list `R` that are `≠ li`. In the function body a function `combine_re` is used that will be explained next.

```
combine_re :: "[α list, α list list] => α list list"
```

In the body of `range_exs_step` the algorithm builds all combinations of range extensions by working through a given list of fixpoints `l` from back to front. Given that the recursion `range_exs R ll` returns all possible combinations of range extensions for the postfix `ll` of the fixpoints `l`, all possible range extensions for the postfix `li # ll` are built by combining each possible range extension for `li`, say `(li,si)` with each of the lists in `range_exs R ll` by adding `(li,si)` as first element. The function `combine_re` now performs exactly the necessary combination. It is similar to combine (see below) but the insertion depends on the check `fp_check`.

```
combine_re_empty: "combine_re [] l = []"
combine_re_step: "combine_re (a # l1) l =
         (map (λ x. if (fp_check a x) then (a # x) else []) l)
         @ (combine_re l1 l)"
```

The function `fp_check` that is used in `combine_re` checks whether the range extension that is created by adding the actual pair to an already range extended postfix, introduces violations of transitivity in which case `combine_re` deletes this element from the combinations.

```
fp_check:: "[(α × α list),(α × α list)list] => bool"
```

This check concerns the property of transitivity, formulated in Theorem 1 as:

$$\forall x \in \mathbf{dom}\ r.\ \forall y_a \in r(x).\ r(y_a) \subseteq r(x)$$

If an element y_a is also in the range of another x than y_a's range has to be contained in the range of x. For fixpoints, which we are considering just now, this property can be slightly simplified to considering such fixpoints that are themselves in the domain of other fixpoints: in the case that a fixpoint l_i is in the range of a fixpoint l_j, the range of l_j has to be a subset of the range of l_i. In the construction of the range extension this criteria applies two ways: the newly added fixpoint l_i could be in the range of an "old" l_j or vice versa. If both are contained in the ranges of each other, clearly their ranges have to be the same[1]. The constructor `set` transforms a list into a set.

```
fp_check_empty: "fp_check a [] = True"
fp_check_step: "fp_check (li,si) ((lj,sj) # lr) =
        (if (li mem sj) then (if (lj mem si) then (set si) = (set sj)
                                    else (set si ⊆ set sj))
         else (if (lj mem si) then (set sj ⊆ set si)
                     else fp_check a lr))"
```

It is safe to cancel such elements in the construction of the range extensions that do not pass the function `fp_check`, because the algorithm goes through all possible combinations. Hence, just a bit further down the line the current range extension `(li,si)` is going to be combined with a slight variation that fits.

The next step in the algorithm is to construct the domain extensions. Here, the procedure is structurally very similar to the range extension. That is, we again use a combination function to build all possible combinations of domain extensions this time using a simple version of the function `combine_re` called simply `combine`.

```
combine_empty: "combine [] l = []"
combine_step: "combine (a # l1) l = (map (Cons a) l) @ (combine l1 l)"
```

Now, a domain extension of a fixpoint on a given domain for the relation and list of fixpoints can be applied to a list of range extensions, that is to a list of elements of type $\alpha \times \alpha$`list`. It returns a list of lists of triples, that are the range extentions extended by an additional list as third element, representing the elements of the domain that share the range of the fixpoints.

```
domain_exs :: "[α list, α list, (α × α list)list]
                      => (α × α list × α list)list list"
```

[1] To make the definition more readable, we use pattern matching of pairs in the argument position of the following primitive recursive definition, but this is actually not supported in Isabelle/HOL.

For a list `D` representing the relations domain and a list `l` of fixpoints, a domain extension of a fixpoint `li` is a subset of the domain that has neither elements of `l` nor elements in the range extensions `si` of any `li`. This constraint is realized by using again the filter construct in `[x:D.(¬(x mem l))∧(¬(x mem fpre))]`. The argument `fpre` is the list containing all range extensions of that relation. In one step, that is for one range extension pair `(li,si)`, the function `domain_exs_fn` builds all possible sublists of the result of that filtering and combines all those with `(li,si)` to build the domain extended point `(li,si,di)`.

```
domain_exs_empty_fn: "domain_exs_fn D l fpre [] = []"
domain_exs_step: "domain_exs_fn D l fpre ((l,si) # lr) =
   let sl = (sublists [x:D.(¬(x mem l))∧(¬(x mem fpre))])
    in if (lr = []) then map (λ x. [(li,si,x)]) sl
       else combine (map(λ x. (li,si,x)) sl)(domain_exs_fn D l fpre lr)"
```

The function `domain_exs_fn` can now be used to define the actual function for domain extension as a constant by building the parameter `fpre` of all range extensions of fixpoints and then applying the former function.

```
domain_exs D l rl == domain_exs_fn D l (concat (map snd rl)) rl
```

The meaning of a range extension point `(li,si)` is that `li` is a fixpoint and has range `si` containing `li`. The domain extension element `di` that is now added as the third element to those points represents all hangers-on of `li`. That is, the elements of `di` are all non fixpoints that have in their range all the elements of the range of `li`, i.e. the elements of `si`.[2] For efficiency of representation and simplicity it is advisable to choose this representation rather than copying the ranges for each hanger-on.

Keeping in mind this explanation of the domain extensions the following function should be easily comprehensible. This function, called `build_rel`, is necessary to close the loop of developing the algorithm from the ideas contained in Theorem 1 back to where it started by giving a procedure to translate the list representation calculated from the previous set of functions into a relation again.

```
build_rel :: "(α × (α)list × (α)list)list => (α × α)set"
```

For a relation represented as a list of the described triple type, the function `build_rel` now builds a set of pairs: a pair `(x,y)` is in the relation if `x` is the fixpoint we are considering and `y` is in the current range extension `si` or `x` is a hanger-on from the domain extension. In the latter case, as hangers-on have the same range as their fixpoints, `y` has to be in `si` too.

```
build_rel_empty: "build_rel [] = {}"
build_rel_step: "build_rel ((li,si,di) # ll) =
 {(x,y). (x = li ∨ x mem di) ∧ y mem si)} ∪ (build_rel ll)"
```

[2] Note that in general a hanger-on can have more elements in its range, as it can simultaneously hang-on to other fixpoints.

As we will see in the next section, the functions presented in this section can be simply applied to produce idempotent relations. As they are primitive recursive function definitions, and the syntax chosen in Isabelle for these functions is very similar to ML, it is possible to translate them one-to-one into ML.

Recall that we wish to produce all idempotent relations for a given domain and range. We can build up the corresponding list in our representation defining a constant.

```
list_of_all_idempotents :: "[α list,α list]
                => (α × α list × α list)list list"
```

The definition of that constant just applies the functions for range extension and domain extension to the list of all possible sets of fixpoints in the domain `D` and building all possible range subsets of `R` as starting points for the process. As we produce a list of lists of relations in each step it is necessary to flatten those lists using `concat` when putting it together.

```
list_of_all_idempotents D R ==
    (let all_ranges = sublists R
     in concat(map (λ l. concat (map (domain_exs D l)
              (concat (map (range_exs R) all_ranges))))(sublists D)))
```

4.3 Properties

Now, the function `build_rel` closes the loop of development. With its help we can express the correctness of the algorithm concisely

```
[| finite r; unique l; set l = {x. (x,x) ∈ r};
   set D = Domain r; set R = Range r |]
⟹ idempotent r = (∃ re_fn. re_fn mem (range_exs R l) ∧
    (∃ de_fn. de_fn mem (domain_exs D l re_fn) ∧ (r = build_rel de_fn)))
```

That is, for a given set l of fixpoints, a given domain and range for the relation, the property of idempotence of a relation is equivalent to the existence of a range extension and a domain extension that are built by the corresponding functions applied consecutively and which represent the relation. The premise `unique l` is a predicate ensuring that the list `l` does not contain repetitions. This restriction is necessary to exclude lists with repetitions as input to the algorithm.

Otherwise, the proof is a rather straightforward unfolding of definitions of function definitions using inductions over lists to show that the properties of the list representation actually translate into the properties of the characterization.

From this property we can derive the more general one concerning the set of all idempotent relations which corresponds to completeness.

```
{r. finite r & Domain r = set R ∧ Range r = set R ∧ idempotent r} =
    set(map build_rel(list_of_all_idempotents D R))
```

4.4 Discussion

In this section we have introduced via characteristic examples an algorithm for the construction of finite idempotent relations. From there we have developed a set of functions that compute them.

The way the function `range_exs` deletes half finished elements when they do not respect transitivity is the main source of inefficiency. It would be nicer to exclude unsuitable lists at the stage when all sublists s_i for the range extensions are generated — that is before `range_exs` is called. However, the problem arises only later — that is, when the single sets s_i are generated we cannot anticipate how they are going to be combined. More precisely, they are going to be combined in all possible ways. As the transitivity property is a property that concerns each element of the relation, it can be judged only when putting the relation together. We could have started building admissible sets of sets s_i before adding the fixpoints l_i to the range extensions but that is basically what we are doing just at the same time when the corresponding fixpoints are added.

Another point worth mentioning is that our function is not efficient for another reason. When the domain extensions are built repetitions may arise: when two fixpoints x and z have exactly the same range, there are two ways a hanger-on can achieve its domain extension: by hanging on to x or by hanging on to z. Hence, our algorithm produces two identical relations. For example, `domain_exs [1,2,3] [1,2] [(1, [1, 2]), (2, [2, 1])]` produces the domain extensions `[(1, [1, 2], []), (2, [2, 1], [3])]` and `[(1, [1, 2], [3]), (2, [2, 1], [])]`. Those two will consequently be mapped to the same relation by `build_rel`. Another frequent repetition occuring in the produced lists, also apparent in the previous sample, is the empty list. This is a remainder of the deletion of non admissible lists during the range extension and vanishes when concatenating. The algorithm is thus abstract in the sense that it is really defined at the level of sets rather than lists.

For those reasons our algorithm is not optimal. Let $|D| = |R| = n$. Then the number of idempotents constituted just by fixpoints is 2^n. Now for the complexity of the algorithm the functions that cost most are the two application of the `combine` functions in `domain_exs_step` and `range_exs`. The `combine` l_1 l_2 (and `combine_re`) are in $O(|l_1||l_2|)$. However, their order-of-n times iterated application in the recursion in `domain_exs_step` and `range_exs` on lists of length order 2^n leads to an estimate for order of `list_of_all_idempotents` as

$$O(2^{\frac{n^2+6n-2}{2}}) .$$

This seems very costly. However for comparison we can obviously not do better than $O(2^n)$ — enumerating all idempotents is at least as costly as enumerating the fixpoint relations — and the complexity of the naive approach of generate-and-test (see Section 1) is of in $O(2^{2^n})$. The square in the power comes through the combination used in range and domain extensions. However, these combinations can only be built by stepping through the single points of the relation (a process of order n) and in each step we have a complexity of $O(2^n)$ by the mere number of relations that have to be combined.

5 Conclusion

We have presented a result characterizing finite idempotent relations. This result has first been proved on paper and then formalized and proved in Isabelle/HOL. A constructive abstract algorithm has been derived from there and its correctness sketched in the same framework.

Independent of the contribution that the result and the algorithm may represent, this work is a case study that illustrates that a logic like HOL in the implementation given by Isabelle/HOL is a framework that may well be used to develop programs rigorously. Although small, it is a convincing demonstration that Isabelle/HOL is suited as a formal method in itself, as has been recognized before, e.g. [NW98, BN00], however not until very recently has this approach been much applied. The close relation to the programming language ML has proved very helpful in this project. All the functions presented here could be tested easily by extracting them from the Isabelle code.

Clearly, it would be wrong to deduce from the current case study that HOL even with a mature support like Isabelle is a substitute for advanced formal methods providing support for abstract high level concepts like refinement, object orientation, and modularity. But it may be an encouragement to extend the existing features, like data types and recursive function definitions, in order to make the rigorous development in Isabelle/HOL more powerful. The method of derivation that is given by a manner of application like the present, is crucial for the complete specification and proof of critical software. For comparison the refinement calculus (with data refinement) has also been used by the authors to derive an algorithm; integration into the development process in Isabelle may also be of interest.

References

[BKS01] J. Burghardt, F. Kammüller and J. W. Sanders. *On the Antisymmetry of Galois Embeddings.* Information Processing Letters, **79**:2, Elsevier, June 2001.

[BN00] Stefan Berghofer and Tobias Nipkow. *Executing Higher Order Logic.* Types for Proofs and Programs (TYPES 2000), Springer LNCS, **2277**, 2002.

[CP61] A. H. Clifford and G. B. Preston. *The Algebraic Theory of Semigroups*, volume 1. Mathematical Surveys, number 7. American Mathematical Society. 1961.

[CP61] C. W. Curtis and I. Reiner. *Representation Theory of Finite Groups and Associative Algebras.* Interscience Publishers, John, Wiley, 1966.

[DP02] R. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition.* Cambridge University Press, 2002.

[DS58] N. Dunford and J. T. Schwartz. *Linear Operators. Part I: General Theory.* Interscience Publishers, John Wiley, 1958.

[NW98] W. Naraschewski and M. Wenzel. *Object-Oriented Verification Based on Record Subtyping in Higher-Order Logic*, In Proceedings of TPHOLs 98, Springer LNCS, **1479**, 1998.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, Springer LNCS, **828**, 1994.

Program Verification Using Automatic Generation of Invariants*,**

Enric Rodríguez-Carbonell and Deepak Kapur

LSI Department, Technical University of Catalonia,
Jordi Girona, 1-3 08034 Barcelona (Spain)
erodri@lsi.upc.es
Department of Computer Science, University of New Mexico,
Albuquerque, NM 87131-0001 (USA)
kapur@cs.unm.edu

Abstract. In an earlier paper, an algorithm based on algebraic geometry was developed for discovering polynomial invariants in loops without nesting, not requiring any a priori bound on the degree of the invariants. Polynomial invariants were shown to form an ideal, a basis of which could be computed using Gröbner bases methods. In this paper, an abstract logical framework is presented for automating the discovery of invariants for loops without nesting, of which the algorithm based on algebraic geometry and Gröbner bases is one particular instance. The approach based on this logical abstract framework is proved to be *correct* and *complete*. The techniques have been used with a verifier to automatically check properties of many non-trivial programs with considerable success. Some of these programs are discussed in the paper to illustrate the effectiveness of the method.

1 Introduction

There has recently been a surge of interest in research on automatic generation of loop invariants of imperative programs. This is perhaps due to the successful development of powerful automated reasoning tools including BDD packages, SAT solvers, model checkers, decision procedures for common data structures in applications (such as numbers, lists, arrays, ...), as well as theorem provers for first-order logic, higher-order logic and induction. These tools have been successfully used in application domains such as hardware circuits and designs, software and protocol analysis.

In an earlier paper [RCK04], an approach for generating polynomials as invariants for loops without nesting was presented. An algorithm employing

* This research was partially supported by an NSF ITR award CCR-0113611, the Prince of Asturias Endowed Chair in Information Science and Technology at the University of New Mexico, a Spanish FPU grant ref. AP2002-3693, and the Spanish project MCYT TIC2001-2476-C03-01.

** An extended version of this paper is available at www.lsi.upc.es/~erodri.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 325–340, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Gröbner basis method was proposed to derive conjunctions of polynomial equalities as loop invariants, without any a priori bound on the degree of the polynomials appearing in the invariants. The main contributions of that paper are:

1. Invariant polynomials are shown to form an *ideal*, a well-known algebraic concept [CLO98]. Consequently, algebraic geometry techniques are brought into play to discover such invariants from a given program without imposing any a priori bound on their degrees.
2. The proposed algorithm for generating polynomial invariants is proved to terminate using algebraic geometry if right-hand sides of assignments are *solvable* mappings either commuting or with positive eigenvalues (see [RCK04] for definitions and details).

In this paper we improve our previous work in a number of aspects:

1. The construction of [RCK04] is generalized and presented in an abstract logical framework, thus highlighting the key properties required for the proposed approach to be applicable to data structures other than numbers. We were thus able to abstract properties needed from algebraic geometry for our results in [RCK04].
2. The abstract framework is based on the forward propagation semantics of program statements. A fixed point computation of formulas approximating the invariant at the loop entry point is carried out by considering all possible execution paths.
3. A procedure for computing loop invariants based on this abstract logical framework is presented. The procedure is proved to be *sound* and *complete*, in the sense that on termination, the procedure generates the strongest possible invariant expressible in the considered language for specifying invariants.
4. The significance of this framework is demonstrated by showing our algorithm in [RCK04] as a nontrivial instance of this abstract procedure.
   In another paper [RCK], we have used the abstract interpretation framework for developing approximations and a widening operator to compute polynomial invariants of a bounded degree, where the bound on their degree is determined by the widening operator. The termination proof of this algorithm is different from the one in [RCK04]; it is based on using the dimension of the vector space generated by polynomials of bounded degree. The advantage of our framework over abstract interpretation is that we are able to ensure that we generate the *strongest* invariant expressible in the language, which is not usually possible in abstract interpretation.
5. The procedure has been implemented and is employed with our tools for program verification to prove the correctness of a number of programs, as shown in a table of examples. Some of these are used for illustrating the key ideas of the approach. Currently, the procedure only generates conjunctions of polynomial equalities as invariants, but plans are underway to generate polynomial inequalities as well.

The rest of the paper is organized as follows. In the next subsection, related work is briefly reviewed. Section 2 introduces the general framework: the

programming model is presented, and necessary properties of the language for expressing invariants are studied so that the generic procedure for finding loop invariants can be formulated. In Section 3, we prove that the language of conjunctions of polynomial equalities satisfies all the required properties of the abstract framework and is thus an instance of it. This gives an algorithm for computing invariant polynomial equalities that turns out to be equivalent to that given in [RCK04]. In Section 4 we show that the framework is applicable even when some of the conditions on the language to express invariants are not met. Section 5 is a brief overview of the verifier we have built for proving properties of programs. In Section 6 we give some illustrative examples of program verification using this tool. Finally, Section 7 concludes with a discussion on future research.

1.1 Related Work

The generation of arithmetic invariants between numerical variables is a long researched area. Karr first showed in [Kar76] an algorithm for finding invariant *linear equalities* at any program point of a procedure. This work was extended by Cousot and Halbwachs [CH78], who applied the model of abstract interpretation [CC77] for finding invariant *linear inequalities*. Like our techniques, both methods are based on forward propagation and fixed point computation [Weg75], which points out that our ideas may be useful for accelerating the termination as well as improving the precision in abstract interpretation.

Recently, there has been a renewed surge of interest in automatically deriving invariants of imperative programs. In [CSS03], Colón et al. used non-linear constraint solving based on Farkas' lemma to attack the problem of finding invariant linear inequalities. Extending Karr's work, Müller-Olm and Seidl [MOS04] proposed an interprocedural method for computing invariant polynomial equalities of bounded degree in programs with affine assignments. The same authors [MOS03] developed a complete technique for finding invariants of a prefixed form in procedures with polynomial assignments and disequality guards. Similarly, in [SSM04] a method was proposed for generating polynomials as invariants, which starts with a template polynomial with undetermined coefficients and attempts to find values for the coefficients so that the template is invariant using the Gröbner basis algorithm. Kapur proposed a related approach using quantifier elimination in November 2003 (see [Kap03]).

In [RCK04], we gave an algorithm based on algebraic geometry and not requiring any degree bounds for generating conjunctions of polynomial equalities as loop invariants. This algorithm served as the basis for developing the proposed abstract logical framework of this paper. In that paper, the discussion and proofs extensively use results of polynomial ideal theory and algebraic geometry, because of which they are not likely to be directly applicable to other data structures such as arrays, records, etc. In contrast, this paper presents a logical framework that is likely to be more widely applicable. Finally, in [RCK] we have employed the framework of abstract interpretation to generate polynomial equalities of bounded degree as invariants in general procedures.

2 Abstract Framework

We consider a simple programming language with multiple assignments, nondeterministic conditional statements and loop constructs. Loops are assumed to have the following form:

while $E(\bar{x})$ **do**
  if $C_1(\bar{x}) \rightarrow \bar{x} := f_1(\bar{x})$;
  ...
  [] $C_i(\bar{x}) \rightarrow \bar{x} := f_i(\bar{x})$;
  ...
  [] $C_n(\bar{x}) \rightarrow \bar{x} := f_n(\bar{x})$;
  end if
end while

where $\bar{x} = (x_1, x_2, ..., x_m)$ denotes the tuple of program variables, E, C_i's are boolean expressions and each f_i is an m-tuple of expressions.

2.1 Loop Invariants

A formula expressing a property of a loop (including an invariant of the loop) is specified using the program variables $\bar{x}$ and variables, denoted by $\bar{x}^*$, representing the initial, usually unknown, values of the program variables before entering the loop.

Let $\mathcal{R}$ stand for a subset of a first-order language with equality used for expressing properties of loops. A formula in $\mathcal{R}$ representing an invariant property will be written as $R(\bar{x}, \bar{x}^*)$. Our goal is to capture the semantics of loops using the strongest invariant expressible in the language $\mathcal{R}$. For that we characterize the expressiveness of $\mathcal{R}$ to admit such strongest invariants.

Definition 1. *A formula $R \in \mathcal{R}$ is* invariant *(with respect to another formula $R_0(\bar{x}^*)$ relating initial values of $\bar{x}$) if:*
i) $R_0(\bar{x}^*) \Rightarrow R(\bar{x}^*, \bar{x}^*)$ *and*
ii) $\forall i : 1 \leq i \leq n$, $(R(\bar{x}, \bar{x}^*) \wedge E(\bar{x}) \wedge C_i(\bar{x})) \Rightarrow R(f_i(\bar{x}), \bar{x}^*)$.

To capture the semantics of the loop, we have to compute the strongest possible invariant in the language $\mathcal{R}$:

Definition 2. *The language $\mathcal{R}$ is* expressive *for a loop if $\exists R_\infty \in \mathcal{R}$ such that*

1. *R_∞ is an invariant of the loop and*
2. *for every invariant R of the loop in the language $\mathcal{R}$, $R_\infty(\bar{x}, \bar{x}^*) \Rightarrow R(\bar{x}, \bar{x}^*)$.*

In Section 3, the language of conjunctions of polynomial equalities is introduced for specifying invariants, and it is shown to be expressive for loops with polynomial assignments (when tests are abstracted and considered to be *true*).

2.2 Fixed Point Procedure for Computing Invariants

We give an iterative procedure for computing the strongest invariant R_∞ of a given loop. Assume that the loop test E and each C_i, the tests in the conditional statement, and each assignment mapping f_i are expressible in $\mathcal{R}$. Let R_0 stand for a formula satisfied by the initial values of the variables before entering the loop. Based on the forward propagation semantics of program statements, the procedure below computes successive approximations of the strongest invariant R_∞ until reaching a fixed point.

Forward Propagation Semantics. If $R(\bar{x}, \bar{x}^*)$ holds at the loop entry point, the loop test $E(\bar{x})$ is true and the i-th conditional branch is executed, then the strongest postcondition at the end of the body of the loop is

$$\exists \bar{y}(\bar{x} = f_i(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge E(\bar{y}) \wedge C_i(\bar{y}))\,.$$

Traditionally abstract interpretation uses this one-step forward propagation to compute invariants, employing a widening operator to guarantee termination. In order to accelerate the procedure for finding invariants and avoid the loss of precision involved in widening, we propose instead a many-step forward propagation along the lines of the meta-transitions of Boigelot [Boi99]. While these meta-transitions were originally utilized to compute the exact reach set of a system, we apply accelerations to the more general problem of computing overapproximations of the set of reachable states, i.e. invariants, in a given specification language.

If the i-th branch is executed s times in a row, the strongest postcondition is:

$$\exists \bar{y}\Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big)\,,$$

assuming that the s-th power of f_i is also expressible in $\mathcal{R}$.

Given that the number of iterations s is undetermined, an infinite disjunction is needed to express the relation (which is not a formula anymore in the language unless existential quantifiers are used):

$$\bigvee_{s=1}^{\infty} \Big(\exists \bar{y}\Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big)\Big)\,. \tag{1}$$

In general, there are several branches in a loop and each of the branches can be executed arbitrarily many times. This results in an infinitary formula capturing the program states at the loop entry point after an undetermined branch has been executed arbitrarily many times:

$$R(\bar{x}, \bar{x}^*) \vee \Big(\bigvee_{i=1}^{n} \bigvee_{s=1}^{\infty} \Big(\exists \bar{y}\Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y}))\big)\Big)\Big)\Big)\Big)\,.$$

In order to capture the semantics of the loop, this approximation of the invariant is computed iteratively until reaching a fixed point (or going on forever). This is the core of the procedure below.

In a highly powerful language for expressing invariants, the infinite disjunction in the above infinitary formula can perhaps be expressed using an equivalent formula with the help of existential quantifiers. If the language does not permit existential quantifiers or even disjunctions (as will be the case for the language of conjunctions of polynomial equalities), the language must be able to express a sufficiently strong approximation.

Definition 3. *The language* $\mathcal{R}$ *is* disjunctively closed *if* $\forall R, S \in \mathcal{R}$, $\exists T \in \mathcal{R}$, *written as* $R \sqcup S$, *such that*

i) $(R(\bar{x}, \bar{x}^*) \vee S(\bar{x}, \bar{x}^*)) \Rightarrow T(\bar{x}, \bar{x}^*)$,

ii) $\forall T' \in \mathcal{R}$, *if* $(R(\bar{x}, \bar{x}^*) \vee S(\bar{x}, \bar{x}^*)) \Rightarrow T'(\bar{x}, \bar{x}^*)$, *then* $T(\bar{x}, \bar{x}^*) \Rightarrow T'(\bar{x}, \bar{x}^*)$.

The language of first-order predicate calculus with equality is disjunctively closed, as $\vee$ can be taken as $\sqcup$. So is the language of polynomial equalities closed under conjunction: for any R and S that are conjunctions of polynomial equalities, there is another conjunction of polynomial equalities $R \sqcup S$ which is equivalent to $R \vee S$, as shown later. The language of conjunctions of linear inequalities, used in [CH78], is also disjunctively closed: given conjunctions R and S of linear inequalities, $R \sqcup S$ is defined as the convex hull of R and S; unlike the previous cases, $R \sqcup S$ is, in general, not equivalent to $R \vee S$ in this case.

To get approximations in $\mathcal{R}$ of infinitary formulas involving infinite disjunctions as well as existential quantifiers, $\mathcal{R}$ is required to have some additional properties. For the i-th conditional branch we assume that $\exists \varphi_i(R) \in \mathcal{R}$, the strongest formula in the language implied by the formula (1) above:

Definition 4. $\mathcal{R}$ allows existential elimination *if* $\forall R \in \mathcal{R}$, $\forall i : 1 \leq i \leq n$ $\exists T \in \mathcal{R}$, *written as* $\varphi_i(R)$, *such that*

i) $\displaystyle\bigvee_{s=1}^{\infty} \exists \bar{y} \Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y})) \big) \Big) \Big) \Rightarrow T(\bar{x}, \bar{x}^*)\,.$

ii) $\forall T' \in \mathcal{R}$ *such that*

$$\bigvee_{s=1}^{\infty} \exists \bar{y} \Big(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}, \bar{x}^*) \wedge \Big(\bigwedge_{t=0}^{s-1} \big(E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y})) \big) \Big) \Big) \Rightarrow T'(\bar{x}, \bar{x}^*)\,,$$

$T(\bar{x}, \bar{x}^*) \Rightarrow T'(\bar{x}, \bar{x}^*)$.

In order to check whether the fixed point has already been reached, we additionally need to decide whether two formulas in the language are equivalent:

Definition 5. $\mathcal{R}$ allows equivalence check *if* $\forall R, S \in \mathcal{R}$, *it can be decided whether* $R \Leftrightarrow S$ *or not.*

After replacing disjunctions $\vee$ by $\sqcup$ and eliminating existential quantifiers by means of the φ_i's, we get the procedure below. It starts assigning to the formula variable R an initial formula satisfied by the initial values of the variables in the loop. This variable R stores the formula corresponding to the successive approximations of the invariant.

Invariant Generation Procedure

Input: Assignment mappings $f_1, ..., f_n$
Formula R_0 satisfied by the initial values of program variables
Output: Strongest invariant formula R_∞

var R, R' : formulas in $\mathcal{R}$ **end var**
$R' := false$
$R := (x_1 = x_1^*) \wedge \cdots \wedge (x_m = x_m^*) \wedge R_0$
while $R' \not\Leftrightarrow R$ **do**
$R' := R$
$R := R \sqcup {}^1\Big(\bigsqcup_{i=1}^{n} \varphi_i(R) \Big)$ (2)
end while
return R

The following theorem captures the correctness and completeness of the above procedure:

Theorem 1. *Given a loop* **L**, *let* $\mathcal{R}$ *be a language expressive for* **L**, *disjunctively closed and admitting existential elimination and equivalence check. Let* R_∞ *stand for the strongest invariant of* **L** *in the language. If the invariant generation procedure terminates with output* R, *then* $R(\bar{x}, \bar{x}^*) \Leftrightarrow R_\infty(\bar{x}, \bar{x}^*)$.

The proof of the theorem, given in the extended version of the paper, is based on two facts: *i*) if the procedure terminates, R is an invariant of the loop; and *ii*), $R \Rightarrow R_\infty$ holds in all steps of the invariant generation procedure. So, if the procedure terminates, $R \Rightarrow R_\infty$ and $R_\infty \Rightarrow R$, which finally leads to the result that the above procedure on termination indeed computes the strongest invariant of the loop.

The key issue in the procedure is to find the appropiate definition for the $\sqcup$ operator as well as for the φ_i functions in order to ensure termination. In the following section we show a nontrivial instance of language satisfying these requirements, the language of polynomial equalities. Further, in Section 4 we will see how, even if the specification language only satisfies some of the requirements for disjunctive closedness and quantifier elimination, the procedure can still yield useful results on termination.

3 Conjunctions of Polynomial Equalities as Invariants

In this section we show that the language of conjunctions of polynomial equalities, denoted by $\mathcal{P}$, is a particular instance of the abstract framework. Assuming that the guards are ignored (i.e. $E = C_i = true$) and that the assignment

[1] The use of $\sqcup$ approximating $\vee$ here may not be sufficient to guarantee termination. Using a widening operator ∇ instead of $\sqcup$ ([CC77]) as a further approximation of $\vee$ can ensure the termination of the procedure, probably at the cost of completeness.

mappings are polynomial, it is shown that this language $\mathcal{P}$ satisfies all the requirements discussed above (Section 3.2). The above iterative procedure for computing invariants can be instantiated as well (Section 3.3).

3.1 Preliminaries

Given a field $\mathbb{K}$, let $\mathbb{K}[\bar{z}] = \mathbb{K}[z_1, ..., z_l]$ denote the ring of polynomials in the variables $z_1, ..., z_l$ with coefficients from $\mathbb{K}$. An *ideal* is a set $I \subseteq \mathbb{K}[\bar{z}]$ that contains 0, is closed under addition and such that, if $p \in \mathbb{K}[\bar{z}]$ and $q \in I$, then $pq \in I$. Given a set of polynomials $S \subseteq \mathbb{K}[\bar{z}]$, the *ideal spanned by* S is $\langle S \rangle = \{p \in \mathbb{K}[\bar{z}] \mid \exists k \geq 1 \; p = \sum_{j=1}^{k} p_j q_j \text{ with } p_j \in \mathbb{K}[\bar{z}], q_j \in S\}$. For an ideal I, a set $S \subseteq \mathbb{K}[\bar{z}]$ such that $I = \langle S \rangle$ is called a *basis* of I.

The *variety* of a set $S \subseteq \mathbb{K}[\bar{z}]$ over $\mathbb{K}^l$ is defined as its set of zeroes, $\mathbf{V}(S) = \{\bar{\alpha} \in \mathbb{K}^l | \; p(\bar{\alpha}) = 0 \; \forall p \in S\}$. On the other hand, if $A \subseteq \mathbb{K}^l$, the ideal $\mathbf{I}(A) = \{p \in \mathbb{K}[\bar{z}] | \; p(\bar{\alpha}) = 0 \; \forall \bar{\alpha} \in A\}$ is the *annihilator* of A.

A mapping $g : \mathbb{K}^l \to \mathbb{K}^l$ is *affine* if it is of the form $g(\bar{z}) = A\bar{z} + b$, where $A \in \mathbb{K}^{l \times l}$ and $b \in \mathbb{K}^l$. In general, a mapping $g \in \mathbb{K}[\bar{z}]^l$ is a *polynomial mapping.*

To each conjunction of polynomial equalities $R \equiv (p_1 = 0 \wedge \cdots \wedge p_k = 0) \in \mathcal{P}$, we associate the ideal $J = \langle p_1, \cdots, p_k \rangle$. Similarly, given an ideal J specified by a finite basis, say B, there is a formula in $\mathcal{P}$ (not necessarily unique) associated with it, written as $\bigwedge_{p \in B}(p = 0)$; depending upon the basis chosen for J, different (but equivalent) formulas can be obtained.

3.2 Properties of $\mathcal{P}$

Expressiveness. Given a loop, the language $\mathcal{P}$ is expressive, i.e., there exists a formula R_∞ in $\mathcal{P}$ such that (i) R_∞ is an invariant of the loop, and (ii) any formula R in $\mathcal{P}$ that is an invariant of the loop is implied by R_∞. The idea of the proof is to take a basis of the ideal generated by all the polynomials that are invariants of the loop. By Hilbert's basis theorem, such an infinite basis has an equivalent finite basis. The conjunction of the polynomial equalities corresponding to the polynomials in the finite basis is precisely R_∞.

Disjunctive Closedness. The language $\mathcal{P}$ is disjunctively closed: if $R \equiv p_1 = 0 \wedge ... \wedge p_k = 0$ and $S \equiv q_1 = 0 \wedge ... \wedge q_l = 0$, there is a conjunction of polynomial equalities $R \sqcup S$ that is equivalent to $R \vee S$. This formula can be constructed by computing a finite basis of the intersection ideal $\langle p_1, ..., p_k \rangle \cap \langle q_1, ..., q_l \rangle$ and taking the corresponding conjunction of polynomial equalities (since ideals of polynomials are always finitely generated by Hilbert's basis theorem).

Existential Elimination. For the i-th conditional branch, we need to show the existence of $\varphi_i(R) \in \mathcal{R}$, the strongest formula in the language implied by the infinitary formula (1) above in Section 2.2. Such a formula in $\mathcal{P}$ can be obtained by computing a finite basis B of the ideal

$$\mathbb{K}[\bar{x}, \bar{x}^*] \bigcap \left(\bigcap_{s=1}^{\infty} \left\langle (-\bar{x} + f_i^s(\bar{y})) \bigcup \Big(\bigcup_{p \in \mathbf{IV}(I)} p(\bar{y}, \bar{x}^*) \Big) \right\rangle \right), \tag{3}$$

where $I = \langle p_1, ..., p_k \rangle$ is the ideal associated to the formula $R \equiv p_1 = 0 \wedge ... \wedge p_k = 0$ and $-\bar{x} + f_i^s(\bar{y})$ denotes the set of m polynomials corresponding to the projections over each of the m coordinates. Then $\varphi_i(R) \equiv (\bigwedge_{q \in B} \ q = 0)$ is the strongest formula in $\mathcal{P}$ implied by (1).

Equivalence Check. The language $\mathcal{P}$ admits equivalence check: if $R \equiv p_1 = 0 \wedge ... \wedge p_k = 0$ and $S \equiv q_1 = 0 \wedge ... \wedge q_l = 0$, then $R \Leftrightarrow S$ is equivalent to $\mathbf{IV}(p_1, ..., p_k) = \mathbf{IV}(q_1, ..., q_l)$. In case that $\mathbf{IV}(p_1, ..., p_k) = \langle p_1, ..., p_k \rangle$ and $\mathbf{IV}(q_1, ..., q_l) = \langle q_1, ..., q_l \rangle$ (which is common in practice), then obviously $R \Leftrightarrow S$ is equivalent to $\langle p_1, ..., p_k \rangle = \langle q_1, ..., q_l \rangle$, which can be easily checked.

3.3 Generating Conjunctions of Polynomial Equalities as Invariants

The abstract procedure discussed in Section 2.2 can be instantiated to be the algorithm presented in [RCK04] as follows. The assignment labelled as (2) in the abstract procedure is the most non-trivial and complicated to perform: it requires computing a basis of the ideal defined by the expression (3) for each assignment mapping, which involves an infinite intersection of ideals. To compute this infinite intersection, elimination theory is used to eliminate s and possibly other auxiliary variables needed to express the f_i^s's as polynomials. In order to represent the f_i^s's as polynomials, we ask assignment mappings to be *solvable mappings*, a particular case of polynomial mappings; solvable mappings are an extension of affine mappings. The following theorem is proved in [RCK04]; the reader can refer to that paper for details about the theorem as well as the proof.

Theorem 2. *Let* $\mathbf{L}$ *be a loop with tests* $E = C_i = true$ *and assignments* $\bar{x} := f_i(\bar{x})$, $1 \leq i \leq n$. *If each of the assignment mappings* f_i *is a solvable mapping with positive rational eigenvalues, the procedure computes the strongest invariant in at most* $2m+1$ *steps, where* m *is the number of program variables in* $\mathbf{L}$. *Moreover, if the assignment mappings commute, i.e.* $f_i \circ f_j = f_j \circ f_i$ *for* $1 \leq i, j \leq n$, *then the algorithm terminates in at most* $n+1$ *steps, where* n *is the number of branches in the non-deterministic conditional statement of the body of* $\mathbf{L}$.

The proof of the first part of the theorem extensively uses algebraic geometry concepts including irreducible decomposition of varieties and their dimension. It is our experience that for instantiating the abstract framework, the most nontrivial task is to find conditions under which the procedure for generating invariants terminates.

4 Heuristic Procedure for Non-expressive Languages

In the previous section, we showed how the language of conjunctions of polynomial equalities satisfies all the conditions required in the abstract logical framework. We thus get a sound and complete algorithm for computing conjunction of polynomial equalities as invariants; further, the invariant generated by the procedure is the strongest possible invariant expressible in this language.

In this section we show that our abstract framework is still useful when the language $\mathcal{R}$ for specifying invariants is not expressive for the loop. Namely, if the language admits equivalence check and the conditions i) of both definitions of disjunctive closedness and existential elimination are satisfied, then the invariant generation procedure can still be formulated and yields correct invariants on termination. The invariant generated by the procedure, however, needs not be the strongest possible one as the non-expressiveness of the language implies that there is no such strongest invariant.

For example, let us consider the first-order language of quantifier-free formulas with polynomial inequalities as atoms, which subsumes conjunctions of linear inequalities [CH78] or polynomial equalities [RCK04, RCK]. This language admits equivalence check and is in fact disjunctively closed; and moreover, it non-trivially satisfies condition i) in the definition of existential elimination, as we can use quantifier elimination [Tar51] to get rid of infinite disjunctions and existential quantifiers. We illustrate this with the following simple program, for which our procedure finds a correct invariant in the language under consideration:

{Pre: $n \geq 0$}
a:=0; **while** $(a+1)^2 \leq n$ **do** a:=$a+1$; **end while**

In this case, given a formula $R = R(a, n, a^*, n^*)$ (where a^*, n^* stand for the initial values of the variables a, n before entering the loop), we can compute the formula for the next iteration by eliminating s, t from

$$\exists s\ (s \geq 0 \wedge R(a-s, n, a^*, n^*) \wedge \forall t\ ((t \geq 0 \wedge t \leq s-1) \Rightarrow (a-t)^2 \leq n))$$

using quantifier elimination. Starting with $R_0(a^*, n^*) \equiv (a^* = 0) \wedge (n^* \geq 0)$, after two iterations we get the fixed point $a \geq 0 \wedge a^2 \leq n \wedge n \geq 0 \wedge a^* = 0 \wedge n = n^*$, which is invariant for the loop. Notice that $a \geq 0 \wedge a^2 \leq n \wedge n \geq 0$ contains a non-linear inequality.

However, the first-order quantifier-free language of polynomial inequalities is not expressive in general. The next loop illustrates this fact:

x:=0; **while** *true* **do** x:=$x+1$; **end while**

In this case, the set of reachable states is $\mathbb{N}$. Any invariant formula $R(x, 0)$ in the language will hold for all natural numbers. In particular, such a formula will necessarily hold for an interval of real numbers of the form $[x_0, \infty)$, for a certain natural number x_0. Then the formula $R(x, 0) \wedge (x = x_0 \vee x \geq x_0 + 1)$ will also be an invariant in the language, but will be strictly stronger than $R(x, 0)$; so there is no strongest invariant. For this example, our invariant generation procedure yields the invariant $x \geq 0$.

5 Verification of Properties of Programs

An implementation in Maple for automatically discovering polynomial loop invariants has been manually interfaced with a prototype of verifier described below. We have successfully used this verifier to automatically prove non-trivial properties of many numerical programs (computing for instance products, divi-

sions, square roots, divisors, ...). Some of these programs are shown in Section 6 to illustrate the power of the techniques.

5.1 Verifier

The verifier takes as input imperative programs with annotated assertions, including preconditions and postconditions. The programming language that it accepts has a similar syntax as that of `C`. It features integer variables, arithmetic operations ($+$, $*$, div, mod, etc.) and function calls. The assertion language is a first-order logic with equality with interpreted function and predicate symbols. Admitted functions are the arithmetic operators of `C` and the gcd, lcm functions, introduced to widen the range of treatable properties. The predicates are equality $=$ and order $>$, $\geq$.

The verifier mainly consists of two components: (*i*) a *verification condition generator* that produces the formulas that ensure that the desired properties of the program are fulfilled; and (*ii*) a *theorem prover*, which checks the validity of these formulas.

Verification Condition Generator. The verification condition generator is based on Floyd-Hoare-Dijkstra's inductive assertion method. It generates formulas (called *verification conditions*) from the code and the annotations that must be satisfied to guarantee the correctness of the program with respect to the specification. This is done by means of a semantics of language constructs as predicate transformers. Given a program postcondition, this semantics allows to mechanically compute an assertion such that if the precondition implies it, then the postcondition holds on termination of the program.

Theorem Prover. The goal of the theorem prover is to check the validity of the verification conditions.

We initially tried SPASS [WBH$^+$02], a general-purpose theorem prover for first-order-logic with equality. Since the conditions are about integer numbers, SPASS had to be given an axiomatization of the integers explicitly; still, this theorem prover had problems handling formulas requiring algebraic manipulation and knowledge on the integers.

This led us to implement an *ad-hoc* prover in Prolog. In our prover, formulas are proved by simplification using rewriting rules until the tautology *true* is obtained; if this is not the case and the prover cannot rewrite further, then it gives up and the problem of the validity of the formula is unsolved. Strategies for proving formulas are implemented using conditional rewriting rules. This allows us to give the prover a knowledge on numbers that overcomes the power of general theorem provers like SPASS for our concrete application. Our prover has given overall good results, as we shall see in the examples.

5.2 Interface of the Loop Invariant Generator and the Verifier

For the time being, the interface between our implementation in Maple for generating polynomial loop invariants and the verifier is manual; that is to say, the

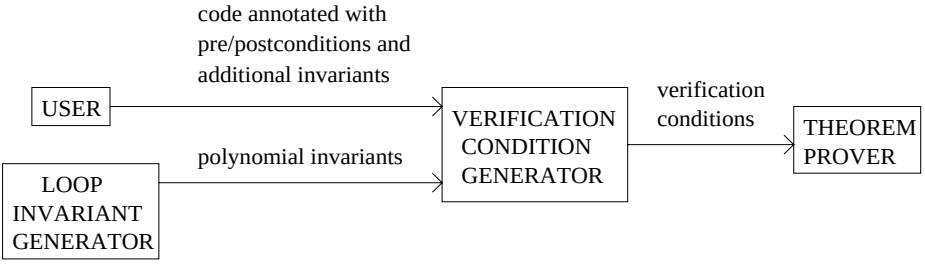


Fig. 1. Scheme of the system for verifying programs

user has to annotate by hand the polynomial invariants obtained by means of the method here described in the code to be verified.

Sometimes an invariant expressed as a conjunction of polynomial equalities is not strong enough to prove the desired properties of a program; then additional invariants are also annotated. Most often there are already methods for automatically finding these assertions, as is the case for linear inequalities [CH78].

6 Examples

In this section we illustrate the power of the proposed techniques for generating polynomial invariants and their effectiveness in proving properties of nontrivial algorithms operating on numbers. Although in some cases the polynomial invariants are not enough to prove the desired properties, we will demonstrate their need in proving properties of programs. For the sake of simplicity, we will focus on the verification conditions that guarantee that the postcondition is met on termination of the program. At the end of the section, a table summarizes the results of applying our tools to a variety of programs. For all the examples here shown, the verifier is powerful enough to check the required properties. The timings are taken in seconds with a Pentium 4 with a 2.5 GHz. processor and 512 MB of memory.

Example 1. The next program is an algorithm for computing the product of two natural numbers A and B. Three of the assignments are non-affine solvable:

function product (A, B: integer) **returns** q: integer
  { Pre: $A \geq 0 \wedge B \geq 0$}
  var a, b, p: integer **end var**
  $(a,\ b,\ p,\ q)$:=$(A,\ B,\ 1,\ 0)$;
  while $(a \neq 0) \wedge (b \neq 0)$ **do**
    if $(a \bmod 2 = 0) \wedge (b \bmod 2 = 0)$
      $\rightarrow (a, b, p, q) := (a \text{ div } 2, b \text{ div } 2, 4p, q)$;
    [] $(a \bmod 2 = 1) \wedge (b \bmod 2 = 0)$
      $\rightarrow (a, b, p, q) := (a - 1, b, p, q + bp)$;
    [] $(a \bmod 2 = 0) \wedge (b \bmod 2 = 1)$

$\rightarrow (a, b, p, q) := (a, b - 1, p, q + ap)$;
[] $(a \bmod 2 = 1) \wedge (b \bmod 2 = 1)$
$\rightarrow (a, b, p, q) := (a - 1, b - 1, p, q + (a + b - 1)p)$;
end if
end while
{ Post: $q = AB$}

Our algorithm yields the invariant $q + abp = AB$ in 3.32 s. In this case, the verification condition that ensures that the postcondition is met is $(q + abp = AB \wedge (a = 0 \vee b = 0)) \Rightarrow q = AB$ (free variables are implicitly universally quantified); this condition is split into $(q + abp = AB \wedge a = 0) \Rightarrow q = AB$ and $(q + abp = AB \wedge b = 0) \Rightarrow q = AB$, which are reduced to $(q = AB \wedge a = 0) \Rightarrow q = AB$ and $(q = AB \wedge b = 0) \Rightarrow q = AB$ respectively, and then both to *true*. The program is shown to be correct automatically by our system in 0.82 s.

Example 2. The next example, taken from [Dij76], is an extension of Euclid's algorithm for computing the least common multiple of two natural numbers a and b. The invariant generation procedure yields $xu + yv = 2ab$ in 2.02 s.

function lcm $(a, b$: integer) **returns** z: integer
{ Pre: $a > 0 \wedge b > 0$}
var x, y, u, v: integer **end var**
$(x, y, u, v) := (a, b, b, a)$;
{ Inv: $\gcd(x, y) = \gcd(a, b)$}
while $x \neq y$ **do**
if $x > y \rightarrow (x, y, u, v) := (x - y, y, u, u + v)$;
[] $x < y \rightarrow (x, y, u, v) := (x, y - x, u + v, v)$;
end if
end while
{$Post : (u + v) \text{ div } 2 = \text{lcm}(a, b)$}

In this case the auxiliary invariant $\gcd(x, y) = \gcd(a, b)$, which can be obtained automatically by other means ([CP93]), is also needed to prove the postcondition. The verification condition in this case is $(\gcd(x, y) = \gcd(a, b) \wedge xu + yv = 2ab \wedge x = y) \Rightarrow (u + v) \text{ div } 2 = \text{lcm}(a, b)$. Our prover reduces this formula to $\gcd(a, b)\,(u + v) = 2\gcd(a, b)\,\text{lcm}(a, b) \Rightarrow (u + v) \text{ div } 2 = \text{lcm}(a, b)$ and then to $u + v = 2\,\text{lcm}(a, b) \Rightarrow (u + v) \text{ div } 2 = \text{lcm}(a, b)$, which is trivially valid. The program is shown to be correct automatically in 0.9 s.

Example 3. The following program has been extracted from [Knu69]. It tries to find a divisor d of the natural number N using a parameter D:

function divisor $(N, D$: integer) **returns** d, r: integer
{ Pre: $N > 0 \wedge N \bmod 2 = 1 \wedge D \bmod 2 = 1 \wedge D \geq 2\sqrt[3]{n} + 1$}
var t, q: integer **end var**
$(d, r, t, q) := (D, N \bmod D, N \bmod (D - 2), 4(N \text{ div } (D - 2) - N \text{ div } D))$;
{ Inv: $d \bmod 2 = 1$}
while $d \leq \lfloor \sqrt{N} \rfloor \wedge r \neq 0$ **do**
$(d, r, t, q) := (d + 2, 2r - t + q, r, q)$;

if $r < 0 \rightarrow (r, q) := (r + d, q + 4)$; **end if**
if $r \geq d \rightarrow (r, q) := (r - d, q - 4)$; **end if**
if $r \geq d \rightarrow (r, q) := (r - d, q - 4)$; **end if**
end while
{ Post: $r = 0 \Rightarrow N \bmod d = 0$}

For this program, our invariant generation algorithm yields $d(dq - 4r + 4t - 2q) + 8r = 8N$ as invariant in 24.56 s. In this case we also need the extra invariant $d \bmod 2 \equiv 1$. To prove the postcondition we have to show the validity of $(d(dq - 4r + 4t - 2q) + 8r = 8N \ \wedge \ d \bmod 2 \equiv 1 \ \wedge \ (r = 0 \vee d > \lfloor\sqrt{N}\rfloor)) \Rightarrow (r = 0 \Rightarrow N \bmod d = 0)$. Our prover reduces this formula into $(d(dq - 4r + 4t - 2q) + 8r = 8N \ \wedge \ d \bmod 2 \equiv 1 \ \wedge \ r = 0) \Rightarrow N \bmod d = 0$, and then to $(d(dq - 4r + 4t - 2q) = 8N \ \wedge \ d \bmod 2 \equiv 1) \Rightarrow N \bmod d = 0$, which is able to prove to be valid. The total time spent on proving the correctness of the program is 2.03 s.

Table of Examples. Table 1 summarizes the results obtained after generating invariants and verifying correctness for a number of programs [2]. There is a row for each program; the columns provide the following information:

1. 1st column is the name of the program; 2nd column states what the program does; 3rd column gives the source where the program was picked from (the entry $(*)$ is for the examples developed up by the authors).
2. 4th column gives the number of variables in the program; 5th column gives the number of loops; 6th column is the number of branches for each loop;
3. 7th column gives the number of loop invariants generated for each loop; 8th column is the time taken by the invariant generation.
4. 9th column indicates if any other kind of additional invariants was needed; 10th column is the time taken by the verifier to prove correctness.

Table 1. Table of examples

1	2	3	4	5	6	7	8	9	10
cohencu	cube	[Coh90]	4	1	1	4	2.15	No	0.61
prod4br	product	$(*)$	6	1	4	1	3.32	No	0.82
hard	integer division	[SSM04]	6	2	1-2	3-3	7.43	Yes	5.46
divbin	integer division	[Kal90]	5	2	1-2	2-1	4.28	Yes	1.99
dijkstra	integer sqrt	[Dij76]	5	2	1-2	2-1	4.73	Yes	18.88
euclidex2	Bezout's coefs	$(*)$	8	1	2	5	3.64	Yes	1.79
lcm2	lcm	[Dij76]	6	1	2	1	2.02	Yes	0.90
fermat2	divisor	[Knu69]	5	1	2	1	2.26	Yes	1.01
factor	divisor	[Knu69]	6	1	4	1	24.56	Yes	2.03

[2] These programs are available at `www.lsi.upc.es/~erodri`.

7 Conclusions and Further Research

An abstract framework for automatically discovering invariants for loops without nesting is proposed. A general procedure for that is given if the language used for expressing invariants is *expressive*, *disjunctively closed* and *allows existential elimination* and *equivalence check*. The procedure computes the strongest possible invariant expressible in the language.

It is shown that our earlier results on computing polynomial equalities as invariants are an instance of the abstract logical framework here presented. We are investigating other languages for expressing invariants for which the framework can be adapted. We are particularly interested in the first-order language of polynomial inequalities, which subsumes that of of linear inequalities [CH78].

The framework has been implemented as a part of a verifier for proving properties of programs. The verifier is interfaced with the Maple computer algebra system, which generates conjunctions of polynomial equalities as invariants. The verifier also features a theorem prover able to reason about numbers, so that the formulas representing the desired program properties can be checked to be valid. This scheme has been applied to many non-trivial programs, successfully generating invariants and then verifying properties of these programs.

We believe that the proposed abstract logical framework will also allow us to design expressive languages to specify invariants of loops manipulating data structures such as records, pointers, etc. We regard that the approach will be particularly useful with arrays, since our framework has already been successful in inferring invariants for some loops involving this data structure.

We are also investigating enriching the programming model to consider nested loops as well as procedure calls; the main idea here is to represent all execution paths using regular expressions and define fixed point computations as prescribed by such regular expressions.

Acknowledgements. The authors would like to thank R. Clarisó, G. Godoy, R. Nieuwenhuis and M. Subramaniam for their help and advice.

References

[Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, Faculté des Sciences Appliquées de l'Université de Liège, 1999.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL 1977*, p. 238–252, 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL 1978*, p. 84–97, 1978.

[CLO98] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer-Verlag, 1998.

[Coh90] Edward Cohen. *Programming in the 1990s.* Springer-Verlag, 1990.

[CP93] R. Chadha and D. A. Plaisted. On the Mechanical Derivation of Loop Invariants. *Journal of Symbolic Computation*, 15(5-6):705–744, 1993.

[CSS03] M. A. Colón, S. Sankaranarayanan, and H.B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *CAV 2003*, volume 2725 of *LNCS*, p. 420–432. Springer-Verlag, 2003.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Kal90] A. Kaldewaij. *Programming. The Derivation of Algorithms*. Prentice-Hall, 1990.

[Kap03] D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. Technical Report TR-CS-2003-58, Department of Computer Science, UNM, 2003. Also in *10th International IMACS Conference on Applications of Computer Algebra (ACA 2004)*, Lamar, TX, July 2004.

[Kar76] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[Knu69] D. E. Knuth. *The Art of Computer Programming. Volume 2, Seminumerical Algorithms*. Addison-Wesley, 1969.

[MOS03] M. Müller-Olm and H. Seidl. Computing Polynomial Program Invariants. Technical report, Fernuni Hagen, 2003. Num. 310. To appear in *IPL*.

[MOS04] M. Müller-Olm and H. Seidl. Computing Interprocedurally Valid Relations in Affine Programs. In *POPL 2004*, p. 330–341, 2004.

[RCK] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. `www.lsi.upc.es/~erodri`. To appear in *SAS'04*.

[RCK04] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *ISSAC'04*, 2004.

[SSM04] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear Loop Invariant Generation Using Gröbner Bases. In *POPL 2004*, p. 318–329, 2004.

[Tar51] A. Tarski. A Decision Method for Elementary Algebra and Geometry. University of California Press, 1951.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In *CADE-18*, volume 2392 of *LNAI*, p. 275–279, 2002. Springer-Verlag.

[Weg75] B. Wegbreit. Property Extraction in Well-founded Property Sets. *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.

Random Generators for Dependent Types

Peter Dybjer[1], Qiao Haiyan[1] and Makoto Takeyama[2]

[1] Department of Computing Science,
Chalmers University of Technology, 412 96 Göteborg, Sweden
{peterd, qiao}@cs.chalmers.se
[2] Research Centre for Verification and Semantics,
National Institute of Advanced Industrial Science and Technology,
Nakoji 3-11-46, Amagasaki Hyogo, 661-097 Japan
makoto.takeyama@aist.go.jp

Abstract. We show how to write surjective random generators for several different classes of inductively defined types in dependent type theory. We discuss both non-indexed (simple) types and indexed families of types. In particular we show how to use the relationship between indexed inductive definitions and logic programs: the indexed inductive definition of a type family corresponds to a logic program, and generating an object of a type in the family corresponds to solving a query for the logic program. As an example, we show how to write a surjective random generator for theorems in propositional logic by randomising the Prolog search algorithm.

1 Introduction

Random testing is a quick way to find bugs both in programs and their specifications [4]. It also facilitates proof development in type theory [9, 10]. When doing random testing in type theory, we need to write random generators for types. A random generator for a type D is a function that has random seeds as inputs and objects of D as outputs. When D is a simple data type, the type of the generator is $\mathtt{Rand} \to D$ [9], where $\mathtt{Rand}$ is the type of random seeds. In the case of a dependent type (an indexed family of types) $P\ i$ for $i :: I$ (we write $i :: I$ to indicate that i is an object of type I), we wish to generate a pair (i, p) of indices $i :: I$ and objects $p :: P\ i$. That is, the type of the generators for the dependent type P is $\mathtt{Rand} \to \mathtt{sig}\ \{i :: I;\ p :: P\ i\}$, where $\mathtt{sig}\ \{i :: I;\ p :: P\ i\}$ denotes a dependent record type: the first field has type I and the second field has a type $P\ i$ that depends on the value i of the first field. However, since $P\ i$ can be empty, we need to decide how to generate an index i so that this is not the case. In this paper, we discuss some difficulties that arise when writing generators for dependent types and present some solutions for several classes of inductive definitions (see Section 4–7). In particular, we get a very general class of generators by using the fact that generating objects of inductively defined indexed families is similar to solving queries in logic programs. This is because certain inductive definitions of indexed families of types (predicates under the Curry-Howard correspondence)

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 341–355, 2005.
© Springer-Verlag Berlin Heidelberg 2005

can be seen as a logic programs and vice versa [11]. We also discuss how to use logic programming techniques for writing generators.

Examples are implemented in Agda/Alfa [5, 12], an interactive proof editor based on Martin-Löf type theory. We slightly modify its concrete syntax to make it easier to follow the examples. The formal proofs which are omitted in the paper can be found at `http://www.cs.chalmers.se/~qiao/papers/`.

Acknowledgement. This research is partly supported by the Cover project funded by SSF (the Swedish Foundation for Strategic Research). The aim of the Cover project is to build tools where random testing and proving (automatic and interactive) can be combined, see `http://coverproject.org/`. In particular we develop tools based on dependent type theory, and we therefore need to develop random generators for dependent types.

2 Inductive Families

In this section, we briefly describe the scheme for introducing new set formers in Martin-Löf's dependent type theory given by Dybjer [7]. We follow the usual terminology where a "set" is a small type. Sets are either inductively defined or formed from previously defined sets by dependent function set formation and dependent record set formation. In general we may simultaneously define a whole indexed family of sets inductively. Such a family is often called an *inductive family* for short. In this article we restrict ourself to ordinary (or finitary) inductive definitions. See [7] for a discussion about ordinary vs. generalised (or infinitary) inductive definitions. See also [8] for a discussion of further generalising the notion of an inductive definition in dependent type theory.

We will only show the formation rule and the introduction rules, and omit the elimination rules and equality rules. The reader is referred to [7] for details.

The dependent type theory here is based on the logical framework for Martin-Löf type theory [13] extended with dependent record types [6]. It has four forms of judgements: $\sigma :: \mathtt{Type}$, $p :: \sigma$, $\sigma = \tau$ and $p = q :: \sigma$. The rules of type formation are the following:

- $\mathtt{Set} :: \mathtt{Type}$,
- if $\alpha :: \mathtt{Set}$, then $\alpha :: \mathtt{Type}$,
- if $\sigma :: \mathtt{Type}$ and $\tau[A] :: \mathtt{Type}$ under the assumption $A :: \sigma$, then
  $(A :: \sigma) \rightarrow \tau[A] :: \mathtt{Type}$ (dependent function type) and
  $\mathtt{sig}\ \{A :: \sigma;\ B :: \tau[A]\} :: \mathtt{Type}$ (dependent record type, also called *signature*).

Notation:

- We mostly use letters $\sigma, \tau, \cdots$ for types; $\alpha, \beta, \cdots$ for sets (observe that sets are special types); $p, q, \cdots$ for elements of a set; $A, B, \cdots$ for variables of a type; and $a, b, u \cdots$ for variables of a set.
- We write $\tau[A]$ when we emphasise that τ may depend on a variable A (that is, A may occur free in τ). However, this notation is optional: τ may depend

on any variable in scope regardless of the notation. The result of substituting the object s for A in τ is written $\tau[s/A]$.
- The general form of a signature is $\texttt{sig}\ \{A_1 :: \sigma_1;\ \cdots;\ A_N :: \sigma_N\}$. It has as objects *records* (also called *structures*) $\texttt{struct}\{A_1 = s_1;\ \cdots;\ A_N = s_N\}$ where $s_i :: \sigma_i[s_1/A_1, \cdots, s_{i-1}/A_{i-1}]$. A structure is a labelled tuple of objects of appropriate types. The dot operation $(-).A_i$ selects its A_i component; writing r for the structure above, we have that $r.A_i = s_i$.
- A nondependent function type, written $\sigma \to \tau$, is the special case of $(A :: \sigma) \to \tau[A]$ where A does not occur in τ.

2.1 Formation Rule

For each set former P, there is one formation rule that has the form

$$\begin{array}{l} P :: (A_1 :: \sigma_1) \to \cdots \to (A_N :: \sigma_N) \to \\ \quad (a_1 :: \alpha_1) \to \cdots \to (a_M :: \alpha_M) \to \\ \quad \texttt{Set} \end{array} \qquad (P\text{-Formation})$$

where σ_i are types and α_i are sets. We call A_i *parameters* and a_i *indices*. For readability, we omit the parameters and write $P\ a_1\ \ldots\ a_M$ instead of $P\ A_1\ \ldots\ A_N\ a_1\ \ldots\ a_M$.

2.2 Introduction Rules

There are finitely many introduction rules for each set former. Each introduction rule for the set former P above has the form

$$\begin{array}{l} \mathit{intro} :: (A_1 :: \sigma_1) \to \cdots \to (A_N :: \sigma_N) \to \\ \quad (b_1 :: \beta_1) \to \cdots \to (b_K :: \beta_K) \to \\ \quad (u_1 :: P\ q_{11}\ \ldots\ q_{1M}) \to \\ \quad \cdots \\ \quad (u_L :: P\ q_{L1}\ \ldots\ q_{LM}) \to \\ \quad P\ p_1\ \ldots\ p_M \end{array} \qquad (P\text{-Intro}_{\mathit{intro}})$$

where β_i are sets, $p_j :: \alpha_j[p_1/a_1, \cdots, p_{j-1}/a_{j-1}]$ $(1 \le j \le M)$, and similarly for q_{ij} for each i. We call b_i *non-recursive* and u_i *recursive* arguments of the constructor *intro*.

2.3 Examples

We show some instances of the general schema [7] and how they are written in Agda/Alfa [5, 12].

Example 1 (Natural numbers). The set `Nat` of natural numbers has no parameters and indices. The rules are

– formation	$\texttt{Nat}$	$:: \texttt{Set}$	$(N = M = 0;$ in $\texttt{Nat}$-Formation$)$
– introduction	$\texttt{zero}$	$:: \texttt{Nat}$	$(K = L = 0;$ in $\texttt{Nat}\text{-Intro}_{\texttt{zero}})$
	$\texttt{succ}$	$:: \texttt{Nat} \to \texttt{Nat}$	$(K = 0,\ L = 1;$ in $\texttt{Nat}\text{-Intro}_{\texttt{succ}})$

The concrete syntax in Agda/Alfa is

```
Nat :: Set = data zero             :: Nat
               | succ (n :: Nat) :: Nat
```

Example 2 (Finite sets). The indexed family $\texttt{Fin}\,n$ ($n :: \texttt{Nat}$) of sets with just n elements has the following rules:

- formation $\quad \texttt{Fin} :: \texttt{Nat} \to \texttt{Set} \qquad (N = 0,\ M = 1)$
- introduction $\quad \mathrm{C}_0 :: (n :: \texttt{Nat}) \to \texttt{Fin}(\texttt{succ}\,n) \qquad (K = 1,\ L = 0)$
  $\quad\quad\quad\quad\ \mathrm{C}_1 :: (n :: \texttt{Nat}) \to \texttt{Fin}\,n \to \texttt{Fin}(\texttt{succ}\,n) \qquad (K = 1,\ L = 1)$

The Agda/Alfa syntax is

```
Fin :: Nat -> Set
  = data C0 (n :: Nat)                :: Fin (succ n)
       | C1 (n :: Nat) (i :: Fin  n) :: Fin (succ n)
```

Example 3 (Untyped λ-terms). The set $\texttt{Term}\,n$ ($n :: \texttt{Nat}$) of λ-terms whose free variables are among $\{\mathrm{var}_0, \cdots, \mathrm{var}_{n-1}\}$ (using de Bruijn indices), is a component of the Nat-indexed family Term defined as follows.

```
Term :: Nat -> Set
 = data var (n :: Nat) (i :: Fin  (succ n)) :: Term (succ n)
      | abs (n :: Nat) (t :: Term (succ n)) :: Term n
      | app (n :: Nat) (t1, t2 :: Term n)   :: Term n
```

Example 4 (Vectors of specified length). An example with one parameter A_1 ($\sigma_1 = \texttt{Set}$) is the Nat-indexed family Vec where elements of $\texttt{Vec}\,n$ are length-n vectors.

```
Vec (A :: Set) :: Nat -> Set
  = data  nil' :: Vec A  zero
       | cons' (n :: Nat) (a :: A) (as :: Vec A n)
               :: Vec A (succ n)
```

In Agda/Alfa, constructors are polymorphic with respect to the parameters and need not be explicitly applied to them.

3 Generators

For the rest of the paper, we restrict σ_i in the schema in Section 2 to be the type Set.

3.1 Definition of Generators

A generator for the family P in Section 2.1 is a function

$$\begin{aligned} genP :: {} & (A_1 :: \texttt{Set}) \to \cdots \to (A_N :: \texttt{Set}) \to \\ & (g_1 :: \texttt{Rand} \to A_1) \to \cdots \to (g_N :: \texttt{Rand} \to A_N) \to \\ & \texttt{Rand} \to \texttt{sig}\,\{a_1 :: \alpha_1;\ \cdots;\ a_M :: \alpha_M;\ p :: P\ a_1\ \ldots\ a_M\} \end{aligned}$$

where A_i are parameters and g_i are *parameter generators*.

We have chosen to implement a seed in `Rand` as a binary tree of natural numbers [9]. The definition in Agda/Alfa is

```
Rand :: Set = data Leaf (k :: Nat)                  :: Rand
                | Node (k :: Nat) (l, r :: Rand) :: Rand
```

Example 5. The following function is a generator for `Vec`.

```
genVec :: (A :: Set) -> (Rand -> A) ->
          Rand -> sig { ind :: Nat; obj :: Vec A ind }

genVec A g (Leaf _ )    = struct ind = zero; obj = nil'
genVec A g (Node _ l r) = let { as = genVec A g r } in
                          struct ind = succ  as.ind
                                 obj = cons' as.ind (g l) as.obj
```

The idea behind this generator is to map the parameter generator `g` to the given tree seen as a (right-spine) list of (left) subtrees. (We omitted some braces and semicolons using the so called layout rule of the Agda/Alfa syntax.)

3.2 Surjective Generators

A generator (with instantiation of parameters and parameter generators) is *surjective* if it can generate, given a suitable seed, any element of any component set of the target family. A reason for writing generators in Agda/Alfa is that it becomes possible to formally prove this fundamental correctness property of generators.

For example, we can prove by induction that `genVec A g` is surjective whenever the parameter generator `g` is surjective. In Agda/Alfa we formally define

```
Surj :: (A :: Set) -> (Rand -> A) -> Set
Surj A g = (x :: A) -> sig rand :: Rand; prf :: Id A (g rand) x
  --  (In predicate logic: ∀x::A. ∃rand::Rand. g rand = x.)

surj_genVec::(A :: Set) -> (g :: Rand -> A) -> Surj A g ->
             Surj sig{ind :: Nat; obj :: Vec A ind} (genVec A g)
surj_genVec A g p = ⟨ ··· proof omitted ··· ⟩
```

4 Generators for Simple Sets

A *simple* set, possibly parameterised, is an inductive family with the following restriction (using the notation from Section 2):

- Its formation rule has only parameters and no indices ($M = 0$).
- For each introduction rule, the type β_i of each non-recursive argument is either a parameter A_j or a previously defined simple set.
- It is inhabited (non-empty); that is, at least one introduction rule has no recursive arguments.

A generator for simple P is easy to write: it randomly chooses a constructor and generates its arguments by parameter generators, by the generators for previously defined simple sets, or by recursive calls, all using sub-seeds of the given seed. When the seed is not large enough, it terminates by choosing a non-recursive constructor. As each seed is finite, the problem of non-termination discussed in [4] does not arise here.

Example 6 (Lists). The set `List A` of lists with elements in the set `A` is parameterised in `A`. A generator for it can be defined as follows:

```
List(A::Set) :: Set = data nil :: List A
                          | cons (a::A) (as::List A) :: List A

genList :: (A :: Set) -> (Rand -> A) -> Rand -> List A
genList A g (Leaf _)     = nil
genList A g (Node _ l r) = cons (g l) (genList A g r)
```

This is indeed a simplified version of `genVec` and easily seen to preserve surjectivity of the parameter generator g.

5 Generators for Inhabited Inductive Families

An *inhabited inductive family* is an inductive family which satisfies the following criteria:

- Its formation rule $P :: I \to \texttt{Set}$ has no parameters, and the single index set I is a simple set with a surjective generator $genI :: \texttt{Rand} \to I$.
- For all $i :: I$, the set $P\,i$ is inhabited.

The extension to families with parameters and several indices is straightforward.

For such a family P, a surjective generator $genP :: \texttt{Rand} \to genPsig$, where $genPsig = \texttt{sig}\ \{ind :: I;\ obj :: P\,ind\}$, can be defined from a surjective generator $genP'\,i$ for each $P\,i$. It first generates an index using $genI$, then an element of $P\,i$ using $genP'\,i$.

```
genP' :: (i :: I) -> Rand -> P i   -- assumed given.

genP :: Rand -> genPsig
genP (Node _ l r) = struct ind = genI l; obj = genP' ind r
genP s            = struct ind = genI s; obj = genP' ind s
```

In fact, one can formally prove that

```
surj_genP :: Surj I genI -> ((i :: I)-> Surj (P i) (genP' i))->
             Surj genPsig genP
surj_genP p q = ···
```

We now present some examples of random generators $genP'$ for various P.

Example 7. `Fin` (`succ` n) is inhabited for all n :: `Nat`. A surjective generator for this family can be defined as follows:

```
genFin' :: (n :: Nat) -> Rand -> Fin (succ n)
genFin'  zero     _           = C0  zero
genFin' (succ m) (Leaf _)     = C0 (succ m)
genFin' (succ m) (Node _ l r) = C1 (succ m) (genFin' m l)
```

Example 8. A binary tree is balanced if, at each node, the height difference between its left and right subtrees is at most 1. One formulation of the set `Bal` n of balanced binary trees of height n, and its surjective generator `genBal'` n are

```
Bal :: (n :: Nat) -> Set = data
   Empty                                    :: Bal zero
 | C00 (t1, t2 :: Bal n)                    :: Bal (succ n)
 | C01 (t1 :: Bal n) (t2 :: Bal (succ n)) :: Bal (succ (succ n))
 | C10 (t1 :: Bal (succ n)) (t2 :: Bal n) :: Bal (succ (succ n))

genBal' :: (n :: Nat) -> Rand -> Bal n
genBal'  zero            _           = Empty
genBal' (succ zero)      _           = C00 Empty Empty
genBal' (succ (succ n)) (Leaf k)     =
   let t  = genBal' (succ n) (Leaf k) in  C00 t t
genBal' (succ (succ n)) (Node k l r) =
   let b1 = genBal' (succ n) l
       b2 = genBal' (succ n) r
       b3 = genBal'       n  r
   in  choice3 k (C00 b1 b2) (C01 b3 b1) (C10 b1 b3)
```

where `choice3` $k\ a_0\ a_1\ a_2 = a_{(k \bmod 3)}$. Note that no part of a (non-leaf) seed contributes to the result twice; this is necessary for surjectivity, and keeps disjoint parts of the result independent of each other.

Example 9. The set `Term` n is nonempty for any n :: `Nat`, and a surjective generator can be given as follows:

```
genTerm' :: (n :: Nat) -> Rand -> Term n
genTerm' zero     (Leaf _)     = abs zero (var zero (C0 zero))
genTerm' zero     (Node k l r) =
    let t1 :: Term (succ zero) = genTerm' (succ zero) l
        t2 :: Term       zero  = genTerm'       zero  l
        t3 :: Term       zero  = genTerm'       zero  r
    in  choice2 k (abs zero t1) (app zero t2 t3)
genTerm' (succ m) (Leaf k)     = var m (genFin' m (Leaf k))
genTerm' (succ m) (Node k l r) =
    let t1 :: Term (succ (succ m)) = genTerm' (succ (succ m)) l
        t2 :: Term       (succ m)  = genTerm'       (succ m)  l
        t3 :: Term       (succ m)  = genTerm'       (succ m)  r
```

```
      in  choice3 k (var m (genFin' m l))
                    (abs (succ m) t1)
                    (app (succ m) t2 t3)
```

6 Generators for Simple Inductive Families

We now consider a family whose component sets are not necessarily inhabited. First, we adopt the method in Section 4 for simple sets to a restricted class of families; for these, surjective generators can be defined without backtracking.

An inductive family is *simple* if the following conditions hold:

- Its formation rule $P :: I \to \texttt{Set}$ has no parameter, and the single index set I is simple.
- Each introduction rule has the form

$$\begin{aligned} intro :: &(x_1 :: I) \to \cdots \to (x_K :: I) \to \\ &(u_1 :: P\ x_1) \to \cdots (u_K :: P\ x_K) \to \\ &P\ p \end{aligned}$$

- P is not empty; there must be a constructor without arguments.

The type of a generator for P is the same as in Section 5: $genP :: \texttt{Rand} \to genPsig$. However, the choice of constructor controls the generation process, as in Section 4. First, $genP$ randomly chooses a constructor. Then it generates the constructor arguments $i_1, \cdots, i_K, o_1, \cdots, o_K$ for $x_1, \cdots, x_K, u_1, \cdots, u_K$. Note that each of the pairs $(i_1, o_1), \cdots, (i_K, o_K)$ can be chosen as an arbitrary object of the type $genPsig$, and thus K recursive calls suffices for that. The result is the pair

$$(p[i_1/x_1, \cdots, i_K/x_K, o_1/u_1\ \cdots, o_K/u_K],\ intro\ i_1 \ldots i_K\ o_1 \ldots o_K) :: genPsig$$

As in Section 4 the process terminates since the sizes of seeds decrease.

It is easy to see that this method gives a surjective generator as long as we use independent random seeds in different recursive calls.

Example 10. A surjective generator for the family `Even` n ($n ::$ `Nat`) (of sets of proofs that n is even) can be defined as follows.

```
Even :: Nat -> Set
= data C0                              :: Even zero
     | C1 (n :: Nat) (p :: Even n) :: Even (succ (succ n))

genEven :: Rand -> sig { ind :: Nat; obj :: Even ind }
genEven (Leaf k)     = struct ind = zero; obj = C0
genEven (Node k l r) = let g1 = genEven l
       in  struct ind = succ (succ g1.ind)
                  obj = C1 g1.ind g1.obj
```

The method can be extended to include parameters, several indices, non-recursive arguments of simple types, etc, under suitable restrictions.

7 Inductive Definitions and Logic Programs

The motivation for considering various restrictions on inductive families is to have as few constraints as possible between indices and elements, in order to facilitate random generation. However, representing intricate constraints is often the very purpose of defining an indexed family. To cover some of those cases, we introduce unification and backtracking in a generation algorithm in the next section. This section explains its basis, the relationship between inductive families and logic programs [11].

A *Horn* inductive family is an inductive family which satisfies the following conditions:

- The index sets in its formation rule, and the types (sets) of non-recursive arguments in its introduction rules, all belong to previously defined Horn inductive families.
- In each introduction rule, indices appearing in types of recursive arguments and in the target type (q_{ij}, p_i) are all of constructor expressions; that is, built up from variables in scope by constructors only.

This covers a large part of ordinary inductive families, including all classes we have considered so far.

Our main example here is the family of sets of derivations in propositional calculus, indexed by their conclusions (theorems). It has no parameters and only one index. We do not explain our method for Horn families in general, but generalising the discussion from our specific example should be routine.

Consider Łukasiewicz's system for propositional calculus. The set of formulas is a simple set with constructors

$$\begin{array}{lll} \texttt{var} & :: & \texttt{Nat} \to \\ \texttt{\sim}(-) & :: & \to \\ (-)\ \texttt{=>}\ (-) & :: & \to\to \end{array}$$

where `Nat` is used to name propositional variables. The axiom schemata are:

$$\begin{array}{lcl} \texttt{Ax1}\,p\,q\,r & = & (p\ \texttt{=>}\ q)\ \texttt{=>}\ ((q\ \texttt{=>}\ r)\ \texttt{=>}\ (p\ \texttt{=>}\ r)) \\ \texttt{Ax2}\,p & = & (\texttt{\sim}p\ \texttt{=>}\ p)\ \texttt{=>}\ p \\ \texttt{Ax3}\,p\,q & = & p\ \texttt{=>}\ (\texttt{\sim}p\ \texttt{=>}\ q) \end{array}$$

The only inference rule is Modus Ponens. Thus the family $\texttt{Thm}\,p$ $(p ::)$ below defines the set of derivations of a theorem p.

```
Thm :: Formula -> Set = data
     ax1 (p, q, r :: Formula) :: Thm (Ax1 p q r)
   | ax2 (p       :: Formula) :: Thm (Ax2 p)
   | ax3 (p, q    :: Formula) :: Thm (Ax3 p q)
   | mp  (p, q    :: Formula) (x :: Thm p) (y :: Thm (p => q))
                              :: Thm q
```

This family is not an instance of the simple schema of Section 6 because of `mp` (`y`'s type is indexed by the non-variable `p => q`). Suppose we try to generate

arguments for `mp`, first generating a derivation $d_{\mathtt{x}} :: \mathtt{Thm}\, t_{\mathtt{p}}$ for arguments `x` and `p`. While any $t_{\mathtt{p}}$ will do here, we then must find, for `y` and `q`, a derivation $d_{\mathtt{y}} :: \mathtt{Thm}\, t_{\mathtt{q}}$ where $t_{\mathtt{q}}$ matches the specific pattern $(t_{\mathtt{p}}$ `=> _`$)$. Although we can find such a derivation in this particular case, for other definitions there may not be such a $t_{\mathtt{q}}$. If so, we need to backtrack, generate another pair $(d'_{\mathtt{x}}, t'_{\mathtt{p}})$, and try again.

This is similar to searching for a solution of a query in logic programming. In Prolog, we can define a predicate `thm` so that `thm` p holds if and only if there exists a derivation d :: `Thm` p (we here assume that we have a complete proof search procedure rather than the standard incomplete one).

```
thm((P => Q) => ((Q => R) => (P => R))).
thm((~P => P) => P).
thm(P => (~P => Q)).
thm(Q) :- thm(P), thm(P => Q).
```

Running the query `thm(X)` on a Prolog implementation, we can obtain theorems as solutions for `X`; for example

```
X = (((_A => _B) => (_C => _B)) => _D) => ((_C => A) => _D)
```

More precisely, this is a theorem pattern (schema) with variables `_A`, $\cdots$, `_D`. We can generate a theorem by instantiating them with any elements in `Formula`.

In general, there is a correspondence between Horn inductive definitions in dependent type theory and Prolog programs under the propositions-as-sets correspondence:

Type theory	Logic programming
Family of sets $P :: D \to$ `Set`	Predicate P
an introduction rule	a Horn clause
inductive definition of P	logic program defining P

For example, a clause in Prolog

$$P(\boldsymbol{t})\ \texttt{:-}\ P_1(\boldsymbol{t}_1), \cdots, P_K(\boldsymbol{t}_K)$$

becomes an introduction rule in type theory:

$$intro :: (x_1, \ldots, x_N :: D) \to P_1\ \boldsymbol{t}_1 \to \cdots \to P_K\ \boldsymbol{t}_K \to P\ \boldsymbol{t}$$

where D is the set inductively generated by the function symbols of the logic program (the term algebra or the Herbrand universe), and $\boldsymbol{t}_i$, $\boldsymbol{t}$ are sequences of terms in D with variables $x_1, \ldots, x_N$.

The above correspondence does not account for derivations (proof objects) d :: `Thm` p, nor for typing of objects in general. We now extend the correspondence for these.

The idea is to regard sets in type theory as unary predicates (on untyped terms) characterising their elements. For `Nat` and , the corresponding predicates are defined by

```
nat(zero).
nat(succ(X)) :-  nat(X).

formula(var(P)) :- nat(P).
formula(~P)     :- formula(P).
formula(P => Q) :- formula(P), formula(Q).
```

A family with M indices becomes an $(M+1)$-place predicate. For example, the unary predicate `Thm` becomes a binary predicate `thm1` so that `Thm`(P) is true iff there exists a proof object (derivation) d such that `thm1`(P, d):

```
thm1((P => Q) => ((Q => R) => (P => R)), ax1(P,Q,R))
                    :- formula(P), formula(Q), formula(R).
thm1((~P => P) => P, ax2(P))   :- formula(P).
thm1(P => (~P => Q), ax3(P,Q)) :- formula(P), formula(Q).
thm1(Q, mp(P,Q,X,Y)) :- thm1(P, X), thm1(P => Q, Y).
```

We can obtain a theorem and its derivation as solutions for `X` and `Y` in the query `thm1(X, Y)`. For example,

```
X = (var(zero) => var(zero)) =>
   ((var(zero) => var(zero)) => (var(zero) => var(zero)))
Y = ax1(var(zero), var(zero), var(zero))
```

So the problem of generating a pair `(X :: Formula, Y :: Thm X)` in dependent type theory corresponds to the task of solving a query `thm1(X, Y)`. In this way, we can directly use a Prolog interpreter to generate some elements of dependent types. If we randomise the Prolog interpreter, then we get a random generator for dependent types.

In general, a typing $b :: P\ \boldsymbol{a}$ can be represented by a predicate $P'\ (\boldsymbol{a}, b)$ in Prolog. For example, the following introduction rule for an inductive family P

$$intro :: (x_1 :: D_1) \to \cdots \to (x_N :: D_N) \to\ P_1\ \boldsymbol{t}_1 \to \cdots P_K\ \boldsymbol{t}_K \to P\ \boldsymbol{t}$$

becomes a clause of the following form:

$$P'(\boldsymbol{t}, intro(X_1, \ldots, X_N, U_1, \ldots, U_K))\ \texttt{:-}$$
$$D'_1(X_1), \cdots, D'_N(X_1, \cdots, X_{N-1}, X_N),\ P'_1(\boldsymbol{t}_1, U_1), \cdots, P'_K(\boldsymbol{t}_K, U_K).$$

where D'_i is the predicate corresponding to the set $D_i[x_1, \cdots, x_{i-1}]$.

The idea is applied to test data generation as follows. A testing form [9] below requires that $Q[\boldsymbol{d}/\boldsymbol{x}]$ to be true (inhabited) for any $\boldsymbol{d} = (d_1, \cdots, d_N)$ that satisfies the preconditions $P_i[\boldsymbol{d}/\boldsymbol{x}]$.

$$(x_1 :: D_1) \to \cdots \to (x_N :: D_N[x_1, \cdots, x_{N-1}]) \to$$
$$P_1[x_1, \cdots, x_N] \to\ \cdots\ \to P_K[x_1, \cdots, x_N] \to$$
$$Q[x_1, \cdots, x_N]$$

Test data $\boldsymbol{d}$ for this can be generated by searching for solutions to the query

$$\texttt{:-}\ D'_1(X_1), \cdots, D'_N(X_1, \ldots, X_{N-1}, X_N),$$
$$P'_1(X_1, \cdots, X_N, _), \cdots, P'_K(X_1, \cdots, X_N, _).$$

In the next section, we show a generator example for theorems by randomising the Prolog search algorithm: instead of always choosing the first clause unifiable with a goal, we let the random seed determine the order in which clauses are tried.

8 A Generator for Theorems

In this section, we describe a generator for the family `Thm` in Section 7. It is based on another more general generator `ThmPat` for theorem *patterns*, that is, formula patterns whose ground instantiations are all theorems.

The type of formula patterns `Pat` is a simple set with four constructors. We have the same three constructors as but also a fourth constructor $\mathtt{X} :: \mathtt{Nat} \to \mathtt{Pat}$ for *pattern variables* (logical variables denoting indeterminate formulas). We write $\mathtt{X}_0, \mathtt{X}_1, \cdots$ rather than $\mathtt{X}\,\mathtt{zero}$, $\mathtt{X}\,(\mathtt{succ}\,\mathtt{zero}), \cdots$. Examples of patterns are $\mathtt{X}_0\mathtt{=>}\mathtt{X}_1$ and $(\mathtt{var}_0\mathtt{=>}\mathtt{var}_1)\mathtt{=>}\mathtt{X}_1$. We distinguish proposition and pattern variables, so that the method applies to indexing types without a `var`-like constructor (for example, that of formulas on a fixed finite set of atomic propositions).

A theorem pattern $t :: \mathtt{Pat}$ becomes a theorem when each of its pattern variables is instantiated by a formula; for example $\mathtt{Ax2}\,\mathtt{X}_0$ is a theorem pattern since $\mathtt{ax2}\ p :: \mathtt{Thm}((\mathtt{Ax2}\ \mathtt{X}_0)[p/\mathtt{X}_0])$ for any p ::. They are precisely those $t :: \mathtt{Pat}$ with some derivation $d :: \mathtt{ThmPat}\,t$, where $\mathtt{ThmPat} :: \mathtt{Pat} \to \mathtt{Set}$ is defined just the same as `Thm`, but with `Pat` replacing everywhere.

In what follows, letters $X, Y, \cdots$ range over pattern variables. Our Agda code uses a standard technique to have access to 'totally fresh' pattern variables at any point, though we omit details. Substitutions $\sigma = [t_1/X_1, \cdots, t_N/X_N]$ are represented by lists of pairs, and `Subst` is their type. The composite $\sigma_1 \triangleright \sigma_2$ of two substitutions are defined so that $t[\sigma_1 \triangleright \sigma_2] = (t[\sigma_1])[\sigma_2]$.

A pattern t *matches* an introduction rule $\mathtt{ax}i$ of `ThmPat` if it can be unified with $\mathtt{Ax}i\ \boldsymbol{X}$, where $\boldsymbol{X}$ is a sequence with the appropriate number of fresh pattern variables. When this is the case, writing σ for the most general unifier of t and $\mathtt{Ax}i\ \boldsymbol{X}$, we call the pair $(\sigma,\ \mathtt{ax}i\ \boldsymbol{X}[\sigma])$ the *match*. For example, a match of $\mathtt{X}_0\mathtt{=>}\mathtt{X}_1$ with `ax2` is $([\sim X\ \mathtt{=>}\ X/\mathtt{X}_0, X/\mathtt{X}_1],\ \mathtt{ax2}\,X)$ with a fresh X.

We now describe a theorem pattern generator `genTP` the purpose of which is to generate not an arbitrary theorem pattern but one that fits into a given $t :: \mathtt{Pat}$.

```
genTP :: Rand -> (t :: Pat) -> Maybe (σ :: Subst, ThmPat t[σ])
```

We use the `Maybe`-type of Haskell which has two constructors: `Just` (for success) and `Nothing` (for failure). With a seed s, $\mathtt{genTP}\,s\,t$ either *succeeds* and returns some $\mathtt{Just}\,(\sigma, d)$, or *fails* and returns `Nothing`. In case of success, we have a theorem pattern $t[\sigma]$ with derivation $d :: \mathtt{ThmPat}\,t[\sigma]$.

The derivation returned by $\mathtt{genTP}\,t\,s$ (t pattern, s seed) has the same shape as s: leaves correspond to axioms and nodes to `mp`. For the leaf case, the result is a success if t matches one of the axioms, else a failure. For the node case, we first apply `genTP` to a fresh variable X to obtain $(\sigma_l, d_l :: \mathtt{ThmPat}\,X[\sigma_l])$. Then

we apply `genTP` to the pattern $(X \texttt{ => } t)[\sigma_l]$ and obtain $(\sigma_r, d_r :: \texttt{ThmPat}\,(X \texttt{ => } t)[\sigma_l][\sigma_r])$. The final result is the composite $\sigma_l \triangleright \sigma_r$, together with the derivations $d_l[\sigma_r]$ and d_r combined by `mp`.

Recursive calls are made with sub-seeds of the random seed given as argument, hence `genTP` always terminates. A failed recursive call is dealt with by backtracking as long as our finite random seed is not exhausted.

The pseudo-code for `genTP` is given below. In the description, *toList s* turns a tree (seed) s into the list of left-subtrees.

```
genTP (Leaf k) t =  do
    if (t matches some of ax1, ax2, ax3) {
        choose a match (σ, d) according to k;
        return Just (σ, d);
    } else
        return Nothing;

genTP (Node k l r) t =  do
    for s_l in (toList l) {
        if (genTP s_l X (X fresh) has the form Just (σ_l, d_l)) {
            for s_r in (toList r) {
                if (genTP s_r ((X => t)[σ_l]) has the form Just (σ_r, d_r)) {
                    // generation succeeded.
                    σ := σ_l ▷ σ_r;
                    return Just (σ,  mp  X[σ] t[σ] d_l[σ_r] d_r);
                }
            }
        }
    }
    // seed exhausted.
    return Nothing;
```

We can prove that this is a surjective generator: for any theorem pattern t, there exists a seed s and fresh X such that `genTP` $s\,X$ is `Just` (σ, d) with $X[\sigma] = t$ and $d :: \texttt{ThmPat}\,t$. The Agda/Alfa code and the surjectivity proof for a slightly different version can be found in Qiao [15].

We now use `genTP` to define a generator `genThm` for `Thm`.

```
genThm :: Rand -> sig { ind :: Formula; obj :: Thm ind }

genThm s =  do
    if (genTP s X (X fresh) has the form Just (σ, d)) {
        τ := substitution of all pattern variables
                by arbitrary elements of ;
        return (X[σ][τ], d[τ]);
    } else {
        choose an axi, and generate arbitrary formulas p for its arguments;
        return (Axi p,  axi p);
    }
```

This generator can for example be used to test properties in [10] (where `BoolExpr` is used for the type of formulas).

9 Discussions and Future Work

We have identified several restricted classes of inductive families of sets for which writing surjective generators is simple. For a Horn inductive family, generating elements of the family of sets is equivalent to solving a query in a corresponding logic program. Therefore, proof search techniques in logic programming can be used for writing generators. As an example, we implemented a surjective generator for theorems by randomising the proof search algorithm that is used in Prolog implementations. However, it is of course inconvenient to ask the user to implement the search algorithm for each new family of sets. One solution is to embed the search algorithm in the proof assistant externally or internally. Such a system would be a bit like a randomised version of Twelf [14], a logical framework where a given type family is interpreted as a logical program.

In Section 6, we described a simple schema for inductive definitions for which we can write surjective generators. It is interesting to consider more general schemata for which we can still write surjective generators without much difficulty. For example, we may add side conditions or allow general terms (and not only variables) as indices in the induction hypotheses. Consider, for example, the set of reachable states of a transition system. This can be defined in the following way:

```
R :: S -> Set = data
       init (s :: S)(p :: P s) :: R s
     | step (s, s' :: S)(q :: Tran s s')(p :: R s) :: R s'
```

where there are side conditions in the introduction rules: `P` characterises the initial states and `Tran` is the transition relation. One sufficient condition to have a surjective generator is: there is a surjective generator for `P`, and for any `s0 :: S`, we have a surjective generator for the family `Tran s0`, because we can then generate all possible next states for a given reachable state.

Recent work on generic programming [1, 2, 3] allows us to write a generic function for a class of data types. It will be interesting to see if we can use generic programming to generate surjective generators for a class of data types.

References

1. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.
2. Marcin Benke. Towards generic programming in type theory. Presentation at Annual ESPRIT BRA TYPES Meeting, Berg en Dal. Submitted for publication, available from `http://www.cs.chalmers.se/~marcin/Papers/Notes/nijmegen.ps.gz`, April 2002.

3. Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10:265 – 289, 2003.
4. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279. ACM Press, New York, September 2000.
5. Catarina Coquand. The Agda homepage. `http://www.cs.chalmers.se/~catarina/agda`.
6. Thierry Coquand, Randy Pollack, and Makoto Takeyama. A logical framework with dependently typed records. In *Proceedings of TLCA 2003*, volume 2701 of *LNCS*, pages 105–119, 2003.
7. Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.
8. Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65, June 2000.
9. Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Combining testing and proving in dependent type theory. In David Basin and Burkhart Wolff, editors, *Proceedings of Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 188–203. Springer-Verlag, 2003.
10. Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying Haskell programs by combining testing and proving. In *Proceedings of Third International Conference on Quality Software*, pages 272–279. IEEE Press, 2003.
11. Masami Hagiya and Takafumi Sakurai. Foundation of logic programming based on inductive definition. *New Generation Comput. (JAPAN) ISSN: 0288-3635*, 2(1):59–77, 1984. QA 76 N 48.
12. Thomas Hallgren. The Alfa homepage. `http://www.cs.chalmers.se/~hallgren/Alfa`.
13. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf type theory: an introduction.* Oxford University Press, 1990.
14. Frank Pfenning and Carsten Schürmann. The Twelf homepage. `http://www-2.cs.cmu.edu/~twelf/`.
15. Qiao Haiyan. *Testing and Proving in Dependent Type Theory.* PhD thesis, Chalmers University of Technology, 2003.

A Proof of Weak Termination Providing the Right Way to Terminate

Olivier Fissore, Isabelle Gnaedig, and Hélène Kirchner

LORIA-INRIA & LORIA-CNRS,
BP 239 F-54506 Vandœuvre-lès-Nancy Cedex,
Phone: + 33 3 83 58 17 00,
Fax: + 33 3 83 27 83 19
{fissore, gnaedig, Helene.Kirchner}@loria.fr

Abstract. We give an inductive method for proving weak innermost termination of rule-based programs, from which we automatically infer, for each successful proof, a finite strategy for data evaluation. We first present the proof principle, using an explicit induction on the termination property, to prove that any input data has at least one finite evaluation. For that, we observe proof trees built from the rewrite system, schematizing the innermost rewriting tree of any ground term, and generated with two mechanisms: abstraction, schematizing normalization of subterms, and narrowing, schematizing rewriting steps. Then, we show how, for any ground term, a normalizing rewriting strategy can be extracted from the proof trees, even if the ground term admits infinite rewriting derivations.

1 Introducing the Problem

In the context of programming in general, termination is a key property that warrants the existence of a result for every evaluation of a program. For rule-based programs, written in languages like ASF+SDF [19], Maude [4], Cafe-OBJ [12], or ELAN [3], data evaluation consists in exploring rewriting derivations of an input term. Strong termination, expressing that every rewriting derivation terminates, often does not hold. When for any term, there is at least one terminating derivation, the rewrite system is said to be weakly terminating. This is an interesting property for languages like ELAN , whose strategies can express that the result of the program evaluation on a data is *one of its possible* finite evaluations, or *the first* one. Weak termination then warrants a result for such evaluation strategies.

Analyzing termination also allows choosing the good way to evaluate data. Indeed, if the program is strongly terminating, a depth-first evaluation can be used, while if the program is only weakly terminating, a breadth-first algorithm, often much more costly, is necessary in general. In the second case, if there is a way to find terminating branches, the breadth-first technique can be avoided, which yields a considerable gain for program executions. This is what we propose.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 356–371, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Specific methods for proving termination of rewriting under strategies have been studied. Let us cite [2] and [13, 9] for the innermost strategy, [10] for the outermost strategy, and [8, 20] for local strategies on operators. All these works tackle the problem of strong termination. Here, we consider the weak innermost termination problem, i.e. we prove that among all innermost rewriting derivations starting from any term, one of them is finite. We focus on the innermost rewriting strategy, consisting in rewriting always at the lowest possible positions, since it is most often used as a built-in mechanism in evaluation of rule-based languages and functional languages.

Like the previously cited methods, the approach presented here also gives a way to prove weak termination of standard rewriting. But to our knowledge, it is the only approach able to handle term rewriting systems (TRSs in short) that are not strongly but only weakly innermost terminating. Moreover, our method is *constructive* in the sense that the proof gives the strategy to follow to obtain one of the finite derivations.

The weak termination property has been studied from several perspectives. For instance, B. Gramlich proved that weak termination can imply strong termination [16]. He also established conditions on TRSs for the property to be preserved by the union operation on TRSs [17]. J. Goubault-Larrecq proposed a proof of weak termination of typed Lambda-Sigma calculi in [15].

In order to illustrate the main ideas of our method on a running example, let us consider the following TRS:

$$f(g(x), s(0)) \to f(g(x), g(x)) \tag{1}$$
$$f(g(x), s(y)) \to f(h(x, y), s(0)) \tag{2}$$
$$g(s(x)) \to s(g(x)) \tag{3}$$
$$g(0) \to 0 \tag{4}$$
$$h(x, y) \to g(x). \tag{5}$$

Obviously, $\mathcal{R}$ is not terminating, nor even, because of the rule (2), innermost terminating. For instance, the following innermost infinite sequence is possible in $\mathcal{R}$: $\underline{f(g(f(0,0)), s(0))} \to^{(2)} f(\underline{h(f(0,0),0)}, s(0)) \to^{(5)} \underline{f(g(f(0,0)), s(0))} \ldots$ However, $\mathcal{R}$ is weakly innermost terminating ; in particular, the cycle above can be avoided by using the rule (1) instead of (2).

We first propose in this paper a method based on the same inductive principle as [9, 8, 10], where we study strong termination: we use an explicit induction on the termination property, but to prove here that every element t of a given set of terms T weakly innermost terminates, i.e. there is at least one finite innermost rewriting derivation starting from t. The general proof principle relies on the simple idea that for establishing weak innermost termination of a ground term t, it is enough to suppose that subterms of t weakly innermost terminate, and that rewriting the context leads to at least one terminating chain. Iterating this process until a non-reducible context is obtained establishes weak innermost termination of t.

Directly using the termination notion on terms has also been proposed in [14], for inductively proving well-foundedness of binary relations, among which path

orderings. The approach differs from ours in that it works on general relations, that can then be used on TRSs, whereas we directly handle the termination proof of a given TRS.

From the proof of weak termination of a given TRS, we then extract for any given ground term, a rewriting strategy to compute one of its normal form, even if the ground term admits infinite rewriting derivations. To some extent, our method has similarities with [18], where an automaton is built for normalization according to a needed-redex strategy in the case of orthogonal rewrite systems.

In Section 2, the background is presented. Section 3 introduces the basic concepts of the inductive proof mechanism. In Section 4, our method is formally described with inference rules and a strategy to apply them. Finally, in Section 5, a strategy is proposed to reach an innermost normal form from a given term, using information of the proof establishing weak termination.

2 Notations

We assume that the reader is familiar with the basic definitions and notations of term rewriting given for instance in [7]. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of terms built from a given finite set $\mathcal{F}$ of function symbols having an arity $n \in \mathbb{N}$, and a set $\mathcal{X}$ of variables denoted $x, y \ldots$. $\mathcal{T}(\mathcal{F})$ is the set of ground terms (without variables). The terms composed of a symbol of arity 0 are called constants; $\mathcal{C}$ is the set of constants of $\mathcal{F}$. Positions in a term are represented as sequences of integers. The empty sequence ϵ denotes the top position. The notation $t|_p$ stands for the subterm of t at position p. The term $u[t_j]_{j \in \{i_1 .. i_k\}}$ denotes the term u in which the subterms $u|_j$ have been replaced by t_j respectively.

A substitution is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = (x \mapsto t) \ldots (y \mapsto u)$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We identify a substitution $\sigma = (x \mapsto t) \ldots (y \mapsto u)$ with the finite conjunction of equations $(x = t) \wedge \ldots \wedge (y = u)$. The result of applying σ to a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is written $\sigma(t)$ or σt. The domain of σ, denoted $Dom(\sigma)$ is the finite subset of $\mathcal{X}$ such that $\sigma x \neq x$. A ground substitution or instantiation is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F})$. The composition of substitutions σ_1 followed by σ_2 is denoted $\sigma_2 \sigma_1$.

Given a set $\mathcal{R}$ of rewrite rules or term rewriting system on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, a function symbol in $\mathcal{F}$ is called a constructor if it does not occur in $\mathcal{R}$ at the top position of the left-hand side of a rule, and is called a defined function symbol otherwise. The set of constructors of $\mathcal{F}$ for $\mathcal{R}$ is denoted by $\mathcal{C}ons_{\mathcal{R}}$, the set of defined function symbols of $\mathcal{F}$ for $\mathcal{R}$ is denoted by $\mathcal{D}ef_{\mathcal{R}}$ ($\mathcal{R}$ is omitted when there is no ambiguity). The rewriting relation induced by $\mathcal{R}$ is denoted by $\rightarrow_{\mathcal{R}}$ ($\rightarrow$ if there is no ambiguity on $\mathcal{R}$). We note $s \rightarrow_{p, l \rightarrow r, \sigma} t$ (or $s \rightarrow^{p, l \rightarrow r, \sigma} t$ where either p or $l \rightarrow r$ or σ may be omitted) if s rewrites into t at position p with the rule $l \rightarrow r$ and the substitution σ, i.e. $s = s[l\sigma]_p$ and $t = s[r\sigma]_p$. The reflexive transitive closure of the rewriting relation induced by $\mathcal{R}$ is denoted by $\rightarrow_R^*$. Given a term t, we call normal form of t, denoted by $t\downarrow$, any irreducible term, if it exists, such that $t \rightarrow_R^* t\downarrow$.

An ordering $\succ$ on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is said to be noetherian iff there is no infinite decreasing derivation (or chain) for this ordering. It is $\mathcal{F}$-stable iff for any pair of terms t,t' of $\mathcal{T}(\mathcal{F},\mathcal{X})$, for any context $f(\ldots\ \ldots)$, $t \succ t'$ implies $f(\ldots t \ldots) \succ f(\ldots t' \ldots)$. It has the subterm property iff for any t of $\mathcal{T}(\mathcal{F},\mathcal{X})$, $f(\ldots t \ldots) \succ t$. Notice that, for $\mathcal{F}$ and $\mathcal{X}$ finite, if $\succ$ is $\mathcal{F}$-stable and has the subterm property, then it is noetherian [6]. If, in addition, $\succ$ is stable by substitution (for any substitution σ, any pair of terms $t,t' \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $t \succ t'$ implies $\sigma t \succ \sigma t'$), then it is called a simplification ordering. Let t be a term of $\mathcal{T}(\mathcal{F})$; like for standard rewriting, we say that t weakly (resp. strongly) (innermost) terminates if and only if at least one (resp. every) (innermost) rewriting derivation starting from t is finite. Obviously, strong (innermost) termination implies weak (innermost) termination. An innermost rewriting normal form of t is also denoted by $t{\downarrow}$, when there is no ambiguity.

3 Induction and Constraints

For proving that the terms t of $\mathcal{T}(\mathcal{F})$ weakly innermost terminate, we proceed by induction on $\mathcal{T}(\mathcal{F})$ with a noetherian ordering $\succ$, assuming that for any t' such that $t \succ t'$, t' weakly innermost terminates. To warrant non emptiness of $\mathcal{T}(\mathcal{F})$, we assume that $\mathcal{F}$ contains at least a constructor constant.

The main intuition is to observe the rewriting derivation tree starting from any ground term $t \in \mathcal{T}(\mathcal{F})$ which is any instance of a term $g(x_1,\ldots,x_m) \in \mathcal{T}(\mathcal{F},\mathcal{X})$, for some defined function symbol $g \in \mathcal{D}ef$, and variables $x_1,\ldots,x_m$. Proving weak innermost termination on ground terms amounts to prove that all these rewriting derivation trees have at least one finite branch.

Each rewriting derivation tree is simulated, using a lifting mechanism, by a proof tree developed from $g(x_1,\ldots,x_m)$ on $\mathcal{T}(\mathcal{F},\mathcal{X})$, for every $g \in \mathcal{D}ef$, by alternatively using two main concepts: narrowing and abstraction. More precisely, narrowing schematizes all innermost rewriting possibilities of terms. The abstraction process simulates the innermost normalization of subterms in the derivations. It consists in replacing these subterms by special variables, denoting one of their possible innermost normal forms, without computing them. This abstraction step is performed on subterms that can be assumed weakly innermost terminating by induction hypothesis.

The schematization of ground rewriting derivation trees is achieved through constraints. The nodes of the developed proof trees are composed of a current term of $\mathcal{T}(\mathcal{F},\mathcal{X})$, and a set of ground substitutions represented by a constraint progressively built along the successive abstraction and narrowing steps. Each node in an abstract tree schematizes a set of ground terms: all ground instances of the current term, that are solutions of the constraint.

The constraint is in fact composed of two kinds of formulas: ordering constraints, set to warrant the validity of the inductive steps, and abstraction constraints combined to narrowing substitutions, which effectively define the relevant sets of ground terms. The latter actually allow controlling the narrowing process, well known to easily diverge.

Unlike [9, 8, 10], where, for proving strong termination, all branches of the proof trees have to be considered, we only develop here the relevant branches that warrant termination of one rewriting derivation for any ground term.

We now introduce the necessary concepts to formalize and automate the technique sketched above.

3.1 Ordering Constraints and Abstraction

The induction ordering $\succ$ is constrained along the proof by imposing constraints between terms that must be comparable, each time the induction hypothesis is used in the abstraction mechanism. As we are working with a lifting mechanism on the proof trees with terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, we directly work with an ordering $\succ_\mathcal{P}$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $t \succ_\mathcal{P} u$ induces $\theta t \succ \theta u$, for every θ solution of the constraint associated to u.

So inequalities of the form $t > u_1, \ldots, u_m$ are accumulated, which are called *ordering constraints*. Any ordering $\succ_\mathcal{P}$ on $\mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying them and which is stable by substitution fulfills the previous requirement on ground terms. The ordering $\succ_\mathcal{P}$, defined on $\mathcal{T}(\mathcal{F}, \mathcal{X})$, can then be seen as an extension of the induction ordering $\succ$, defined on $\mathcal{T}(\mathcal{F})$. For convenience, $\succ_\mathcal{P}$ is also written $\succ$.

It is important to remark that, for establishing the inductive termination proof, it is sufficient to decide whether there exists such an ordering.

Definition 1 (ordering constraint). An ordering constraint *is a pair of terms of* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ *noted* $(t > t')$. *It is said to be* satisfiable *if there exists an ordering* $\succ$, *such that for every instantiation* θ *whose domain contains* $\mathcal{V}ar(t) \cup \mathcal{V}ar(t')$, *we have* $\theta t \succ \theta t'$. *We say that* $\succ$ *satisfies* $(t > t')$.

A conjunction C *of ordering constraints is satisfiable if there exists an ordering satisfying all conjuncts. The empty conjunction, always satisfied, is denoted by* $\top$.

Satisfiability of a constraint C of this form is undecidable. But a sufficient condition for an ordering $\succ$ to satisfy C is that $\succ$ is stable by substitution and $t \succ t'$ for any constraint $t > t'$ of C.

Other constraints are introduced by the abstraction mechanism. To abstract a term u at positions $i_1, \ldots, i_p$, where the $u|_j$ are supposed to have a normal form $u|_j\!\downarrow$, we replace the $u|_j$ by abstraction variables X_j representing respectively one of their possible innermost normal forms. Let us define these special variables more formally.

Definition 2 (NF-variable). *Let* $\mathcal{N}$ *be a set of new variables disjoint from* $\mathcal{X}$. *Symbols of* $\mathcal{N}$ *are called* NF-variables. *Substitutions and instantiations are extended to* $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ *in the following way. Let* $X \in \mathcal{N}$; *for any substitution* σ *(resp. instantiation* θ*) such that* $X \in Dom(\sigma)$, σX *(resp.* θX*) is in normal form, and then* $\mathcal{V}ar(\sigma X) \subseteq \mathcal{N}$.

Definition 3 (term abstraction). *The term* u *is said to be* abstracted *into the term* u' *(called* abstraction *of* u*) at positions* $\{i_1, \ldots, i_p\}$ *iff* $u' = u[X_j]_{j \in \{i_1, \ldots, i_p\}}$, *where the* $X_j, j \in \{i_1, \ldots, i_p\}$ *are fresh distinct NF-variables.*

Weak termination on $\mathcal{T}(\mathcal{F})$ is proved by reasoning on terms with abstraction variables, i.e. on terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. Ordering constraints are extended to pairs of terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. When subterms t_i are abstracted by X_i, we state constraints on abstraction variables, called *abstraction constraints* to express that their instances can only be normal forms of the corresponding instances of t_i. Initially, they are of the form $t{\downarrow} = X$ where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, and $X \in \mathcal{N}$, but we will see later how they are combined with the substitutions used for the narrowing process.

3.2 Narrowing

After abstraction of the current term t into $t[X_j]_{j\in\{i_1,\ldots,i_p\}}$ we test whether the possible ground instances of $t[X_j]_{j\in\{i_1,\ldots,i_p\}}$ are reducible, according to the possible values of the instances of the X_j. This is achieved by innermost narrowing $t[X_j]_{j\in\{i_1,\ldots,i_p\}}$.

To schematize innermost rewriting on ground terms, we need to refine the usual notion of narrowing. In fact, with the usual innermost narrowing relation, if a position p in a term t is a narrowing position, no suffix position of p can be a narrowing position too. However, if we consider ground instances of t, we can have rewriting positions p for some instances, and p' for some other instances, such that p' is a suffix of p. So, when using the narrowing relation to schematize innermost rewriting of ground instances of t, the narrowing positions p to consider depend on a set of ground instances of t, which is defined by excluding the ground instances of t that would be narrowable at some suffix position of p. For instance, with the TRS $R = \{g(a) \to a, f(g(x)) \to b\}$, the innermost narrowing positions of the term $f(g(X))$ are 1 with the narrowing substitution $\sigma = (X = a)$, and ϵ with any σ such that $\sigma X \neq a$.

Let σ be a substitution on $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. In the following, we identify σ with the equality formula $\bigwedge_i (x_i = t_i)$, with $x_i \in \mathcal{X} \cup \mathcal{N}$, $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$. Similarly, we call *negation* $\overline{\sigma}$ of the substitution σ the formula $\bigvee_i (x_i \neq t_i)$.

Definition 4. *A substitution σ is said* to satisfy *a constraint $\bigwedge_j \bigvee_{i_j} (x_{i_j} \neq t_{i_j})$, iff for all ground instantiation θ, $\bigwedge_j \bigvee_{i_j} (\theta\sigma x_{i_j} \neq \theta\sigma t_{i_j})$. A* constrained substitution *σ is a formula $\sigma_0 \wedge \bigwedge_j \bigvee_{i_j} (x_{i_j} \neq t_{i_j})$, where σ_0 is a substitution, and $\bigwedge_j \bigvee_{i_j} (x_{i_j} \neq t_{i_j})$ the constraint to be satisfied by σ_0.*

Definition 5 (innermost narrowing). *A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$* innermost narrows *into a term $t' \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$ at the non-variable position p, using the rule $l \to r \in \mathcal{R}$ with the constrained substitution $\sigma = \sigma_0 \wedge \bigwedge_{j\in[1..k]} \overline{\sigma_j}$, which is written $t \rightsquigarrow^{Inn}_{p,l\to r,\sigma} t'$ iff*

$$\sigma_0(l) = \sigma_0(t|_p) \text{ and } t' = \sigma_0(t[r]_p)$$

where σ_0 is the most general unifier of t and l at position p, and $\sigma_j, j \in [1..k]$ are all most general unifiers of $\sigma_0 t$ and a left-hand side of rule of $\mathcal{R}$, at suffix positions of p.

Notice that we are interested in the narrowing substitution applied to the current term t, but not in its definition on the variables of the left-hand side of the rule. So the narrowing substitutions we consider are restricted to the variables of the narrowed term t.

3.3 Cumulating Constraints

Abstraction constraints have to be combined with the narrowing constrained substitutions to characterize the ground terms schematized by the proof trees. A narrowing step is applied to a current term u if the narrowing substitution σ effectively corresponds to a rewriting step of ground instances of u, i.e. if σ is *compatible* with the abstraction constrained formula A associated to u (i.e. σA is satisfiable). Else, the narrowing step is useless. So the narrowing constraint attached to the narrowing step is added to the abstraction constraints initially of the form $t{\downarrow} = X$. This motivates the introduction of abstraction constrained formulas.

Definition 6. *An* abstraction constrained formula *(ACF in short) is a formula* $\bigwedge_i(t_i{\downarrow} = t'_i) \wedge \bigwedge_j \bigvee_{k_j}(u_{k_j} \neq v_{k_j})$*, where* $t_i, t'_i, u_{k_j}, v_{k_j} \in \mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$.

Definition 7. *An* abstraction constrained formula $A = \bigwedge_i(t_i{\downarrow} = t'_i) \wedge \bigwedge_j \bigvee_{k_j}(u_{k_j} \neq v_{k_j})$ *is* satisfiable *iff there exists at least one instantiation* θ *such that* $\bigwedge_i(\theta t_i{\downarrow} = \theta t'_i) \wedge \bigwedge_j \bigvee_{k_j}(\theta u_{k_j} \neq \theta v_{k_j})$*. The instantiation* θ *is then said to satisfy the ACF A and is called solution of A.*

Applying a constrained substitution $\sigma = \sigma_0 \wedge \bigwedge_j \bigvee_{i_j}(x_{i_j} \neq t_{i_j})$ to an ACF A gives a formula σA obtained by applying σ_0 to A and then by adjoining the disequality part to the result.

An ACF A is attached to each term u in the proof trees; its solutions characterize the interesting ground instances of this term, that are the θu such that θ is a solution of A. When A has no solution, the current node of the proof tree does not represent any ground term. Such nodes are then irrelevant for the weak termination proof. So we have the choice between generating only the relevant nodes of the proof tree, by testing satisfiability of A at each step, or stopping the proof on a branch on an irrelevant node, by testing unsatisfiability of A. These are both facets of the same question, but in practice, they lead to different solutions.

Checking satisfiability of A is in general undecidable. The disequality part of an ACF is a particular instance of a disunification problem (a quantifier free equational formula, qfef in short), whose satisfiability has been addressed in [5], that provides rules to transform any disunification problem into a solved form. Testing satisfiability of the equational part of an ACF is undecidable in general, but sufficient conditions can be given, relying on a characterization of normal forms.

Unsatisfiability of A is also undecidable in general, but simple sufficient conditions can be used, very often applicable in practice. They rely on reducibility, unifiability, narrowing and constructor tests, and can be found in [11].

So both satisfiability and unsatisfiability checks need to use sufficient conditions. But in the first case, the proof process stops with failure as soon as satisfiability of A cannot be proved. In the second one, it can go on, until A is proved to be unsatisfiable, or until other stopping conditions are fulfilled. In the approach followed below, narrowing and abstraction are applied without checking the satisfiability of abstraction constraints, and the process stops as soon as they are detected to be unsatisfiable.

4 Inference Rules for Inductive Termination Proofs

We are now ready to describe the different steps of our mechanism on a term t, with initial empty constraints conjunctions A, C. It consists in iterating the three following steps.

The first step abstracts the current term u at given positions $i_1, \ldots, i_p$. The constraints $t > u|_{i_1}, \ldots, u|_{i_p}$ are set, allowing to suppose, by induction, the existence of irreducible forms for $u|_{i_1}, \ldots, u|_{i_p}$. Then, $u|_{i_1}, \ldots, u|_{i_p}$ are abstracted into abstraction variables $X_{i_1}, \ldots, X_{i_p}$ (or $X_1, \ldots, X_p$ for simplifying the indices). The abstraction constraint $u|_{i_1}\downarrow = X_1, \ldots, u|_{i_p}\downarrow = X_p$ is added to the ACF A. This is the *abstract* step. The abstraction positions are chosen so that the abstraction mechanism captures the greatest possible number of rewriting steps: we abstract the greatest possible subterms of $u = f(u_1, \ldots, u_m)$. Note also that it is not useful to abstract non narrowable subterms: their ground instances are always in normal form, since the variables of these subterms are NF-variables.

The second step innermost narrows the resulting term in one step with all possible rewrite rules of the rewrite system $\mathcal{R}$, and all possible substitutions σ, into terms v, according to Definition 5. This step is a branching step, creating as many states as narrowing possibilities. The substitution σ is integrated to A, as explained after Definition 7. This is the *narrow* step.

We then have a *stop* step halting the proof process on the current branch of the proof tree, when A is detected to be unsatisfiable, or when the ground instances of the current term can be stated weakly innermost terminating, which happens when the induction hypothesis applies on it.

The previously presented steps are performed by inference rules that transform 3-tuples (T, A, C) where T is a set of terms of $\mathcal{T}(\mathcal{F}, \mathcal{X} \cup \mathcal{N})$, containing the current term whose weak innermost termination has to be proved: this is either a singleton or the empty set, A is an ACF and C is a conjunction of ordering constraints stated by the abstract steps.

Before giving the corresponding inference rules, let us notice that the inductive reasoning can be completed in the following way. When the induction hypothesis cannot be applied on a term u, it is sometimes possible to prove weak innermost termination of every ground instance of u by another way. Let $WT(u)$ be a predicate that is true iff every ground instance of u weakly innermost terminates. In the first (resp. third) previous step of the induction reasoning, we then associate the alternative predicate $WT(u|_{i_j})$ (resp. $WT(u)$) to the condition

Table 1. Inference rules for the weak innermost termination proof

Abstract: $$\frac{\{u\},\ A,\ C}{\{u'\},\ A \wedge u|_{i_1}\downarrow = X_{i_1} \ldots \wedge u|_{i_p}\downarrow = X_{i_p},\ C \wedge H_C(u|_{i_1}) \ldots \wedge H_C(u|_{i_p})}$$

where u is abstracted into u' at the positions $i_1, \ldots, i_p \neq \epsilon$
if $C \wedge H_C(u|_{i_1}) \ldots \wedge H_C(u|_{i_p})$ is satisfiable
where $H_C(u|_j)_{j \in \{i_1,\ldots,i_p\}} = \begin{cases} true & \text{if } WT(u|_j) \\ t_{ref} > u|_j & \text{otherwise.} \end{cases}$

Narrow: $$\frac{\{u\},\ A,\ C}{\{v\},\ \sigma A,\ C} \text{ if } u \rightsquigarrow_{\sigma}^{Inn} v$$

Stop: $$\frac{\{u\},\ A,\ C}{\emptyset,\ A,\ C \wedge H_C(u)}$$

if $(C \wedge H_C(u))$ is satisfiable or **A is unsatisfiable**
where $H_C(u) = \begin{cases} true & \text{if } WT(u) \text{ or } \textbf{A is unsatisfiable} \\ t_{ref} > u & \text{otherwise.} \end{cases}$

$t > u|_{i_j}$ (resp. $t > u$). For establishing that $WT(u)$ is true, in some cases, the notion of usable rules [1] can be used. This approach is fully developed in [13].

The termination proof procedure is described by the set of rules given in Table 1. These rules must be applied on the initial pairs $(\{t_{ref} = g(x_1, \ldots, x_m)\}, \top, \top)$, where g is a defined symbol, with the strategy S

$$(\textbf{Abstract}; \text{dk}(\textbf{Narrow}); \textbf{Stop})\ *$$

where ";" denotes the sequential application of rules, "dk" the application of a rule in all possible ways and "*" the iterative application of a strategy, until it is not possible anymore. The process stops if no inference rule applies anymore.

There are two cases for the behavior of the termination proof procedure. The strategy applied to the initial state $(\{t_{ref}\}, \top, \top)$ terminates if the rules do not apply anymore and all states are of the form $(\emptyset, A, C)$. Otherwise, the strategy does not terminate if there is an infinite number of applications of **Abstract** and **Narrow**.

A branch of the derivation tree is said to be successful if it is ended by an application of **Stop**, i.e. if its final state is of the form $(\emptyset, A, C)$.

Thus, the inductive weak termination proof is successful if there is at least one successful branch corresponding to each possible ground term. Let us develop this point.

In fact, branching, produced by **Narrow**, can generate different states with narrowing substitutions $\sigma_1, \ldots \sigma_n$. These substitutions can be compared (see [11]). For σ_i and σ_j, three situations may occur: σ_i is strictly less general than σ_j, which is noted $\sigma_i > \sigma_j$, (or σ_j is strictly less general than σ_i), σ_i and σ_j are equal up to a renaming, or else σ_i and σ_j are incomparable.

States corresponding to substitutions that are more general than other ones then represent a set of ground instances that contains the other ones. So, for proving weak termination for all ground instances at a branching point, it is sufficient to prove weak termination only for the "most general states".

Note that the ignored states may schematize different rewriting steps than those we consider (at different positions, with different rewrite rules). So for the considered instances, if a "most general state" doesn't exclusively give rise to successful branches, we lose the possibility to test whether the other branches are successful. In practice, this case rarely occurs and the gain is greater in avoiding to consider redundant subsets of instances.

A branching node in a proof tree is a state, on which the Narrow rule applies. Let Σ be the set of narrowing substitutions (possibly with different rewrite rules) at a given branching node. Let Σ_0 be the reduced set from Σ such that $\sigma \in \Sigma_0$ iff $\sigma \in \Sigma$ and $\nexists\ \sigma' \in \Sigma$ such that $\sigma > \sigma'$ on $(Dom(\sigma) \setminus Var(l)) \cup (Dom(\sigma') \setminus Var(l'))$, where l and l' are the left-hand sides of rules respectively used to produce the narrowing substitutions σ and σ'. The set Σ_0 may yet contain equivalent (equal up to a renaming) substitutions which are marked as such. So for any two substitutions in Σ_0, either they are equivalent, or they are incomparable.

A proof tree is *weakly successful* if it is reduced to a state of the form $(\emptyset, A, C)$, or if at each branching node:

- for each class of equivalent substitutions, there exists at least one weakly successful subtree corresponding to a substitution in this class,
- all subtrees corresponding to incomparable substitutions are weakly successful.

So the strategy S can be optimized as follows: at each branching point of a proof tree, with set of substitutions Σ, we only develop the subtrees corresponding to Σ_0. Moreover, given two subtrees corresponding to equivalent substitutions, as soon as one of them is weakly successful, the other one is cut.

We write $SUCCESS(g, \succ)$ if the proof tree obtained by application on $(\{g(x_1, \ldots, x_m)\}, \top, \top)$, with the strategy S, of the inference rules whose conditions are satisfied by an ordering $\succ$, is weakly successful.

Theorem 1. *Let $\mathcal{R}$ be a TRS on a set $\mathcal{F}$ of symbols. If there exists an $\mathcal{F}$-stable ordering $\succ$ having the subterm property, such that for each defined symbol g, we have $SUCCESS(g, \succ)$, then every term of $\mathcal{T}(\mathcal{F})$ weakly innermost terminates.*

A formal description with a complete set of inference rules for describing the subtree cut process, and proofs of theorems are given in [11].

5 Finding a Good Derivation Chain

As said previously, establishing weak termination of an undeterministic evaluation process warrants a result if a breadth-first strategy is adopted for this process. But such a strategy is in general very costly, and it is much better to

have hints about the terminating derivations to compute them directly with a depth-first mechanism.

Our proof process, as it simulates the rewriting mechanism, gives complete information on a terminating rewriting branch. It allows extracting the exact application of rewrite rules that yields a normal form. To rewrite a term, it is enough to follow the rewriting scheme simulated by abstraction and narrowing in the proof trees.

We now formalize the use of the proof trees to compute a normal form for any term.

Definition 8. *Let $\mathcal{R}$ be a TRS proved weakly terminating with Theorem 1. The strategy tree ST_f associated to $f \in \mathcal{D}ef_{\mathcal{R}}$ is the proof tree obtained from the initial state $(\{f(x_1, \ldots, x_m)\}, \top, \top)$.*

Definition 9. *Let $\mathcal{R}$ be a TRS proved weakly terminating with Theorem 1. Let $ST = \{ST_f | f \in \mathcal{D}ef_{\mathcal{R}}\}$ be the set of strategy trees of $\mathcal{R}$ and $s = f(s_1, \ldots, s_m) \in \mathcal{T}(\mathcal{F})$. Normalizing s with respect to ST into $norm_{ST}(s)$ is defined in the following way:*

- *if $f \in \mathcal{C}ons_{\mathcal{R}}$, then $norm_{ST}(f(s_1, \ldots, s_n)) = f(norm_{ST}(s_1), \ldots, norm_{ST}(s_n))$,*
- *if $f \in \mathcal{D}ef_{\mathcal{R}}$, then normalizing s with respect to ST into $norm_{ST}(s)$ is performed by following the steps in the strategy tree ST_f of f, where $t = g(t_1, \ldots, t_n)$ is any term of the transformation chain of s with respect to ST and $u = g(u_1, \ldots, u_n)$ is the corresponding term in ST_f:*
  - *if the step is* **Abstract**, *and abstracts u at positions $i_1, \ldots, i_p$, then $t \mapsto t[t'_1]_{i_1} \ldots [t'_p]_{i_p}$,*
    where $t'_j = \begin{cases} t|_{i_j}\downarrow & \text{if } WT(u|_{i_j}) \\ norm_{ST}(t|_{i_j}) & \text{otherwise,} \end{cases}$
  - *if the step is* **Narrow** *with $g(u_1, \ldots, u_n) \leadsto^{Inn}_{p,l\to r,\sigma} u'$, then $g(t_1, \ldots, t_n) \mapsto t'$ where t' is defined by $g(t_1, \ldots, t_n) \to^{Inn}_{p,l\to r,\mu} t' = \mu u'$, with $\theta = \mu\sigma$ on $Var(g(u_1, \ldots, u_n))$ and $g(t_1, \ldots, t_n) = \theta g(u_1, \ldots, u_n)$ if μ exists,*
    $t' = g(t_1, \ldots, t_n)$ and the normalizing process stops, otherwise,
  - *if the step is* **Stop**, *then $g(t_1, \ldots, t_n) \mapsto t'$,*
    where $t' = \begin{cases} g(t_1, \ldots, t_n)\downarrow & \text{if } WT(g(u_1, \ldots, u_n)) \\ norm_{ST}(g(t_1, \ldots, t_n)) & \text{otherwise.} \end{cases}$

Given a TRS $\mathcal{R}$, the previous definition assumes that if the predicate WT has been used to prove termination of a particular term t during the termination proof of $\mathcal{R}$, one is able to find a normalizing strategy for t. A simple sufficient condition is that t is proved strongly terminating, which can be established in most cases, like for WT, with the usable rules. Under this assumption, the following theorem holds.

Theorem 2. *Let $\mathcal{R}$ be a TRS proved weakly terminating with Theorem 1 and ST its set of strategy trees. Then for any term $t \in \mathcal{T}(\mathcal{F})$, $norm_{ST}(t)$ is an innermost normal form of t for $\mathcal{R}$.*

Let us come back to the TRS $\mathcal{R}$ presented in the introduction, built on $\mathcal{F} = \{f : 2, h : 2, g : 1, s : 1, 0 : 0\}$. We first prove that every ground term t of $\mathcal{T}(\mathcal{F})$ can be innermost normalized with $\mathcal{R}$, and then infer from this proof a strategy allowing normalization of any ground term of $\mathcal{T}(\mathcal{F})$.

Since the defined symbols of $\mathcal{R}$ are f, g, and h, we have to apply the inference rules to $f(x_1, x_2)$, $g(x_1)$ and $h(x_1, x_2)$. The proof trees, given in Table 2, show how the inference rules are applied, and provide the information needed to infer a strategy for normalizing any ground term. When **Narrow** applies, we specify the narrowing substitution, when it is useful for normalization, and in parentheses, the rewrite rule number used to narrow.

The subtree marked by $\bigodot$ in the proof tree of f is cut as soon as the subtree generated on the left from $\mathbf{f(X_6, s(0))}$ with the same substitution (up to a renaming) $\sigma = (X_6 = g(X_7)) \wedge (X_7 \neq s(X_8) \wedge X_7 \neq 0)$ is successful.

The final proof trees are **bold**. Since they are all successful, $\mathcal{R}$ is proved weakly innermost terminating on the ground term algebra. These proof trees are respectively the strategy trees $\mathbf{ST_g}, \mathbf{ST_h}$ and $\mathbf{ST_f}$, from which we can now infer a strategy normalizing any ground term t, according to Definition 9.

As an example, let us use the strategy to normalize the term $f(g(f(0,0)), s(0))$ following the steps of ST_f.

(Step 1 in $\mathbf{ST_f}$: Abstract) The first step is **Abstract** at positions 1 and 2 by application of the induction hypothesis, and then we get $f(g(f(0,0)), s(0)) \mapsto f(norm_{ST}(g(f(0,\ 0))), norm_{ST}(s(0))$. Since s is a constructor, we have $norm_{ST}\ (s(0)) = s(norm_{ST}(0))$. Since 0 is a constructor constant, we have $norm_{ST}(0) = 0$, and finally $norm_{ST}(s(0)) = s(0)$. We now have to compute $norm_{ST}(g(f(0,\ 0)))$, by following the steps of ST_g.

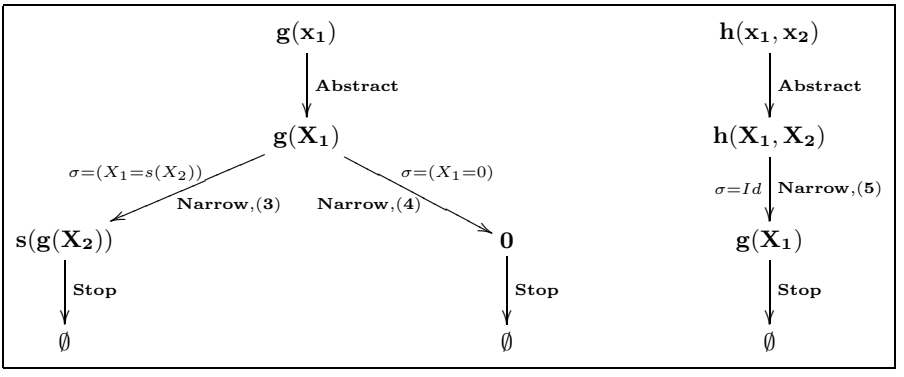
(Step 1 in $\mathbf{ST_g}$: Abstract) The first step is **Abstract** at position 1 by application of the induction hypothesis, and then we get $g(f(0,0)) \mapsto g(norm_{ST}(f(0,0)))$. To compute $norm_{ST}(f(0,0))$, we have to follow the steps of ST_f.

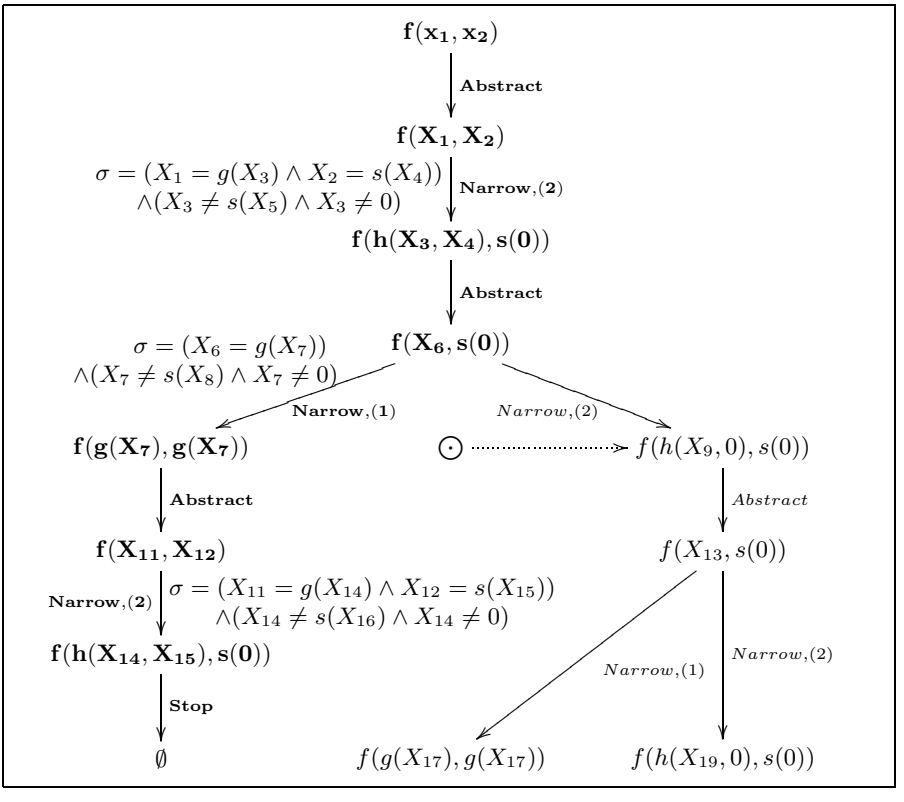
(Step 1 in $\mathbf{ST_f}$: Abstract) The first step is **Abstract** at positions 1 and 2 by application of the induction hypothesis, and then we get $f(0,0) \mapsto f(norm_{ST}(0),\ norm_{ST}(0))$. Since 0 is a constant constructor, we have $norm_{ST}(0) = 0$, and then $f(0,0) \mapsto f(0,0)$.

(Step 2 in $\mathbf{ST_f}$: Narrow) The second step is **Narrow** at the top position, with rule (2). The narrowing substitution σ is such that our current term $f(0,0)$ is not a ground instance of $\sigma f(X_1, X_2)$. Therefore $f(0,0) \mapsto f(0,0)$, and finally $norm_{ST}(f(0,0)) = f(0,0)$. We then come back to normalization of $g(f(0,0))$.

(Step 2 in $\mathbf{ST_g}$: Narrow) Our current term is $g(f(0,0))$, and the second step of ST_g is **Narrow** at the top position, with rules (3) and (4). None of the narrowing substitutions σ is such that our current term $g(f(0,0))$ is a ground instance of $\sigma g(X_1)$. Therefore $g(f(0,0)) \mapsto g(f(0,0))$, and finally $norm_{ST}(g(f(0,0))) = g(f(0,0))$. We then come back to normalization of our main term.

Table 2. Proof trees for symbols g, h and f





(Step 2 in $\mathbf{ST_f}$: Narrow) Our current term is $f(g(f(0,0)),s(0))$, and the current step in $\mathbf{ST_f}$ is **Narrow** at the top position with rule (2). The narrowing substitution σ is such that our current term is a ground instance of $\sigma f(X_1, X_2)$. So $f(g(f(0,0)),s(0)) \rightarrow^{\epsilon,(2)} f(h(f(0,0),0),s(0))$.
(Step 3 in $\mathbf{ST_f}$: Abstract) The current step in the proof tree is **Abstract** at position 1 thanks to the WT predicate, and more precisely thanks to

the usable rules which give a strong terminating system. Then we have $h(f(0,0),0) \mapsto h(f(0,0),0)\downarrow$, and it suffices to rewrite $h(f(0,0),0)$ as long as a normal form is reached, which is guaranteed by the termination of the usable rules. Here we have $h(f(0,0),0) \rightarrow^{\epsilon,(5)} g(f(0,0))$. Finally we get $f(h(f(0,0),0),s(0)) \mapsto f(g(f(0,0)),s(0))$.

(Step 4 in $\mathbf{ST_f}$: Narrow) The current step in the tree is **Narrow** at the top position with rule (1). The narrowing substitution σ is such that our current term is a ground instance of $\sigma f(X_6, s(0))$. So $f(g(f(0,0)),s(0)) \rightarrow^{\epsilon,(1)} f(g(f(0,0)),g(f(0,0)))$.

(Step 5 in $\mathbf{ST_f}$: Abstract) The current step in the tree is **Abstract** at positions 1 and 2 thanks to the WT predicate, and then $f(g(f(0,0)),g(f(0,0))) \mapsto f(g(f(0,0))\downarrow,g(f(0,0))\downarrow)$. Since $g(f(0,0))$ is in normal form, we get $f(g(f(0,0)),g(f(0,0))) \mapsto f(g(f(0,0)),g(f(0,0)))$.

(Step 6 in $\mathbf{ST_f}$: Narrow) The current step of ST_f is **Narrow** at the top position, with rule (2). The narrowing substitution σ is such that our current term is a not a ground instance of $\sigma f(X_{11}, X_{12})$. Therefore the normalizing process stops on $f(g(f(0,0)),g(f(0,0)))$, which hence is a normal form of $f(g(f(0,0)),s(0))$.

For a more detailed development of this example, as well as for other examples, see [11].

6 Conclusion and Perspectives

In this paper, we have proposed a method to prove weak innermost termination of term rewriting systems by explicit induction on the termination property. To simulate the innermost rewriting derivations of any ground term, we generate proof trees issued from patterns $g(x_1,\ldots,x_m)$ where g is a defined function symbol, in using two mechanisms: abstraction, introducing variables that represent ground normal forms, and narrowing, schematizing rewriting on ground terms.

When all proof trees have a successful branch for all ground instances of the patterns, the weak innermost termination property of the rewrite system is proved. Then from these successful branches, a normalizing strategy can be inferred for any ground term. We show how to extract the relevant information from the proof trees to guide the innermost normalization process.

Proving weak termination of a program and deducing a normalizing strategy can be achieved at *compile-time*. Then, to evaluate a data at *run-time* with no risk of non-termination, it suffices to follow the strategy described in Section 5, that states which rule to apply and at which position in the term, at each step of the normalization process. Henceforth, evaluation at run-time is made very efficient, since it always leads to a result, i.e. an irreducible term.

Up to our knowledge, this is the first method proposed to ensure weak termination of rewriting systems, allowing to find a finite evaluation for every term.

The important point to automate our proof principle is the satisfaction of the constraints at each step of the proof. On many examples, this is immediate:

as the ordering constraints only express the subterm property, they are trivially satisfied by any simplification ordering. Otherwise, we can use automatic ordering constraint solvers. As for abstraction constraints, they can be managed with an unsatisfiability test, for which simple sufficient conditions exist, that are automated. Thus, in general, weak termination proof can be completely automatic.

As in our approach, the rewriting strategy is explicitly handled in the proof principle, the method should be easily applicable to other strategies, especially to the outermost strategy, and to local strategies on operators. This potentially leads to a new functionality for CARIBOO, a toolbox for proving termination under strategies [9].

References

1. T. Arts and J. Giesl. Proving innermost normalization automatically. Technical Report 96/39, Technische Hochschule Darmstadt, Germany, 1996.
2. T. Arts and J. Giesl. Proving innermost normalisation automatically. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer-Verlag, 1997.
3. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume15.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science. Report LORIA 98-R-316.
4. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
5. H. Comon. Disunification: a survey. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.
6. N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.
7. Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.
8. O. Fissore, I. Gnaedig, and H. Kirchner. Termination of rewriting with local strategies. In M. P. Bonacina and B. Gramlich, editors, *Selected papers of the 4th International Workshop on Strategies in Automated Deduction*, volume 58 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 2001.
9. O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO : An induction based proof tool for termination with strategies. In *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 62–73, Pittsburgh (USA), October 2002. ACM Press.

10. O. Fissore, I. Gnaedig, and H. Kirchner. Outermost ground termination. In *Proceedings of the Fourth International Workshop on Rewriting Logic and Its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, Pisa, Italy, September 2002. Elsevier Science Publishers B. V. (North-Holland).
11. O. Fissore, I. Gnaedig, and H Kirchner. Proving weak termination also provides the right way to terminate - Extended version. Technical report, LORIA, Nancy (France), March 2004. Available at http://www.loria.fr/∼gnaedig/PAPERS/REPORTS/wt-extended-2004.ps.
12. K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment – an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proceedings of the 1st IEEE Int. Conference on Formal Engineering Methods*, 1997.
13. I. Gnaedig, H. Kirchner, and O. Fissore. Induction for innermost and outermost ground termination. Technical Report A01-R-178, LORIA, Nancy (France), September 2001.
14. Goubault-Larreck. Well-founded recursive relations. In *Proc. 15th Int. Workshop Computer Science Logic (CSL'2001)*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001. Springer-Verlag.
15. J. Goubault-Larrecq. A proof of weak termination of typed lambda-sigma-calculi. In *Proceedings of the TYPES'96 Workshop*, volume 1512 of *Lecture Notes in Computer Science*, Aussois (France), 1998. Springer-Verlag.
16. Bernhard Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. In Andrei Voronkov, editor, *Proceedings of the 3rd International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *Lecture Notes in Computer Science*, pages 285–296, St. Petersburg, Russia, July 1992. Springer-Verlag.
17. Bernhard Gramlich. On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems. *Theoretical Computer Science*, 165(1):97–131, September 1996.
18. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*, chapter 11, pages 395–414. The MIT press, 1991.
19. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
20. S. Lucas. Termination of rewriting with strategy annotations. In A. Voronkov and R. Nieuwenhuis, editors, *Proc. of 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'01*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 669–684, La Habana, Cuba, December 2001. Springer-Verlag, Berlin.

Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn

Silvio Ranise, Christophe Ringeissen, and Duc-Khanh Tran

LORIA — INRIA, 615, rue du Jardin Botanique,
BP 101, 54602 Villers-lès-Nancy Cedex France
{ranise, ringeiss, tran}@loria.fr

Abstract. We consider the problem of building satisfiability procedures for unions of disjoint theories. We briefly review the combination schemas proposed by Nelson-Oppen, Shostak, and others. Three inference systems are directly derived from the properties satisfied by the theories being combined and known results from the literature are obtained in a uniform and abstract way. This rational reconstruction is the starting point for further investigations. We introduce the concept of extended canonizer and derive a modularity result for a new class of theories (larger than Shostak and smaller than Nelson-Oppen theories) which is closed under disjoint union. This is in contrast with the lack of modularity of Shostak theories. We also explain how to implement extended canonizers by using the basic building blocks used in Shostak schema or by means of rewriting techniques.

1 Introduction

There is an obvious need of using decision procedures in deduction systems and constraint programming environments since their use allows us to reason on a specific computation domain (or a class of computation domains), to improve efficiency and reduce user-interaction. In almost all applications, the computation domain is an amalgamation of domains or a union (combination) of theories whose domains are axiomatized by formulae. For example, program verification usually assumes a union of theories axiomatizing classical data-structures such as lists, arrays, and arithmetics. To tackle this kind of problems, an appealing approach is to proceed in a modular way, by combining decision procedures available for component theories. This line of research was started in the early 80's by two combination schemas independently presented by Nelson-Oppen [19] and Shostak [24] for unions of theories with disjoint signatures. Each schema makes different assumptions on the properties the theories to be combined should satisfy. The former requires the theories to have a satisfiability procedures and to be such that a satisfiable formula in a component theory T is also satisfiable in an infinite model of T (*stable-infiniteness*). The latter assumes the theories admit procedures for reducing terms to canonical form (*canonizers*) and algorithms for solving equations (*solvers*). A *NO* theory admits a satisfiability procedure and is stably-infinite while a canonizer and a solver are defined for a *SH* theory.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 372–386, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Recently, a series of papers [5, 22, 3, 14, 13, 17, 23, 4, 18] have clarified the subtle issues of combining *SH* theories by studying their relationships with *NO* theories. Unfortunately, these papers lack uniformity and non-experts may be confused. For example, some works [5, 22, 3, 23] use pseudo-code to describe the combination algorithms while others [13, 17, 4, 18] adopt a more abstract (rule-based) presentation. There are advantages (and disadvantages) in both approaches: the pseudo-code offers a better starting point for implementation while inference systems make correctness proofs easier. The **first contribution** of this paper is to provide a synthesis of Nelson-Oppen and Shostak approaches to disjoint combination by using a rule-based approach in which many recent results are recast and proved correct in a uniform, rigorous, and simple way.

Our rational reconstruction proceeds as follows. First, we recall that *SH* theories are contained in the class of (convex) *NO* theories (Section 2.1). According to this abstract classification, three possible scenarios are to be considered when combining two theories: (a) both are *NO* theories (Section 3.1), (b) both are *SH* theories (Section 3.2), and (c) one is a *SH* and the other is a *NO* theory (Section 3.3). We formalize the combination schema for each scenario as an inference system. The applicability conditions of the inference rules are derived from the properties of the theories being combined. Along the lines of [13, 18, 4], the combination schema for (b) is obtained as a refinement of that for (a). The inference system formalizing the combination schema for (c), already considered in [3], is obtained by modularly reusing those for (a) and (b) in a natural and straightforward way. As a final remark, we mention the possibility of refining the abstract inference systems presented here with strategies as done in [4], so to get a more fine-grained rule-based implementation which mimics a Shostak procedure as described in [23]. We do not do this here, since we are interested in modularity rather than efficiency.

Our synthesis of combination schemas serves two purposes. First, although the results are not new, we believe that presenting them in a uniform framework could provide a valuable reference for people interested in combination problems, especially for non-experts of the field. Second, it can serve as the starting point for further investigations. As an example, a problem of greatest importance when combining *SH* theories is the lack of modularity for solvers [17]: no general method exists to produce a solver for the union of *SH* theories from the solvers of the component theories. This lack of modularity together with the observation that the theory of equality (ubiquitous in virtually any application where combinations of decision procedures are needed) is not a *SH* theory seem to suggest that any *ad hoc* combination schema for scenario (c) constitute a reasonable trade-off between efficiency and generality: solvers and canonizers for *SH* theories efficiently derive new equalities and cooperate in a Nelson-Oppen way. This solution (adopted, for example, in ICS [11]) can be easily specified in the framework proposed in this paper. In fact, the schema of Section 3.1 can be applied to construct a satisfiability procedure for the union of many *NO* theories which can then be used as the component *NO* theory in a simple generalization of the schema in Section 3.3 to accommodate several solvers and canonizers.

However, this solution leaves open the question about the existence of a suitable concept that would allow us to obtain a modularity result and retain some of the efficiency of the canonizers and solvers. By investigating this question in our framework, we propose the concept of *extended canonizer* which constitutes the **second contribution** of our paper. Intuitively, an extended canonizer allows us to canonize terms with respect to a given theory T and a given T-satisfiable set of equations Γ, so that the uniform word problem for T, i.e. $T \models \Gamma \Rightarrow s = t$, reduces to the problem of checking the identity $ecan(\Gamma)(s) = ecan(\Gamma)(t)$, where $ecan(\Gamma)(s)$ and $ecan(\Gamma)(t)$ are the "extended canonical forms" of s and t, respectively (Section 4.1). A similar concept was introduced in [22] for the theory of equality and its combination with one Shostak theory is also described by a rigorous version of Shostak schema. In [23], such a schema is generalized to consider the combination of the theory of equality with an arbitrary number of *SH* theories by an interesting generalization of Shostak schema requiring only the construction of a canonizer for the union of the theories and invoking the solvers for the constituent theories. The main difference with our work is that the concept of extended canonizer introduced in this paper is *modular*, i.e. there exists a procedure that, given two extended canonizers for two component theories, yields an extended canonizer for their union (Section 4.3). Another interesting feature of extended canonizers is that they can be *efficiently built* by reusing a wealth of existing techniques such as canonizers and solvers for *SH* theories and rewriting techniques (as advocated in [15, 2, 1]) for theories which do not admit a solver (Section 4.2). To summarize, the concept of extended canonizer offers an interesting trade-off between modularity and the possibility to reuse disparate techniques to solve the uniform word problem under a common interface. As a final remark, we notice that our definition of extended canonizer is orthogonal to the line of research (advocated in [17]) which suggests that modular solvers may exist in modified settings such as multi-sorted logic.

Structure of the Paper. Section 2 introduces basic concepts of first-order logic and combination of theories. Section 3 presents a rational reconstruction of combination methods. Section 4 defines the concept of extended canonizer and shows how it can be built out of canonizers and solvers, or rewriting based procedures; the modularity of extended canonizers is also studied. Section 5 presents some conclusions and discusses the future work. All proofs omitted in this version of the paper can be found in [21].

2 Preliminaries

We assume the usual first-order syntactic notions of signature, term, position, and substitution, as defined, e.g., in [7].

Let Σ be a first-order signature containing only function symbols with their arity and $\mathcal{X}$ a set of variables. A 0-ary function symbol is called a *constant*. A *Σ-term* is a first-order term built out of the symbols in Σ and the variables in $\mathcal{X}$. We use the standard notion of substitution. We write substitution applications in postfix notation, e.g. $t\sigma$ for a term t and a substitution σ. The set of variables

occurring in a term t is denoted by $Var(t)$. If l and r are two Σ-terms, then $l = r$ is a Σ-*equality* and $\neg(l = r)$ (also written as $l \neq r$) is a Σ-*disequality*. A Σ-literal is either a Σ-equality or a Σ-disequality. A Σ-*formula* is built in the usual way out of the universal and existential quantifiers, Boolean connectives, and symbols in Σ. If φ is a formula, then $Var(\varphi)$ denotes the set of free variables in φ. We call a formula *ground* if it has no variable, and a *sentence* if it has no free variables. Substitution applications are extended to arbitrary first-order formulas, and are written in postfix notation, e.g. $\varphi\sigma$ for a formula φ and a substitution σ.

We also assume the usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, as given, e.g., in [10]. A *first-order theory* is a set of first-order sentences. A Σ-*theory* is a theory all of whose sentences have signature Σ. All the theories we consider are first-order theories *with equality*, which means that the equality symbol $=$ is always interpreted as the identity relation. The theory of equality is denoted by $\mathcal{E}$. A Σ-structure $\mathcal{A}$ is a model of a Σ-theory T if $\mathcal{A}$ satisfies every sentence in T. A Σ-formula is *satisfiable in* T if it is satisfiable in a model of T. Two Σ-formulas φ and ψ are *equisatisfiable in* T if for every model $\mathcal{A}$ of T, φ is satisfiable in $\mathcal{A}$ iff ψ is satisfiable in $\mathcal{A}$. The *satisfiability problem* for a theory T amounts to establishing whether any given finite quantifier-free conjunction of literals (or equivalently, any given finite set of literals) is T-satisfiable or not. A *satisfiability procedure* for T is any algorithm that solves the satisfiability problem for T.[1] Note that we can use free constants instead of variables to equivalently redefine the satisfiability problem for T as the problem of establishing the consistency of $T \cup S$ for a finite set S of ground literals. The *uniform word problem* for a theory T amounts to establishing whether $T \models \Gamma \Rightarrow e$, where Γ is a conjunction of Σ-equalities, e is a Σ-equality, and all the variables in $\Gamma \Rightarrow e$ are (implicitly) universally quantified.

Given an inference system R composed of inference rules (written as $P \vdash C$), the binary relation $\vdash_{\mathsf{R}}$ is defined on formulas as follows: $\Phi \vdash_{\mathsf{R}} \Phi'$ if Φ' can be derived from Φ by applying a rule in R. The reflexive and transitive closure of $\vdash_{\mathsf{R}}$, denoted by $\vdash^*_{\mathsf{R}}$, is called the *derivation relation* of R. Also, a *derivation* in R is a sequence $\Phi \vdash_{\mathsf{R}} \Phi' \vdash_{\mathsf{R}} \Phi'' \vdash_{\mathsf{R}} \cdots$. A formula Φ is in *normal form w.r.t.* $\vdash_{\mathsf{R}}$ if there is no derivation in R starting from Φ. The relation $\vdash^*_{\mathsf{R}}$ is *terminating* if there is no infinite derivation. We will write the *configuration* Γ, Δ to denote a formula $\Gamma \wedge \Delta$, where Γ is a conjunction of equalities and Δ is a conjunction of disequalities.

2.1 Combination of Theories

In the sequel, let Σ_1 and Σ_2 be two disjoint signatures (i.e. $\Sigma_1 \cap \Sigma_2 = \emptyset$) and T_i be a Σ_i-theory for $i = 1, 2$. A $\Sigma_1 \cup \Sigma_2$-term t is an i-*term* if it is a variable or it has the form $f(t_1, ..., t_n)$, where f is in Σ_i (for $i = 1, 2$ and $n \geq 0$). Notice that

[1] The satisfiability of any quantifier-free formula can be reduced to the satisfiability of sets of literals by converting to disjunctive normal form and then splitting on disjunctions, e.g., checking whether $S_1 \vee S_2$ (where S_1 and S_2 are conjunction of literals) is T-satisfiable reduces to checking the T-satisfiability of both S_1 and S_2.

a variable is both a 1-term and a 2-term. A non-variable subterm s of an i-term is *alien* if s is a j-term, and all superterms of s are i-terms, where $i, j \in \{1, 2\}$ and $i \neq j$. An i-term is *i-pure* if it does not contain alien subterms. A literal is i-pure if it contains only i-pure terms. A formula is said to be *pure* if there exists $i \in \{1, 2\}$ such that every term occurring in the formula is i-pure. We will write the *configuration* $\Phi_1; \Phi_2$ to denote a formula $\Phi_1 \wedge \Phi_2$, where Φ_i is a conjunction of i-pure literals ($i = 1, 2$).

In this paper, we shall consider the problem of solving the satisfiability problem for $T_1 \cup T_2$ (i.e. the problem of checking the $T_1 \cup T_2$-satisfiability of conjunctions of $\Sigma_1 \cup \Sigma_2$-literals) by using the satisfiability procedures for T_1 and T_2. For certain theories, more basic algorithms exist which can be used to build satisfiability procedures, e.g. canonizers and solvers for the class of Shostak theories (see below for a formal definition). When such algorithms exist for either T_1, T_2, or both, we are interested in using them to solve the satisfiability problem for $T_1 \cup T_2$. In order to know which basic algorithms are available for T_1 and T_2 and what are the assumptions on T_1 and T_2, the following notions and results are useful.

Definition 1. [20] A conjunction Γ of Σ-literals is convex in a Σ-theory T iff for any disjunction $\bigvee_{i=1}^{n} x_i = y_i$ (where x_i, y_i are variables and $i = 1, ..., n$) we have that $T \cup \Gamma \models \bigvee_{i=1}^{n} x_i = y_i$ iff $T \cup \Gamma \models x_i = y_i$, for some $i \in \{1, ..., n\}$. A Σ-theory T is *convex* iff all the conjunctions of Σ-literals are convex. A Σ-theory T is *stably-infinite* iff for any T-satisfiable Σ-formula φ, there exists a model of T whose domain is infinite and which satisfies φ. A *Nelson-Oppen* theory (**NO**-theory, for short) is a stably-infinite theory which admits a satisfiability algorithm. A **NOconvex**-theory is a convex **NO**-theory. The class of **NO**-theories (resp. **NOconvex**-theories) is denoted by **NO** (resp. **NOconvex**).

Theorem 1. **NO** *and* **NOconvex** *are closed under disjoint-union.*

Definition 2. A *solver* (denoted by *solve*) for a Σ-theory T is a function which takes as input a Σ-equality $s = t$ and such that (a) $solve(s = t)$ returns $false$, if $T \models s \neq t$, or (b) $solve(s = t)$ returns a substitution $\lambda = \{x_1 \leftarrow t_1, ..., x_n \leftarrow t_n\}$ such that (b.1) x_i is a variable occurring in s or t for $i = 1, ..., n$, (b.2) x_i does not occur in any t_j for $i, j = 1, ..., n$, and (b.3) $T \models s = t \Leftrightarrow \exists y_1, ..., y_m . \bigwedge_{i=1}^{n} x_i = t_i$, where $y_1, ..., y_m$ ($m \geq 0$) are "fresh" variables s.t. y_k does not occur in s or t, for all $k = 1, ..., m$. A conjunction of Σ-equalities is in *solved form* iff it has the form $\bigwedge_{i=1}^{n} x_i = t_i$, which will be denoted by $\hat{\lambda}$, where $\lambda = \{x_1 \leftarrow t_1, ..., x_n \leftarrow t_n\}$ is the substitution returned by *solve*. A *canonizer canon* for a Σ-theory T is an idempotent function from Σ-terms to Σ-terms such that $T \models a = b$ iff $\models canon(a) = canon(b)$. A *Shostak* theory is a convex theory which admits a solver and a canonizer. A **SH**-theory is a stably-infinite Shostak theory. The class of **SH**-theories is denoted by **SH**.

We assume **SH**-theories to be stably-infinite since this is necessary to combine them with other theories as suggested by many recent papers (see e.g. [18]). This

is not too restrictive since, as shown in [3], any convex theory with no trivial models is stably-infinite.

Proposition 1. [18] **SH** $\subseteq$ **NOconvex** $\subseteq$ **NO**.

3 Rational Reconstruction of Combination Schemas

Let T_i be a Σ_i-theory ($i = 1, 2$) such that $\Sigma_1 \cap \Sigma_2 = \emptyset$. We consider the problem of building a satisfiability procedure for $T_1 \cup T_2$. As a preliminary step, we consider a *purification process* converting any conjunction Φ of $\Sigma_1 \cup \Sigma_2$-literals into a conjunction of pure literals. Such a process is achieved by replacing each alien subterm t by a new variable x and adding the equality $x = t$ to Φ. This mechanism, called *variable abstraction*, is repeatedly applied to Φ until no more alien subterms t can be abstracted away. Obviously, the purification process always terminates yielding $\Phi_1 \wedge \Phi_2$, where Φ_i is a conjunction of Σ_i-literals ($i = 1, 2$) such that $\Phi_1 \wedge \Phi_2$ is equisatisfiable to Φ in $T_1 \cup T_2$. In the rest of this paper, without loss of generality, we consider the satisfiability of formulae of the form $\Phi_1 \wedge \Phi_2$ (or, equivalently, of configurations $\Phi_1; \Phi_2$), where Φ_i is a conjunction of i-pure literals.

Our combination schemas are specified by inference systems. To prove that an inference system R yields a satisfiability procedure, we follow a three steps methodology. First, we show that the derivation relation $\vdash_{\mathsf{R}}$ induced by R is terminating. Second, we prove that $\vdash_{\mathsf{R}}$ preserves (un-)satisfiability. Finally, we check that the normal forms defined by $\vdash_{\mathsf{R}}$ (i.e. configurations to which no rule in R can be applied) distinct from $false$ must be satisfiable. The proof of the last step proceeds by contradiction showing that a normal form distinct from $false$ cannot be unsatisfiable by using the following (technical) lemma from which the proof of correctness of Nelson-Oppen schema in [25] essentially depends.

Lemma 1. [25] If T_1 and T_2 are two signature-disjoint stably-infinite theories, then any conjunction $\Phi_1 \wedge \Phi_2$ of pure quantifier-free formulas is $T_1 \cup T_2$-satisfiable if and only if there exists some *identification* of shared variables in $Var(\Phi_1) \cap Var(\Phi_2)$—i.e. an idempotent substitution ξ from variables to variables—such that $\Phi_i \xi \wedge \xi_{\neq}$ is T_i-satisfiable for $i = 1, 2$, where $\xi_{\neq}$ is the formula $\bigwedge_{\{(x,y) \mid x\xi \neq y\xi\}} x \neq y$.

3.1 Combining Theories in NOconvex

We assume that T_1 and T_2 are in **NOconvex**. This implies the availability of two satisfiability procedures for T_1 and T_2. We consider the inference system NO obtained as the union of NO_1 presented in Figure 1 and NO_2 obtained from NO_1 by symmetry.[2] NO takes configurations of the form $\Phi_1; \Phi_2$ where Φ_i is a set of Σ_i-literals ($i = 1, 2$). Rule $\mathsf{Contradiction}_1$ reports the T_1-unsatisfiability of Φ_1 (and hence of $\Phi_1 \wedge \Phi_2$), detected by the available satisfiability procedure.

[2] A symmetric rule for T_2 is obtained from a rule for T_1 by swapping indexes 1 and 2. A symmetric inference system for T_2 is the set of symmetric rules for T_2 obtained from the rules for T_1.

$$\mathsf{Contradiction}_1 \quad \Phi_1;\Phi_2 \vdash \mathit{false} \qquad \text{if } \Phi_1 \text{ is } T_1\text{-unsatisfiable}$$

$$\mathsf{Deduction}_1 \quad \Phi_1;\Phi_2 \vdash \Phi_1;\Phi_2 \cup \{x = y\} \text{ if } \begin{cases} \Phi_1 \textit{ is } T_1\text{-satisfiable}, \\ \Phi_1 \wedge x \neq y \text{ is } T_1\text{-unsatisfiable}, \\ \Phi_2 \wedge x \neq y \text{ is } T_2\text{-satisfiable, and} \\ x, y \in Var(\Phi_1) \cap Var(\Phi_2) \end{cases}$$

Fig. 1. The Inference System NO_1

Rule $\mathsf{Deduction}_1$ propagates equalities between shared variables known to the procedure for T_1 to that for T_2 (if they are not already known to the latter). The problem of checking whether the equality $x = y$ is a logical consequence of $T_1 \cup \Phi_1$ is transformed into the problem of checking the T_1-unsatisfiability of $\Phi_1 \cup \{x \neq y\}$ so to be able to exploit the available satisfiability procedure.

Theorem 2. *Let T_1, T_2 be two signature-disjoint* **NOconvex***-theories. Let* NO *be the inference system defined as the union $\mathsf{NO}_1 \cup \mathsf{NO}_2$, where NO_1 is depicted in Figure 1 and NO_2 is obtained from NO_1 by symmetry. The relation $\vdash^*_{\mathsf{NO}}$ is terminating and $\Phi_1;\Phi_2 \vdash^*_{\mathsf{NO}} \mathit{false}$ iff $\Phi_1 \wedge \Phi_2$ is $T_1 \cup T_2$-unsatisfiable.*

Indeed, NO specifies only the essence of the Nelson-Oppen schema. Such a schema can be refined to increase efficiency. For example, the satisfiability procedures of some theories, such as Linear Arithmetic, can be extended so to derive entailed equalities while checking for satisfiability (see, e.g. [16, 26]) thereby avoiding the guessing done when applying $\mathsf{Deduction}_1$. In this paper, we will not consider this kind of amelioration (the interested reader is referred to [8] for a comprehensive guide-line to the efficient implementation of the Nelson-Oppen schema). In the following, we will consider refinements of NO which allow us to incorporate solvers and canonizers for theories in **SH**.

3.2 Combining Theories in SH

We assume that T_1 and T_2 are in **SH**. This implies the availability of a canonizer $canon_i$ and a solver $solve_i$ for each theory T_i $(i = 1, 2)$.

Preliminary to the combination schema, we extend solvers (cf. Definition 2) to handle sets of equalities as follows: $solve(\emptyset)$ returns the identity substitution ϵ; $solve(\Gamma \cup \{s = t\}) = \mathit{false}$, if $solve(s = t) = \mathit{false}$; and $solve(\Gamma \cup \{s = t\}) = \sigma \circ solve(\Gamma\sigma)$, if $solve(s = t) = \sigma$, where $\circ$ denotes composition of substitutions.

We consider the inference system SH obtained as the union of SH_1 presented in Figure 2 and SH_2 obtained from SH_1 by symmetry. SH takes configurations of the form $\Gamma_1, \Delta_1; \Gamma_2, \Delta_2$, where Γ_i is a set of Σ_i-equalities and Δ_i is a set of Σ_i-disequalities for $i = 1, 2$. Rule $\mathsf{Solve} - \mathsf{fail}_1$ reports the T_1-unsatisfiability of Γ_1 (and hence of $\Gamma_1 \wedge \Delta_1 \wedge \Gamma_2 \wedge \Delta_2$) detected by $solve_1$. Rule $\mathsf{Solve} - \mathsf{success}_1$ replaces the Σ_1-equalities Γ_1 with their solved form which is obtained again by using $solve_1$. This is important for the next two rules. Dealing with solved

$$\mathsf{Solve}-\mathsf{fail}_1 \quad \Gamma_1, \Delta_1; \Gamma_2, \Delta_2 \vdash \mathit{false} \quad \text{if } \mathit{solve}_1(\Gamma_1) = \mathit{false}$$

$$\mathsf{Solve}-\mathsf{success}_1 \quad \Gamma_1, \Delta_1; \Gamma_2, \Delta_2 \vdash \widehat{\gamma_1}, \Delta_1; \Gamma_2, \Delta_2 \text{ if } \begin{cases} \Gamma_1 \text{ is not in solved form,} \\ \gamma_1 = \mathit{solve}_1(\Gamma_1) \neq \mathit{false} \end{cases}$$

$$\mathsf{Contradiction}_1 \quad \widehat{\gamma_1}, \Delta_1 \cup \{s \neq t\}; \Gamma_2, \Delta_2 \vdash \mathit{false} \quad \text{if } \mathit{canon}_1(s\gamma_1) = \mathit{canon}_1(t\gamma_1)$$

$$\mathsf{Deduction}_1 \quad \widehat{\gamma_1}, \Delta_1; \widehat{\gamma_2}, \Delta_2 \vdash \widehat{\gamma_1}, \Delta_1; \widehat{\gamma_2} \cup \{x = y\}, \Delta_2 \text{ if } \begin{cases} \mathit{canon}_1(x\gamma_1) = \mathit{canon}_1(y\gamma_1), \\ \mathit{canon}_2(x\gamma_2) \neq \mathit{canon}_2(y\gamma_2), \\ x, y \in \mathit{Var}(\gamma_1) \cap \mathit{Var}(\gamma_2) \end{cases}$$

Fig. 2. The Inference System SH_1

forms allows us to simply determine entailed equalities (possibly between shared variables, see $\mathsf{Deduction}_1$) using canonizers. Hence, it is possible to lazily report unsatisfiability as soon as we find a disequality whose corresponding equality is entailed (see $\mathsf{Contradiction}_1$). Indeed, convexity allows us to handle disequalities one by one.

Theorem 3. *Let T_1, T_2 be two signature-disjoint* **SH***-theories. Let* SH *be the inference system defined as the union* $\mathsf{SH}_1 \cup \mathsf{SH}_2$*, where* SH_1 *is depicted in Figure 2 and* SH_2 *is obtained by symmetry. The relation* $\vdash^*_{\mathsf{SH}}$ *is terminating and* $\Gamma_1, \Delta_1; \Gamma_2, \Delta_2 \vdash^*_{\mathsf{SH}} \mathit{false}$ *iff* $\Gamma_1 \wedge \Delta_1 \wedge \Gamma_2 \wedge \Delta_2$ *is* $T_1 \cup T_2$*-unsatisfiable.*

It is easy to see that a strategy applying rules $\mathsf{Solve}-\mathsf{fail}_1$, $\mathsf{Solve}-\mathsf{success}_1$, and $\mathsf{Contradiction}_1$ in SH to a configuration $\Gamma_1, \Delta_1; \Gamma_2, \Delta_2$ yields the same result as that of applying rule $\mathsf{Contradiction}_1$ in NO to $\Gamma_1 \cup \Delta_1; \Gamma_2 \cup \Delta_2$. Similarly, the application of rules $\mathsf{Solve}-\mathsf{success}_1$ and $\mathsf{Deduction}_1$ in SH simulates the application of $\mathsf{Deduction}_1$ in NO; showing that equalities between shared variables can be derived by invoking a solver (and a canonizer) rather than resorting to guessing as for NO when applying the rule $\mathsf{Deduction}_i$ $(i = 1, 2)$. This is one of the key insights underlying Shostak schema. Furthermore, similarly to [13], the abstract schema presented here seems to emphasize the importance of the solver w.r.t. the canonizer. In fact, if the solved form returned by the solver is also canonical, the canonizer can be trivially implemented as the identity function. Nonetheless, we believe that the concept of canonizer is quite important for two crucial reasons. First, it offers the entry point to refinements of the proposed schema to increase efficiency. In fact, solving a set of equalities in "one-shot", as done when applying rule $\mathsf{Solve}-\mathsf{success}_1$, may not be as efficient as solving equalities incrementally, as done e.g. in [22, 14]. This can be incorporated in our schema by refining the inference system SH along the lines described in [4] so that the solver is applied to only one equality at a time and the canonizer needs to return a canonical form for arbitrary terms. The second reason is that a generalization of the concept of canonizer will be the basis for a new combination schema as we will see in Section 4.

3.3 Combining a Theory in NOconvex with One in SH

Without loss of generality, let us assume that T_1 is in **NOconvex** and that T_2 is in **SH**. This situation frequently arises in practical verification problem, e.g. the union of a theory in **SH** and $\mathcal{E}$ (which is *not* in **SH**). We consider the inference system NS obtained as the union of NO_1 in Figure 1 and SH_2, the symmetric of SH_1 in Figure 2. NS takes configurations of the form $\Phi_1; \Gamma_2, \Delta_2$ where Φ_1 is a set of Σ_1-literals, Γ_2 is a set of Σ_2-equalities, and Δ_2 is a set of Σ_2-disequalities. We furtherly assume that when a rule of NO is applied, $\Phi_1; \Gamma_2, \Delta_2$ stands for $\Phi_1; \Gamma_2 \cup \Delta_2$ and when a rule of SH is applied, $\Phi_1; \Gamma_2, \Delta_2$ is considered as $\Gamma_1, \Delta_1; \Gamma_2 \cup \Delta_2$, where $\Phi_1 = \Gamma_1 \cup \Delta_1$ and Γ_1 (Δ_1) is a set of Σ_1-equalities (-disequalities, respectively). NS can be seen as an abstract version of the one proposed in [3].

Theorem 4. *Let T_1, T_2 be two signature-disjoint theories such that T_1 is in* **NOconvex** *and T_2 is in* **SH**. *Let NS be the inference system defined as the union $\mathsf{NO}_1 \cup \mathsf{SH}_2$, where NO_1 is in Figure 1 and SH_2 is obtained from SH_1 in Figure 2 by symmetry. The relation $\vdash^*_{\mathsf{NS}}$ is terminating and $\Phi_1; \Gamma_2, \Delta_2 \vdash^*_{\mathsf{NS}} false$ iff $\Phi_1 \wedge \Gamma_2 \wedge \Delta_2$ is $T_1 \cup T_2$-unsatisfiable.*

Let $T_1, ..., T_k$ and $T_{k+1}, ..., T_{k+n}$ be k theories in **NOconvex** and n theories in **SH**, respectively, and such that $\Sigma_i \cap \Sigma_j \neq \emptyset$ for $i, j = 1, \ldots, k+n$, $i \neq j$, and $n, k \geq 1$. It is possible to modularly build a satisfiability procedure for $T = \bigcup_{j=1}^{k+n} T_j$ as follows. Repeatedly use NO to obtain a satisfiability procedure for $U_0 = \bigcup_{j=1}^{k} T_j$, then repeatedly use NS to build satisfiability procedures for $U_1 = U_0 \cup T_{k+1}, \ldots, U_n = U_{n-1} \cup T_{k+n}$, where U_n is T. An alternative would be to repeatedly use SH to construct satisfiability procedures for unions of two theories in **SH**, followed by a repeated use of NO on the resulting theories. The particular case of combining $\mathcal{E}$ with one or more theories in **SH** (i.e. $k = 1$) has been extensively studied by many researchers [5, 22, 14, 3, 13, 17, 23, 4], Shostak [24] being the first. The correctness of the combination schemas outlined above immediately follows from the correctness of NO, SH, NS, the fact that the union of two theories in **NOconvex** is also in **NOconvex** (Theorem 1), and that **SH** is contained in **NOconvex** (Proposition 1). Similar results are given in [18]. Finally, let us mention still another possibility to combine k theories in **NOconvex** and n theories in **SH**. It would be possible to slightly modify our inference rules to take into account $k + n$ theories; configurations would be of the form $\Phi_1; \ldots; \Phi_k; \Gamma_{k+1}, \Delta_{k+1}; \ldots; \Gamma_{k+n}, \Delta_{k+n}$ and the rule $\mathsf{Deduction}$ would propagate an equality between shared variables, deduced in one theory, to the other $(k + n) - 1$ theories. At this point, it would not be difficult to modify the proof of correctness for NS to show that the resulting rules (taken from $\mathsf{NO}_1, \ldots, \mathsf{NO}_k, \mathsf{SH}_{k+1}, \ldots, \mathsf{SH}_{k+n}$) yield a satisfiability procedure for T. The resulting proof would be a bit more involved because of the more complex notation.

4 Combining ECANconvex-Theories

Although the combination schemas of Section 3 are already sufficient to combine several theories either in **NOconvex**, **SH**, or both, we investigate how to find a generic combination schema which features the modularity of NO and retains some of the efficiency of SH. To this end, we introduce a new basic building block which generalizes the concept of canonizer for **SH**-theories and can be modularly combined either to (1) build a satisfiability procedure for the union of theories (admitting extended canonizers) by a schema which allows to efficiently propagate entailed equalities as in SH but does not require to solve equalities, or to (2) obtain an extended canonizer out of two extended canonizers in a modular way, thereby showing that the class of theories for which an extended canonizer exists is closed under disjoint union.

4.1 Extended Canonizers and ECANconvex-Theories

Definition 3. Let T be a Σ-theory with decidable uniform word problem, and let Γ be a conjunction of Σ-equalities. Given any T-satisfiable Γ, an *extended canonizer* for T is a function $ecan(\Gamma) : T(\Sigma, X) \to T(\Sigma \cup K(\Gamma), X)$, such that, for any terms s, t, we have $T \models \Gamma \Rightarrow s = t$ iff $ecan(\Gamma)(s) = ecan(\Gamma)(t)$, where $K(\Gamma)$ is a finite set of fresh constant symbols such that $\Sigma \cap K(\Gamma) = \emptyset$.

ECANconvex denotes the class of convex theories admitting an extended canonizer.

The concept of extended canonizer presents many similarities with the function $can(\Gamma)$ in [22].[3] An important difference is that our extended canonizers can be modularly combined to yield satisfiability procedures for union of disjoint theories (see Section 4.3 below). However, [23] describes a solution to the problem of combining $\mathcal{E}$ with several theories in **SH** by means of an interesting generalization of Shostak algorithm which only requires to build a canonizer for the union of the theories (which is always possible for convex theories [17]) and invokes the solvers for the theories being combined.

If a theory T admitting an extended canonizer *ecan* is also convex, then it is always possible to build a satisfiability procedure for T by recalling that $\Gamma \wedge \neg e_1 \wedge \cdots \wedge \neg e_n$ is T-unsatisfiable if and only if there exists some $i \in \{1, \ldots, n\}$ such that $\Gamma \wedge \neg e_i$ is T-unsatisfiable, or equivalently $T \models \Gamma \Rightarrow e_i$. This immediately implies the following proposition.

Proposition 2. ECANconvex $\subseteq$ NOconvex.

Although we can always decide the uniform word problem for a convex theory T by invoking a satisfiability procedure, it is not clear whether an extended canonizer always exists for T in **NOconvex**. Recall that in the Definition 3 of

[3] $can(\Gamma)(t)$ returns a canonical form of the term t in which any (uninterpreted) subterm that is congruent to a known left hand side in an equation of Γ is replaced by the associated right hand side.

extended canonizer, we require it to return terms over $T(\Sigma \cup K(\Gamma), X)$ where $K(\Gamma)$ must be a *finite* set of fresh constant symbols. The intuition is that a constant in $K(\Gamma)$ is the representative of an equivalence class induced by $T \cup \Gamma$. Since $K(\Gamma)$ is finite, extended canonizers can only be built for a theory T such that, for each finite set Γ of ground equalities, the equivalence relation induced by $T \cup \Gamma$ has a finite number of equivalence classes. So, the problem of determining that the inclusion in Proposition 2 is strict amounts to proving the existence of a theory T in **NOconvex** such that for some set Γ of ground equalities, $T \cup \Gamma$ induces an equivalence relation with an infinite number of equivalence classes. We conjecture that such a theory exists and hence conclude the inclusion in Proposition 2 is strict.

4.2 Extended Canonizers, Solvers, Canonizers, and Satisfiability Procedures

We describe the relationships between theories in **ECANconvex**, those in **SH**, and some in **NOconvex** which are not in **SH**.

Proposition 3. **SH** $\subseteq$ **ECANconvex**, i.e. theories in **SH** admits an extended canonizer.

Proof. Let T be an **SH**-theory and *solve* and *canon* its solver and canonizer, respectively. We define an extended canonizer *ecan* for T, as follows. If $solve(\Gamma) = false$, then *ecan* is undefined. If $solve(\Gamma)$ returns a substitution λ, then $ecan(\Gamma)(s)$ returns $canon(s\lambda)$. □

The proof of the Proposition above suggests how to reduce the uniform word problem $T \models \Gamma \Rightarrow s = t$ for a theory T in **SH** to the word problem $T \models s\lambda = t\lambda$, where λ is the substitution obtained by iteratively applying *solve* to Γ. In turn, $T \models s\lambda = t\lambda$ reduces to checking whether $canon(s\lambda)$ is syntactically equal to $canon(t\lambda)$ (a similar observation is done in [13]). The key observation here is that substituting equalities in Γ with their solved form $\hat{\lambda}$ can be done without backtracking thanks to the properties of *solve*. This is not possible for some theories whose uniform word problem can be decided by using an extended canonizer. For example, $\mathcal{E}$ admits efficient algorithms to solve its uniform word problem (see, e.g. [9]) but it is easy to show that it does not admit a solver (see e.g. [18]); so $\mathcal{E}$ is not in **SH**.

Proposition 4. $\mathcal{E} \in$ **ECANconvex**, i.e. $\mathcal{E}$ admits an extended canonizer.

The extended canonizer for $\mathcal{E}$ is a total function since any set Γ of ground equalities is $\mathcal{E}$-satisfiable. Because of Proposition 4 and the fact that $\mathcal{E}$ is not in **SH**, the inclusion between **SH** and **NOconvex** in Proposition 3 is strict. There are other interesting theories not in **SH** for which an extended canonizer exists as the following Proposition shows.

Proposition 5. Let C_f be the theory axiomatized by $\forall X, Y. f(X, Y) = f(Y, X)$. Then, the theory $C'_f := C_f \cup \{\exists X, Y. X \neq Y\}$ is not in **SH** and admits an extended canonizer.

Also, associative-commutative theories can be shown to admit extended canonizers by using the result in [2].

An efficient implementation of the uniform word problem for the theory of equality and commutative symbols based on a fast congruence closure algorithm is given in [9]. This can be used as the basis for efficient extended canonizers.

4.3 Extended Canonizers and Combination of ECANconvex-Theories

For technical reasons (that will become clear in a moment), we introduce the concept of *equational simplifier*, which is a partial function *eqs* taking conjunctions of equalities and returning a function whose input is an equality and which returns either *true* or $false$ such that for any conjunction of equalities Γ and equality e, (a) *eqs* is defined for Γ and e iff Γ is T-satisfiable, and (b) $eqs(\Gamma)(e)$ is *true* if $T \models \Gamma \Rightarrow e$, and $false$ otherwise. Indeed, for T in **ECANconvex**, clause (b) can be restated as follows: for any T-satisfiable Γ and any equality $s = t$, $eqs(\Gamma)(s = t) = true$ iff $ecan(\Gamma)(s) = ecan(\Gamma)(t)$. In the rest of this section, we assume that equational simplifiers are defined in terms of extended canonizers and we study the problem of building satisfiability procedures for unions of **ECANconvex**-theories. There are two possibilities. First, adapt NO to combine satisfiability procedures built out of equational simplifiers. (To see this, observe that equational simplifiers decides uniform word problems since these can be transformed to satisfiability problems as described in Section 4.1.) This gives a straightforward reformulation of the inference rules in NO where side conditions are expressed in terms of *eqs*. Since this is easy, the details are left to the reader. Second, build an equational simplifier for the union of theories and then derive the corresponding satisfiability procedure. In the following, we develop the second possibility. Let T_i be a Σ_i-theory in **ECANconvex** and $ecan_i$ its extended canonizer for $i = 1, 2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$. First, we show how to obtain an equational simplifier $eqs_{1,2}$ for $T_1 \cup T_2$ by using a variant of NO restricted to equalities. Then, we show how to build an extended canonizer $ecan_{1,2}$ for $T_1 \cup T_2$ out of $ecan_1, ecan_2$ and $eqs_{1,2}$. The reader may ask why we need to build the equational simplifier for $T_1 \cup T_2$ to be able to build an extended canonizer. The answer is in the definition of extended canonizer which requires Γ to be satisfiable for $ecan(\Gamma)$ to be defined. So, we need to check the $T_1 \cup T_2$-satisfiability of conjunctions of $\Sigma_1 \cup \Sigma_2$-equalities to decide whether $ecan_{1,2}$ is defined.

Lemma 2. Let T_1 and T_2 be two signature-disjoint convex and stably infinite theories. If an equational simplifier eqs_i is known for T_i (for $i = 1, 2$), then it is possible to construct an equational simplifier *eqs* for $T_1 \cup T_2$ using the combination method described in Figure 3.

Notice that the result above can be repeatedly applied to build an equational simplifier for the union of n signature-disjoint, convex, and stably-infinite theories $T_1, \ldots, T_n$. So, a satisfiability procedure for $T_1 \cup \cdots \cup T_n$ can be immediately obtained. However, this still does not answer the question: does there exist an extended canonizer $ecan_{1,2}$ for $T_1 \cup T_2$ given two extended canonizers

Given a set Γ of equalities and an equality $s = t$, the following procedure shows how to construct *eqs* for $(\Gamma, s = t)$, when defined. Let EEC be the inference system defined as the union $\mathsf{EEC}_1 \cup \mathsf{EEC}_2$, where EEC_1 is depicted in Figure 4 and EEC_2 is obtained by symmetry.

1. Purify Γ into $\Gamma_1; \Gamma_2$.
2. If $\Gamma_1; \Gamma_2 \vdash^*_{\mathsf{EEC}} false$, then *eqs* is undefined for $(\Gamma, _)$.
3. Otherwise, let $\Gamma'_1; \Gamma'_2$ be the normal form w.r.t. $\vdash_{\mathsf{EEC}}$ such that $\Gamma_1; \Gamma_2 \vdash^*_{\mathsf{EEC}} \Gamma'_1; \Gamma'_2$. Furthermore, purify $x = s \wedge y = t$, where x, y are new variables not occurring in $Var(\Gamma'_1 \wedge \Gamma'_2)$. Let $\Gamma''_1; \Gamma''_2$ be the result of purifying $\Gamma'_1 \wedge \Gamma'_2 \wedge x = s \wedge y = t$.
4. Let $\Gamma'''_1; \Gamma'''_2$ be the normal form w.r.t. $\vdash_{\mathsf{EEC}}$ such that $\Gamma''_1; \Gamma''_2 \vdash^*_{\mathsf{EEC}} \Gamma'''_1; \Gamma'''_2$. This normal form is necessarily different from $false$ since $\Gamma_1 \wedge \Gamma_2$ is $T_1 \cup T_2$-satisfiable and x, y are different new variables.
5. If there exists $i \in \{1, 2\}$ such that $x, y \in Var(\Gamma''_i)$, then $eqs(\Gamma)(s = t)$ is defined and it is equal to $eqs_i(\Gamma'''_i)(x = y)$.
6. Otherwise ($x \in Var(\Gamma''_i), y \in Var(\Gamma''_j)$, for $i \neq j$), $eqs(\Gamma)(s = t)$ is defined, and it is equal to $true$ if there exists $z \in Var(\Gamma''_1) \cap Var(\Gamma''_2)$ such that $eqs_i(\Gamma'''_i)(x = z) = eqs_j(\Gamma'''_j)(y = z) = true$, otherwise it is defined as $false$.

Fig. 3. Equational Simplifier for the Union of Theories

$ecan_1$ and $ecan_2$ for T_1 and T_2, respectively, and an equational simplifier $eqs_{1,2}$ for their union? To answer this question (constructively), we analyze the equational simplifier for $eqs_{1,2}$ for $T_1 \cup T_2$ given in Figure 3 and we show how an extended canonizer can be obtained. The key technique underlying the analysis consists of unfolding the fresh variables (abstracting alien subterms) introduced by purification so to get terms back in the right signature. This unfolding must be done with care since we must take into account the equivalence relation on fresh variables induced by the propagation of equalities between shared variables.

Theorem 5. ECANconvex *is closed under disjoint union.*

$$\mathsf{Contradiction}_1 \quad \Gamma_1; \Gamma_2 \vdash false \qquad \text{if } eqs_1(\Gamma_1) \text{ is undefined}$$

$$\mathsf{Deduction}_1 \quad \Gamma_1; \Gamma_2 \vdash \Gamma_1; \Gamma_2 \cup \{x = y\} \text{ if } \begin{cases} eqs_1(\Gamma_1) \text{ is defined,} \\ eqs_2(\Gamma_2) \text{ is defined,} \\ eqs_1(\Gamma_1)(x = y) = true, \\ eqs_2(\Gamma_2)(x = y) = false, \\ x, y \in Var(\Gamma_1) \cap Var(\Gamma_2) \end{cases}$$

Fig. 4. The Inference System EEC_1

5 Conclusions and Future Work

We have presented combination schemas for disjoint unions of (a) two theories in **NOconvex**, (b) two theories in **SH**, and (c) one theory in **NOconvex** with one in **SH**. We have shown how such schemas are related to Nelson-Oppen and Shostak approaches to combination as well as with many of the refinements available in the literature. Our formalization highlights the key ideas underlying each combination and allows us to derive proofs of correctness which are easy to grasp. We believe this is a valuable synthesis for further investigations. To justify this claim, we have introduced the concept of extended canonizer which abstracts algorithms for deciding the uniform word problem of a theory and it is modular, i.e. an extended canonizer can be built out of the extended canonizers for the component theories. This is in contrast to the lack of modularity of solvers for Shostak combination schema. Another advantage is the fact that it can be easily implemented in terms of solvers and canonizers for Shostak theories or by rewriting techniques as suggested e.g. in [1].

There are several main lines for future work. First, we want to derive a more precise characterization of the theories admitting an extended extended canonizer. In this respect, a promising line of research would be to study for which theories the uniform word problem can be reduced to a word problem. Second, we want to study the complexity of extended canonizers in the union of theories. We believe it would be interesting to apply our combination results to polynomial time decidable uniform word problems as described in [12]. Third, we intend to empirically evaluate the efficiency of extended canonizers by conducting some experiments in haRVey [6]. The interest here is to obtain an efficient combination between extended canonizers and propositional solvers. This requires to equip extended canonizers with the capability of generating useful theory-specific facts which, once projected into the propositional domain, allow to reduce the search space. Finally, we plan to study how extended canonizers can be used when non-convex theories are combined.

References

1. A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Info. and Comp.*, 183(2):140–164, June 2003.
2. L. Bachmair, A. Tiwari, and L. Vigneron. Abstract Congruence Closure. *Journal of Automated Reasoning*, 31(2):129–168, 2003.
3. C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In *Proc. of the 4th Int. Workshop on Frontiers of Combining Systems*, volume 2309 of *LNCS*, pages 132–147, 2002.
4. S. Conchon and S. Krstić. Strategies for combining decision procedures. In *Proc. of the 9th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *LNCS*, pages 537–553. Springer-Verlag, April 2003.
5. D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *Proc. of the 13th Int. Conference on Automated Deduction*, volume 1104 of *LNCS*, pages 463–477. Springer-Verlag, 1996.

6. D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In I. C. S. Press, editor, *Proc. of the Int. Conf. on Software Engineering and Formal Methods (SEFM03)*, 2003.
7. N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. 1990.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Laboratories, 2003.
9. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the Common Subexpression Problem. *J. of the ACM*, 27(4):758–771, October 1980.
10. H. B. Enderton. *A Mathematical Introduction to Logic*. Ac. Press, Inc., 1972.
11. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In *Proc. of CAV'2001*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.
12. H. Ganzinger. Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In *Proc. 16th IEEE Symp. on Logic in Computer Science*, pages 81–92. IEEE Comp. Soc. Press, 2001.
13. H. Ganzinger. Shostak light. In *Proc. of the 18th Int. Conference on Automated Deduction*, volume 2392 of *LNCS*, pages 332–346. Springer-Verlag, jul 2002.
14. D. Kapur. A rewrite rule based framework for combining decision procedures. In *Proc. of the 4th Int. Workshop on Frontiers of Combining Systems*, volume 2309 of *LNCS*, pages 87–102. Springer-Verlag.
15. D. Kapur. Shostak's congruence closure as completion. In *Proc. of the 8th Int. Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*. Springer-Verlag, 1997.
16. D. Kapur and X. Nie. Reasoning about Numbers in Tecton. In *Proc.* 8^{th} *Inl. Symp. Methodologies for Intelligent Systems*, pages 57–70, 1994.
17. S. Krstić and S. Conchon. Canonization for disjoint unions of theories. In *Proc. of the 19th Int. Conference on Automated Deduction*, volume 2741 of *LNCS*, Miami Beach, FL, USA, 2003. Springer Verlag.
18. Z. Manna and C. G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 381–422. Springer, 2003.
19. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, Oct. 1979.
20. D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.
21. S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn (Full Version). Available at `http://www.loria.fr/~ranise/pubs/long-ictac04.ps.gz`.
22. H. Rueß and N. Shankar. Deconstructing Shostak. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28. IEEE Computer Society, June 2001.
23. N. Shankar and H. Rueß. Combining shostak theories. In *Proc. of the 13th Int. Conf. on Rewriting Techniques and Applications*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.
24. R. E. Shostak. Deciding combinations of theories. *J. of the ACM*, 31:1–12, 1984.
25. C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, 2003.
26. P. van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 5:303–319, 1992.

Real Time Reactive Programming in Lucid Enriched with Contexts*

Kaiyu Wan, Vasu Alagar, and Joey Paquet

Concordia University, Montreal, Canada
{ky_wan, alagar, paquet}@cs.concordia.ca

Abstract. We present a synchronous approach to real-time reactive programming in Lucid enriched with contexts as first class objects. The declarative intensional approach allows real-time reactive programs to manipulate both events and state-based representations of complex systems. We show the formal specification of the *Train-Gate-Controller* problem, a standard case study in real-time systems community, and formally verify the safety property.

Keywords: Real-time reactive programming, intensional programming, contexts, formal verification.

1 Introduction

Reactive systems continuously interact with their environment. The two properties that characterise reactive systems are that the process always reacts to a stimulus from its environment (*stimulus synchronisation*), and the time elapsed between a stimulus and its response is acceptable to the relative dynamics of the environment, so that the environment is still receptive to the response (*response synchronisation*). Reactive systems include many real-time systems that are subject to hard real-time requirements. Examples of such systems include railroad control systems, nuclear reactor control systems, and air traffic control systems. In this paper we discuss Lucid extended with *contexts* and clocks for programming real-time reactive systems.

The design of synchronous dataflow languages Lustre [2], and RLucid [10] are based on Lucid [12]. They have been used for reactive programming and verification approaches for such programs have been developed. Clocks were added to Lustre programs so that certain parts of the programs need not always run. This enabled the introduction of constrained reaction. In RLucid the operator *before* to deal with real time has been introduced. That is, one can write the expression E_1 *before* E_2 to determine whether the first value in the stream E_1 arrived at time $t_1 < t_2$, where t_2 is the time of arrival of the first value of E_2. SIGNAL [5] language manipulates *signals* that are timed sequences of typed values. In all these approaches time is discrete, and streams implicitly have the time dimension, although clocks associated with dimensions may be different. It is possible to write an expression in Lucid whose evaluation is context-dependent, the

* This work is supported by grants from the Natural Sciences and Engineering Research Council of Canada.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 387–402, 2005.
© Springer-Verlag Berlin Heidelberg 2005

context being $[d : t]$ where t is the (time) tag in the dimension d associated with the expression (stream). However, a context in Lucid can not be explicitly manipulated. This restricts the ability of Lucid to be an effective programming language for dealing with constraints. So we have extended Lucid by adding the capability to explicitly manipulate contexts. This is achieved by introducing *context* as a first class object in the language. From now on, we call this extended language as *Lucx* (Lucid extended with *contexts*).

The notion of *context* was introduced by McCarthy and later used by Guha [3] as a means of expressing assumptions made by natural language expressions in Artificial Intelligence (AI). Hence, a formula, which is an expression combining a sentence in AI with contexts, can express the exact meaning of the natural language expression. By making difference between *intension* and *extension* of an expression, *Intensional language*(Lucid) can express a natural language expression succinctly without loss of accuracy. According to Carnap, the real meaning of a natural language expression whose truth-value depends on the context in which it is uttered is its *intension*. The *extension* of that expression is its actual truth-value in the different possible contexts of utterance, evaluated [8]. For example the intension in the expression "E: the average temperature for this month here is greater than $0°C$" is itself. The two intensional operators in this expression are *this month* and *here*, which refer respectively to the time and space dimension. The extension of the expression varies according to the different evaluation context, which are different cities and months in this example. The major distinction between contexts in AI and in intensional programming language is that in the former case they are *rich objects* that are not *completely expressible* and in the later case they are *implicitly* expressible. The introduction of contexts explicitly as first class objects in Lucid improves the expressive power of Lucid in the following ways:

- Contexts can be dynamically modified through operators defined for contexts. New contexts can be dynamically created from those defined in a program.
- Context calculus provides compilation rules for calculating a context from a context expression, and evaluation rules for expressions over context expressions.
- In Lucid, the dimensions of a multidimensional stream can be named explicitly. A context is also multidimensional. It is possible to extract different sub-streams independently from a stream and manipulate them, by evaluating the sub-streams at dynamically changing context expressions.
- Different clocks may be used with different dimensions and the program will be able to combine them through contexts whose dimensions are clocks.
- Lucid programs can be written for continuous time models.

2 Lucx: Lucid with Contexts

Wadge and Ashcroft [12] invented Lucid in 1974 as a dataflow language. After evolving through different versions, Lucid has become an intensional language.

$$
\begin{array}{ll}
E ::= id & Q ::= \texttt{dimension}\ id \\
\quad \mid E(E_1, \ldots, E_n) & \quad \mid id = E \\
\quad \mid \texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E'' & \quad \mid id(id_1, \ldots, id_n) = E \\
\quad \mid \#E & \quad \mid Q\ Q \\
\quad \mid E\ \texttt{@}E'\ E'' & \\
\quad \mid E\ \texttt{where}\ Q &
\end{array}
$$

The abstract syntax of Lucid is given above. The operators @ and # are context navigation and query operators. The non-terminals E and Q respectively refer to *expressions* and *definitions*. The general form of evaluation in Lucid is $\mathcal{D}, \mathcal{P} \vdash E : v$ which means that in the definition environment $\mathcal{D}$, and in the evaluation context $\mathcal{P}$, expression E evaluates to v. The definition environment $\mathcal{D}$ retains the definitions of all of the identifiers that appear in a Lucid program. Formally, $\mathcal{D}$ and $\mathcal{P}$ are partial functions $\mathcal{D} : \mathbf{Id} \rightarrow \mathbf{IdEntry}$, $\mathcal{P} : \mathbf{Id} \rightarrow \mathbf{N}$, where **Id** is the set of all possible identifiers and **IdEntry** has five possible kinds of value such as: *Dimensions*, *Constants*, *Data Operators*, *Variables*, and *Functions*[8]. The evaluation context $\mathcal{P}$, associates a tag to each relevant dimension.

Example 1 is a program in Lucid to extract a value from the stream representing the natural numbers, beginning from the ubiquitous number 42. We arbitrarily pick the third value of the stream, which is assigned tag number two (indexes starting at 0). We also set the stream's variance in the d dimension. Intuitively, we can expect the program to return the value 44.

Example 1

```
N @.d 2
  where
      dimension d;
      N = if (#.d <= 0)  then 42 else (N+1) @.d (#.d-1);
  end;
```

The operational semantics for Lucid is defined in [8]. The implementation technique of evaluation for Lucid programs is an interpreted mode called *eduction*.

2.1 Contexts

In [1] we have defined contexts as first class objects in Lucid, introduced a set of operators on contexts, and given a semantics for evaluating expressions over context expressions. We have also shown that *Lucx* has the generality and expressivity to be used as *agent communication language*. In this section we review the basic definitions and give examples on context operators.

Informally a context is a reference to a multidimensional stream, making an explicit reference to the dimensions and the *tags* (indexes) along each dimension. The syntax for context is $[d_1 : x_1, \ldots, d_n : x_n]$, where $d_1, \ldots, d_n$ are dimension names, and x_i is the tag for dimension d_i. Given an expression E and a context c, the Lucx expression E @ c directs the eduction engine to evaluate E in the context c. The semantics for evaluation is an extension of the semantics given by Paquet [8]. Formally, contexts are defined as *a subset of a finite union of relations*. Let $DIM = \{d_1, d_2, \ldots, d_n\}$ denote a finite set of dimension

names. With each dimension d_i, a unique tag set X_i is associated. Let $TAG = \{X_1, \ldots, X_r\}$ denote the set of tags. We define the function $f_{dimtotag} : DIM \rightarrow TAG$, that associates with every $d_i \in DIM$ exactly one tag set X_j in TAG.

Definition 1. *Consider the relations*

$$P_i = \{d_i\} \times f_{dimtotag}(d_i) \quad 1 \leq i \leq n$$

A context C, given $(DIM, f_{dimtotag})$*, is a finite subset of* $\bigcup_{i=1}^{n} P_i$*. The* degree *of the context C is* $\mid \Delta \mid$*, where* $\Delta \subset DIM$ *includes the dimensions that appear in C.*

A context is written using *enumeration* syntax. The set enumeration syntax of a context C is $C = \{(d,x) \mid d \in \Delta, x \in f_{dimtotag}(d)\}$. We use the syntax $[d_{i_1} : x_{j_1}, \ldots, d_{i_k} : x_{j_k}]$ in *Lucx* to explicitly denote the aggregation of dimension, tag pairs. Note that the d_is need not be distinct, and

$$C \subseteq \bigcup_{i=1}^{n} P_i \subset DIM \times I, \quad I = \bigcup_{X_i \in TAG} X_i$$

Consequently, every subset of $\bigcup_{i=1,n} P_i$ is a context, but not every subset of $DIM \times I$ is a context. However, if $X_1 = X_2 \ldots = X_n$, every subset of $DIM \times I$ is a context. This follows from the fact that $f_{dimtotag}(d_i) = X_i$, $i = 1, \ldots, n$ implies that

$$\cup_{i=1}^{n} P_i = \cup_{i=1}^{n} (\{d_i\} \times I) = (\cup_{i=1}^{n} \{d_i\}) \times I = DIM \times I$$

We say a context C is *simple* (s_context), if $(d_i, x_i), (d_j, x_j) \in C \Rightarrow d_i \neq d_j$. A simple context C of degree 1 is called a *micro* (m_context) context.

Example 2. *Let* $DIM = \{X, Y, Z, U\}$*,* $TAG = \{\mathbb{N}, \{blue, red\}\}$*,* $f_{dimtotag}(X) = \mathbb{N}$*,* $f_{dimtotag}(Y) = \mathbb{N}$*,* $f_{dimtotag}(U) = \{blue, red\} = f_{dimtotag}(Z)$*. The subsets of* $P = P_1 \cup P_2 \cup P_3 \cup P_4$*, where* $P_1 = X \times \mathbb{N}, P_2 = Y \times \mathbb{N}, P_3 = Z \times \{blue, red\}, P_4 = U \times \{blue, red\}$ *are contexts.*

1. $c_1 = [X : 1.5, Y : red]$ *is not a subset of P, hence is not a context.*
2. $c_2 = [Z : blue]$ *is a m_context.*
3. $c_3 = [X : 3, Y : 2]$ *is a s_context of degree 2.*
4. $c_4 = [X : 3, X : 4, Y : 3, Y : 2, U : blue]$ *is a context of degree 3.*

We have defined several functions on contexts [1]. The basic functions dim_m and tag_m extract the dimension and the tag from a micro context. Extending their definitions, the functions *dim* and *tag* are defined for a context.

Example 3. *Consider the contexts introduced in Example 2. An application of dim and tag functions to these contexts produces the following results:*

1. *dim and tag are not defined for context* c_1.
2. $dim_m(c_2) = Z$*,* $tag_m(c_2) = blue$*.*
3. $dim(c_3) = \{X, Y\}$*,* $tag(c_3) = \{3, 2\}$*.*
4. $dim(c_4) = \{X, Y, U\}$*,* $tag(c_4) = \{3, 4, 2, blue\}$*.*

In general, a set of contexts may include contexts of different degrees. We use the syntax $Box[\Delta \mid p]$ to introduce a finite set of contexts in which all contexts are defined over $\Delta \subseteq DIM$, have the same degree $\mid \Delta \mid$, and the tags in every context satisfy the predicate p. For example, the set of contexts defined by $Box[X, U \mid \frac{x}{4} + \frac{u}{5} \leq 1]$, where $f_{dimtotag}(X) = f_{dimtotag}(U) = \mathbb{N}$ is given below:

$$\{[X:0, U:0..5], [X:1, U:0..3], [X:2, U:0..2], [X:3, U:0..1], [X:4, U:0]\}$$

A non-simple context is to be understood as a set of simple contexts, that may not be expressible in *Box* notation. The context $C_4 = [X:3, X:4, Y:3, Y:2, U:blue]$ in Example 2 is equivalent to the set of simple contexts $\{[X:3, Y:3, U:blue], [X:3, Y:2, U:blue], [X:4, Y:3, U:blue], [X:4, Y:2, U:blue]\}$. In [1] we have given a method to construct the set of simple contexts that is equivalent to a non-simple context.

2.2 Context Calculus

Context operators in *Lucx* are: *constructor* $[_ : _]$, *override* $[_ \oplus _]$, *difference* $[_ \ominus _]$, *choice* $[_ \mid _]$, *conjunction* $[_ \sqcap _]$, *disjunction* $[_ \sqcup _]$, *undirected range* $[_ \rightleftharpoons _]$, *directed range* $[_ \rightharpoonup _]$, *projection* $[_ \downarrow _]$, *hiding* $[_ \uparrow _]$, *substitution* $[_ / _]$, *comparison* $[_ = _]$, *superset* $[_ \supseteq _]$, and *subset* $[_ \subseteq _]$. The operators for sets of contexts, in particular for *Box*, are *join* $[_ \boxtimes _]$, *union* $[_ \boxplus _]$, and *intersection* $[_ \boxdot _]$. In this paper we have omitted the formal definitions. They can be found in [1]. Below, we give examples to illustrate some of the operator definitions.

1. Context Operators:
   (a) [*Constructor* $[_ : _]$]
   This operator constructs a *m_context* for a given dimension d, and tag $f_{dimtotag}(d)$.
   (b) [*Override* $_ \oplus _$]
   Let $c_1 = [d:1]$, $c_2 = [e:2]$, $c_3 = [e:5]$, $c_4 = [d:2, d:3, f:4]$,
   Then $c_1 \oplus c_2 = [d:1, e:2]$, $c_2 \oplus c_3 = [e:5]$, $c_3 \oplus c_2 = [e:2]$,
   $c_4 \oplus (c_1 \oplus c_2) = [d:1, e:2, f:4]$,
   $(c_4 \oplus c_1) \oplus c_2 = [d:1, e:2, f:4]$.
   (c) [*Difference* $_ \ominus _$]
   Let $c_1 = [d:3, d:2, e:3]$, $c_2 = [d:1, e:4]$, $c_3 = [d:1]$,
   Then $c_2 \ominus c_3 = [e:4]$, $c_1 \ominus (c_2 \ominus c_3) = c_1$.
   (d) [*Hiding* $_ \uparrow _$]
   Let $c_1 = [d:1, e:4, f:3]$, $c_2 = [d:3]$, $c_3 = [f:3]$, $\Delta = \{d, e\}$
   Then $c_1 \uparrow \Delta = [f:3]$, $c_2 \uparrow \Delta = \varnothing$, $c_3 \uparrow \Delta = [f:3]$.
   (e) [*Undirected range* $_ \rightleftharpoons _$]
   Let $c_1 = [d:1]$, $c_2 = [d:4]$, $c_3 = [e:3, d:1]$, $c_4 = [e:1, d:3]$,
   Then $c_1 \rightleftharpoons c_2 = \{[d:1], [d:2], [d:3], [d:4]\} = c_2 \rightleftharpoons c_1$
   $c_3 \rightleftharpoons c_4 = \{[e:1, d:1..3], [e:2, d:1..3], [e:3, d:1..3],$
2. Box operators:
   Let $DIM = \{X, Y, Z\}$, and $f_{dimtotag}(X) = f_{dimtotag}(Y) = f_{dimtotag}(Z) = \mathbb{N}$. Let $B_1 = Box[X, Y \mid x, y \in \mathbb{N} \wedge x + y = 5]$, and $B_2 = Box[Y, Z \mid y, z \in \mathbb{N} \wedge y = z^2 \wedge z \leq 3]$.
   We have $B_1 = \{[X:1, Y:4], [X:2, Y:3], [X:3, Y:2], [X:4, Y:1]\}$
   $B_2 = \{[Y:1, Z:1], [Y:4, Z:2], [Y:9, Z:3]\}$

(a) [Join:]
$B_1 \boxtimes B_2 = Box[X, Y, Z \mid x + y \leq 5 \wedge y = z^2 \wedge z \leq 3]$
$= \{[X:1, Y:4, Z:2], [X:4, Y:1, Z:1]\}$

(b) [Intersection:]
$B_1 \boxdot B_2 = Box[Y \mid x + y \leq 5 \wedge y = z^2 \wedge z \leq 3]$
$= \{[Y:1], [Y:4]\}$

(c) [Union:]
$B_1 \boxplus B_2 = Box[X, Y, Z \mid x + y \leq 5 \vee (y = z^2 \wedge z \leq 3)]$
$= \{[X:1, Y:4, Z:1..3], [X:2, Y:3, Z:1..3], [X:3, Y:2, Z:1..3],$
$[X:4, Y:1, Z:1..3], [X:1..3, Y:1, Z:1], [X:2..4, Y:4, Z:2],$
$[X:1..4, Y:9, Z:3]\}$

The following table shows the formal syntax for context expression C, and precedence rules for context operators [1].

syntax	precedence
$C ::= c \mid C = C$ $\mid C \supseteq C \mid C \subseteq C$ $\mid C \mid C \mid C/C$ $\mid C \oplus C \mid C \ominus C$ $\mid C \sqcap C \mid C \sqcup C$ $\mid C \rightleftharpoons C \mid C \rightharpoonup C$ $\mid C \downarrow D \mid C \uparrow D$	1. $\downarrow, \uparrow, /$ 2. $\mid$ 3. $\sqcap, \sqcup$ 4. $\oplus, \ominus$ 5. $\rightleftharpoons, \rightharpoonup$ 6. $=, \subseteq, \supseteq$

Similarly, we define a *Box* expression. The three *Box* operators have the same precedence. An expression $b_1 \boxdot b_2 \boxtimes b_3$ is evaluated from left to right.

2.3 Syntax and Semantics of Lucx

The abstract syntax for contexts in *Lucx* is given below.

$E ::= \mathbf{E} \,@\, \mathbf{E'}$
$\quad \mid [\mathbf{E_1} : \mathbf{E'_1}, \ \ldots, \ \mathbf{E_n} : \mathbf{E'_n}]$

But for this change, the rest of the original syntax is retained. The operator @ is the navigation operator, which evaluates an expression E in context E', where E' is an expression evaluating to a context. The syntactic construct $[E_1 : E'_1, \ \ldots, \ E_n : E'_n]$ is for introducing context as an enumerated aggregation of m_contexts. As reflected in the semantic rules below, $E \,@\, E'$ is evaluated. The complete operational semantics is defined in [8, 1].

$$\mathbf{E_{context}}: \frac{\mathcal{D}, \mathcal{P} \vdash E_{d_j} : id_j \quad \mathcal{D}(id_j) = (\texttt{dim}) \quad \mathcal{D}, \mathcal{P} \vdash E_{i_j} : v_j \quad v = [id_j \mapsto v_j]}{\mathcal{D}, \mathcal{P} \vdash [E_{d_1} : E_{i_1}, E_{d_2} : E_{i_2}, \ldots, E_{d_n} : E_{i_n}] : v}$$

$$\mathbf{E_{at(c)}}: \frac{\mathcal{D}, \mathcal{P} \vdash E' : P' \qquad \mathcal{D}, \mathcal{P}' \vdash E \ : \ v}{\mathcal{D}, \mathcal{P} \vdash E \ @E' : v}$$

Based upon the precedence rules and semantics, evaluation rules for context expressions are given. For instance, $E \,@\, c_1 \oplus c_2 \sqcap c_3 \ = \ E \,@\, (c_4 \sqcup c_1 \ \uparrow \ \Delta_{c_4})$, where $c_4 \ = \ c_2 \sqcap c_3$.

3 Real-Time Reactive Programming in Lucx

In this section we discuss the representation of *events*, *states*, *functions*, *contexts* and *boxes* of timed systems as streams in the language. We discuss the representations for both discrete and continuous time models.

We denote a set of clocks by $\mathcal{C}$. A *clock valuation* v is defined as a higher-order function $v : \mathcal{C} \rightarrow (\Omega \rightarrow \overline{\Pi})$ such that the function $v(c)$, $c \in \mathcal{C}$ is monotonically and synchronously increasing function. For continuous time model, $\Omega = \mathcal{R}^{\infty}$, and the function $v(c)$ is continuous for every $c \in C$. For discrete time model, $\Omega = \mathbb{N} \cup \{0\}$. For both time models, $v(c)(0) = 0$.

3.1 Global Clock

Let $\mathbb{C}$ denote the global clock, $\mathbb{N}$ denote the set of nonnegative integers, $\mathbb{R}$ denote the set of reals, and $\mathbb{R}^{\geq 0}$ the set of nonnegative reals. We assume that continuously varying time is modelled as $\overline{\Pi} = \{t \mid t \in \mathbb{R}^{\geq 0}\}$. The model of discrete time is $\overline{\Pi} = \mathbb{N} \cup \{0\} \cup \{+\infty\}$.

Event Streams. Let $\mathcal{E}$ denote a finite non-empty set of *events* in the formal model of the system. An event $e \in \mathcal{E}$ may occur any number of times within the system. The function $TIME : \mathcal{E} \rightarrow (\mathbb{N} \rightarrow \overline{\Pi})$ defines for $e \in \mathcal{E}$, the function $TIME(e)$, whose value at $k \in \mathbb{N}$ is $t_k = TIME(e)(k)$, $t_k \in \overline{\Pi}$, interpreted as the time of k-th occurrence of the event e. The function $TIME(e) = \{\langle k, t_k \rangle\}$ is represented in the language as a 1-dimensional stream $\bar{\mathbf{e}}$, $\bar{\mathbf{e}}_k = t_k$. In the language the representation for an event e under continuous time model is the stream $\bar{\mathbf{e}}$, and under discrete time model, the representation can be either $\bar{\mathbf{e}}$ or a boolean stream $\mathbf{e}$ with rank = $\{\mathbb{C}\}$, such that $\mathbf{e}$ @ $[\mathbb{C} : t_k] = true$. The function $COUNT : \mathcal{E} \rightarrow (\overline{\Pi} \rightarrow \mathbb{N})$ defines for $e \in \mathcal{E}$, the function $COUNT(e)$, whose value at $t \in \overline{\Pi}$ is $k = COUNT(e)(t)$, $k \in \mathbb{N}$ is the number of occurrences of the event e up to and including the time t. That is, the function $COUNT$ is a pseudo inverse of $TIME$ function. That is, $TIME(e)(0) = 0$; $TIME(e)(+\infty) = +\infty$, and $COUNT(e)(0) = 0$; $COUNT(e)(+\infty) = +\infty$.

Example 4. *Let the times for* $1^{st}, 2^{nd}, 3^{rd}, 4^{th} \ldots$ *occurrences of an event e be* $1, 4, 5, 7, \ldots$

For discrete time, the representation of the event e is the stream

$\mathbf{e} =$ *true false false true true false true false* $\ldots$

Meanwhile, the representation of the stream $\bar{\mathbf{e}}$ *is*

$\bar{\mathbf{e}} = 1 \quad 4 \quad 5 \quad 7 \ldots$

The representation of the stream COUNT(e) is

$COUNT(e) = 1 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \ldots$

Let the times for $1^{st}, 2^{nd}, 3^{rd}, 4^{th}$, *... occurrences of an event f be* $1.3, 4.5, 5.6, 7.8, \ldots$, *in some clock valuation. The representation is the stream*

$\bar{\mathbf{f}} = 1.3 \quad 4.5 \quad 5.6 \quad 7.8, \ldots$

The representation of the stream COUNT(f) is

$COUNT(f)(1) = 0, \ \ldots \ COUNT(f)(1.3) = 1, \ \ldots, \ COUNT(f)(2) = 1, \ \ldots, \ COUNT(f)(4.5) = 2, \ \ldots, \ COUNT(f)(5) = 2, \ \ldots \ COUNT(f)(5.6) = 3, \ \ldots$

The following primitive functions defined in the language are useful to manipulate event streams. Let *now* denote current clock valuation. The arguments to the following functions are streams corresponding to events.

- The function $\texttt{includes}(\mathbf{e},\mathbf{f})$ returns *true* if $TIME(e) \leq TIME(f)$, otherwise it returns false.
- The function $\texttt{sum}(\mathbf{e},\mathbf{f})$ returns the stream obtained by merging the two input streams. The resulting stream represents the event $e+f$, which occurs whenever e occurs or f occurs.
- The function $\texttt{last_time}(\bar{\mathbf{e}},\texttt{t})$ returns the latest time $t_1 < t < now$ at which e occurred.
- The function $\texttt{next_time}(\bar{\mathbf{e}},\texttt{t})$ returns the most recent time $t_1 > t < now$ at which e occurred.
- The function $\texttt{extract}(\mathbf{e},\texttt{p})$, where p is a predicate, extracts the sub-stream $\mathbf{f}$ of stream $\mathbf{e}$ such that the predicate p is true at every occurrence of f. For instance, if the predicate is $\texttt{count}(\texttt{e},\texttt{t}) = \texttt{count}(\texttt{g},\texttt{t})$ the function $\texttt{extract}(\mathbf{e},\texttt{p})$ extracts the sub-stream $\mathbf{f}$ of stream $\mathbf{e}$ such that $\texttt{count}(\texttt{f},\texttt{t}) = \texttt{k}$ implies that there exist $0 \leq t_1 < t_2 < \ldots < t_{k-1} < t_k = t$, such that $\texttt{count}(\texttt{g},\texttt{t}_\texttt{i}) = \texttt{count}(\texttt{e},\texttt{t}_\texttt{i})$, for $i = 1,\ldots,k$.

Value and Function Streams. A variable v in the model is represented by a stream $\mathbf{v}$ in the language. In the language the event $ASSIGN(v) \in \mathcal{E}$ is a stream, denoted as $\mathbf{e_v}$. If $\mathbf{e_v}$ is a boolean stream, t_j and t_k, $t_j < t_k$, are the times of two successive occurrences of an event $e \in \mathcal{E}$, the streams $\mathbf{e_v}$ and $\mathbf{v}$ satisfy the properties:

$\mathbf{e_{v_t}} = false\ , t_j < t < t_k;\ \mathbf{v}_t = \mathbf{v}_{t_j}, t_j < t < t_k$

Sampling the stream $\mathbf{v}$ at times $t \in clock(\mathbf{e_v})$ is sufficient to know the history of the variable v.

A function stream is a sequence of functions that have been defined in the program. A function in the function stream is represented as a *tuple*, where a tuple is regarded as a *finite* stream. The tuple corresponding to the function $f(v_1,\ldots,v_n)$ defined in the program is $\langle \mathbf{f}, \mathbf{v_1},\ldots,\mathbf{v_n}\rangle$, where $\mathbf{v_i}$s are stream variables, and $\mathbf{f}$ is the function definition. The evaluation of a function $f(v_1,\ldots,v_n)$ at time t is an instantaneous transformation of the inputs $[\mathbf{v_i}/\mathbf{w_i}]_t$. The evaluation of a higher order function in Lucid is given by Paquet [9]: variables $\mathbf{v_i}$, $i = 1,\ldots,n$ are bound to *actual* streams $\mathbf{w_i}$, and the values at time t are extracted from the actuals $\mathbf{w_i}$ to evaluate the function. A stream variable may be bound to a multidimensional stream together with a chosen dimension of the stream. The evaluations of the function $f(v_1,\ldots,v_n)$ at different instances produce a stream $\mathbf{v_f}$ of values.

A predicate p is evaluated, as a function of its free variables, whenever a free variable in p gets a new value in the system.

The following functions manipulate value and function streams: The parameter $\mathbf{v}$ is a value stream, $\mathbf{F}$ is a tuple, and *now* is the current clock valuation.

1. The function $\texttt{last_assign}(\mathbf{v},\texttt{t})$ returns the latest time $t_1 < t < now$ at which the variable v changed its value.
2. The function $\texttt{next_assign}(\mathbf{v},\texttt{t})$ returns the most recent time $t_1 > t < now$ at which the variable v changes its value.

3. The function $\texttt{eval}(\mathbf{F}, \mathbf{w}, \texttt{t})$, evaluates the tuple **F** at time t by binding the stream variables in **F** to the streams in the tuple **w**, in the order specified. For each variable the latest assigned value (see 1 above) is used in evaluating the function. The current value of the function is stream $\mathbf{v}_{\mathbf{f}t}$.
4. The function $\texttt{eval}(\mathbf{F}, \mathbf{w}, \texttt{p})$, evaluates the tuple **F** whenever the predicate p on a subset of the variables $v_1, \ldots, v_n$ of the function $f(v_1, \ldots, v_n)$ becomes true. That is, if at time t the predicate p becomes true, the function $\texttt{eval}(\mathbf{F}, \mathbf{w}, \texttt{t})$ is invoked.

Streams for State Machine Models. We assume that a timed system is modelled by a variant of the *Timed Input Output Automaton*, which we refer to as an extended state machine (ESM). In the formal model we assume that one or more clocks may be used and constraints on state transitions are specified in *guard-action* paradigm. The guard g on a transition from state s_i to s_j is of the form $p \ \wedge \ tc$, where p, a predicate on the variable in state s_i, serves as a precondition for enabling the transition and tc is the time constraint predicate $lower \ \leq \ t \ < \ upper$. The action a is a predicate on the variable in the post state s_j.

For simplicity, in our discussion, we assume that each state has at most one active variable, namely the variable that may change its value in that state. The static aspects of a state machine specification are represented as follows:

1. State transitions are modelled as a 2-dimensional stream **tf**, which has dimensions $STATE_{from}$ and $STATE_{to}$ with state names as tags. The evaluation **tf** @ $[STATE_{from} : s_i, \ STATE_{to} : s_j]$ is the tuple $\langle tn, e \rangle$, where tn is the transition number and e is the event triggering the transition from s_i to s_j in this example.
2. A precondition is modelled as a 1-dimensional stream **pre**, with dimension *TRAN* and tag $\mathbb{N}$. The evaluation **pre** @ $[TRAN : k]$ is a tuple $\langle p_k, \mathbf{v} \rangle$ giving the predicate p_k for variable **v**.
3. A postcondition is modelled similarly, as a stream **post**, with dimension *TRAN* and tag $\mathbb{N}$. The evaluation **post** @ $[TRAN : k]$ is a tuple $\langle a_k, \mathbf{v} \rangle$, giving the postcondition for variable **v**.
4. A time constraint is modelled as a 1-dimensional stream **tc**, which has one dimension *TRAN* with tag $\mathbb{N}$. The evaluation **tc** @ $[TRAN : k]$ is a tuple of integers $\langle time_k, lower_k, upper_k \rangle$ corresponding to the constraint $lower \ \leq \ t \ < \ upper$ for transition k.

The dynamic behaviour of the state machine is the set of traces produced according to the state transition semantics. For each state s_i, let $\mathcal{E}(s_i)$ denote the set of events that are possible in s_i.

$$\frac{(s_i, v_i) \ \wedge \ e \ \in \mathcal{E}(s_i) \ \wedge \ p[(v_i)_t] \ \wedge \ tc(t)}{(s_i, v_i) \ \xrightarrow{e} \ (s_j, v_j) \ \wedge \ v'_j \ = \ a[(v_j)_t]}$$

We represent each trace of a machine by a stream of tuples $\langle s, v \rangle$ in the program, where $s \ \in \ \mathcal{S}$, a finite set of states in the formal model, and v is the active variable. An element of the trace is computed by applying the state transition semantics to the element that was generated at the previous step. If event e occurs at time t, and is admissible for the

current element in the trace, the transition happens instantaneously; if it is not admissible in this state, transition does not happen, but time is allowed to progress.

In general, if there are several state machines, the program will have a 2-dimensional stream $\mathbf{P}$, in which each "i-th row" is a state stream $\mathbf{M_i}$ corresponding the the state machine M_i in the model. At each instant t, $\mathbf{P}_t$ gives the stream in the tth column of the 2-dimensional stream, namely the stream showing the current status of all the machines in the system. The system state changes if there exists an event e that is admissible for the state in a tuple on the t-th column, otherwise time is allowed to progress. In the former case, e is admissible for some machine M_i, implying the specified constraints are satisfied by the state variable and clock valuations in the tuple $\mathbf{M_{i_t}}$. We calculate the function $progress(\mathbf{P}, t, e)$ to determine the next state tuple $\mathbf{M_{i}}_{t+1} = \langle s_i, \overline{s_{v_i}} \rangle_{t+1}$, and for all other rows, there is no change in time $t+1$:

$\mathbf{P_{t+1}} = \langle \mathbf{M_{1_t}}, \ldots, \mathbf{M_{(i-1)_t}}, \mathbf{M_{i_{t+1}}}, \mathbf{M_{(i+1)_t}}, \ldots, \mathbf{M_{n_t}} \rangle$

If the state machines $\mathbf{M_i}$ and $\mathbf{M_j}$ synchronise at time t on an event, then the state changes happen simultaneously in both machines, in rows i and j of the 2-dimensional stream while no other row in the stream will change. If no event occurs, time progresses in all clocks. The following program implements the function $progress(\mathbf{P}, t, e)$ with two dimensions *Time* and *Machine*:

Example 5. $progress.Time, Machine(P, t, e) =$
P@[*Time* : *t*] *fby.Machine*
if (*IsAdmissible*(*P @.Time #.Machine*, *e*, *t*))
then NextState(*P*, *e*, *t*)
else P @[*Time* : *t*] *#.Machine*;

Lucx expressions for the functions *IsAdmissible* and *NextState* are shown below. In GIPSY environment [7] Lucx programs may call external functions written in Java, the target language of our compiler. Hence, *IsAdmissible* and *NextState* functions will be implemented in Java, based on the following definitions. A tuple, being a finite stream, has a selector function which retrieves a specific component of the tuple for the given tag [10]. Below, we use the functions 1^{st}, 2^{nd}, and 3^{rd} to select respectively the first, second and third components of tuple streams.

$IsAdmissible(s_i, e, t) = eval((1^{st}.\mathbf{pre} \ @ \ [TRAN : k], \mathbf{v_{s_i}}, t) \wedge$
$(2^{nd}.\mathbf{tc} \ @[TRAN : k] \leq 1^{st}.\mathbf{tc} \ @ \ [TRAN : k]) \wedge (1^{st}.\mathbf{tc} \ @[TRAN : k] < 3^{rd}.\mathbf{tc} \ @[TRAN : k])$
where
$k = 1^{st}.\mathbf{tf} \ @ \ [STATE_{from} : s_i, STATE_{to} : s_j]$
end

Function $NextState(s_i, e, t)$ uses the function $IsAdmissible(s_i, e, t)$ and returns s_j and v_j such that $v_j = eval((1^{st}.\mathbf{post} \ @ \ [TRAN : k], \mathbf{v_{s_j}}, t)$

Context and Box Streams. Context and *Box* are first class objects, as stream, in the language. A stream $\mathbf{c}$ of contexts is a stream such that for $t \geq 1$, $\mathbf{c_t}$ is a context. The contexts in a stream need not have the same dimension sets. All context operators can be promoted to context streams. For instance, the expression $\mathbf{c_1} \oplus \mathbf{c_2}$ is the stream $\mathbf{c_3}$ such that $\mathbf{c_3}_t = \mathbf{c_1}_t \oplus \mathbf{c_2}_t, t \geq 1$.

A *Box* stream $\mathbf{B}$ is a stream such that for $t \geq 1$, $\mathbf{B_t}$ is a *Box*. The dimension sets of *Box*es in a stream need not be equal. Since a *Box* is a finite collection of contexts, a *Box*

stream may be viewed as a stream of tuples. All *Box* operators can be promoted to *Box* streams. For instance, the expression $\mathbf{b_1} \boxtimes \mathbf{b_2}$ is the stream $\mathbf{b_3}$ such that $\mathbf{b_3}_t = \mathbf{b_1}_t \boxtimes \mathbf{b_2}_t$, $t \geq 1$.

Stream modifier functions available in *Lucx*, combined with context and *Box* operators provide a rich mechanism to express time-varying situations in a real-time program. In particular, we explain now that *Box* streams are necessary to represent *clock regions* that arise when several clocks are used in the system model.

3.2 Multiple Clocks

We let C denote the set of clocks in the system. The clock evaluation functions for clocks satisfy the synchrony and monotonicity properties. Let us consider applications in which a clock c is never compared with a time constant greater than m. Then, the actual clock valuation, once it exceeds m, is of no consequence in deciding the allowed execution paths in the program. Hence every clock $c \in C$ has bounded support, $Intv(c)$.

For continuous time model an equivalence relation for clock valuations is given in [11]. We modify it as follows: $v \cong v'$ iff, for all $c_1,\ c_2 \in C,\ x \in \mathbb{R}^{\geq 0}$:

1. $Intv(c_1) = Intv(c_2)$
2. $\lfloor v(c_1)(x) \rfloor = \lfloor v'(c_1)(x) \rfloor$ and $(fract(v(c_1)(x)) = 0$ iff $fract(v'(c_1)(x)) = 0)$,
3. $fract(v(c_1)(x)) \leq fract(v(c_2)(x))$ iff $fract(v'(c_1)(x)) \leq fract(v'(c_2)(x))$.

If two clock valuations v and v' are equivalent, then $v(c)(x)[\delta] = v'(c)(x)[\delta]$ for any clock predicate δ.

A *clock region* is an equivalence class of clock valuations induced by equivalence relation $\cong$. We say that a clock region α satisfies a clock constraint δ iff every $v \in \alpha$ satisfies δ. Each region can be uniquely characterised by a (finite) set of clock constraints it satisfies. Each region can be represented by specifying

(1) for every clock c, one clock constraint from the set $\{v(c)(x) = m \mid m = 0, 1, \ldots, m_c\}$ $\cup \{m - 1 < v(c)(x) < m \mid m = 1, \ldots, m_c\} \cup \{v(c)(x) > m_c\}$, where m_c is the supremum of $Intv(c)$, and $x \in \mathbb{R}^{\geq 0}$
(2) for every pair of clocks c_1 and c_2 such that $m_1 - 1 < v(c_1)(x) < m_1$ and $m_2 - 1 < v(c_2)(x) < m_2$ appear in (1) for some m_1,m_2, whether $fract(v(c_1)(x))$ is less than, equal to, or greater than $fract(v(c_2)(y))$.

As an example, consider clocks c_1 and c_2 with $m_1 = 4$, and $m_2 = 6$. This gives rise to 59 clock regions, as shown in Figure 1. Each region is interpreted by the clock values according to the equivalence relation definition. For instance, (open) regions $\alpha 1$ and $\alpha 16$ are defined by the inequalities

$\alpha 1 : \quad 0 < v(c_1)(x) < 1, 0 < v(c_2)(y) < v(c_1)(x)$
$\alpha 16 : \quad 3 < v(c_1)(x) < 4, v(c1)(x) - 2 < v(c_2)(y) < 2$

A clock region α' is a *time-successor* of a clock region α iff for each $v \in \alpha$, there exists a positive $t \in \mathbb{R}$ such that $v + t \in \alpha'$. The time-successors of a clock region α are all the clock regions that will be visited by a clock valuation $v \in \alpha$ as time progresses. The time-successors of a region α can be derived by moving along a line drawn from some point in α in the diagonally upwards direction(parallel to the line $x = y$). For

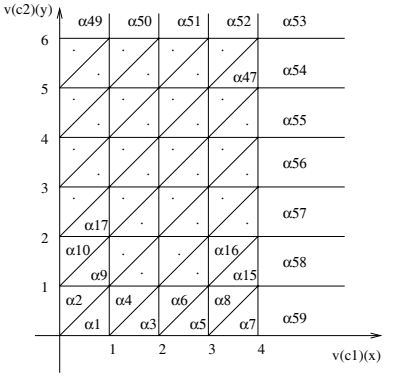


Fig. 1. Clock Regions

instance, in Figure 1, the successors of region $\alpha 1$ are : $\alpha 4$, $\alpha 11$, $\alpha 14$, $\alpha 21$, $\alpha 24$, $\alpha 31$, $\alpha 53$, $\alpha 54$, $\alpha 55$, $\alpha 56$.

The clock regions corresponding to a set of clocks is represented as a finite stream of *Boxes*. Every *Box* in the stream corresponds to one region. Each *Box* is defined by the dimension set $\Delta = \{c_1, \ldots, c_k\}$, and a constraint on the clock valuations. For example, $Box[\Delta \mid p_1]$, where $\Delta = \{c_1, c_2\}$ and $p_1 = 0 < v(c_1)(x) < 1, 0 < v(c_2)(y) < v(c_1)(x)$ refers to the region α_1. The tag sets for clocks are reals. For discrete time modelled by multiple clocks the tag sets are integers, and regions become lattice points, vertices of convex regions.

4 Railroad Crossing Problem

In this section we provide a specification of the generalized railroad crossing problem, an example studied in real-time systems community [4]. This is the first step in our efforts to experiment with the language for specification, programming, and verification of programs for real-time reactive systems.

4.1 Problem Statement

Several trains cross a gate controlled by a monitor. Trains may be running on several tracks, and hence cross the gate simultaneously. When a train approaches the gate, it sends a message to the corresponding controller, which then commands the gate to close. When the last train crossing a gate leaves the crossing, the controller commands the gate to open. The safe operation of the controller depends on the satisfaction of certain timing constraints, so that the gate is closed before a train enters the crossing, and the gate is opened after the last train leaves the crossing.

1. [C1] A train enters the crossing within an interval of 2 to 4 time units after having indicated its presence to the controller.
2. [C2] The train informs the controller that it is leaving the crossing within 6 time units of sending the approaching message.

3. [C3] The controller instructs the gate to close within 1 time unit of receiving an approaching message from the first train entering the crossing, and starts monitoring the gate. The controller continues to monitor the closed gate if it receives an approaching message from another train.
4. [C4] The controller instructs the gate to open within 1 time unit of receiving a message from the last train to leave the crossing.
5. [C5] The gate must close within 1 time unit of receiving instructions from the controller.
6. [C6] The gate must open within an interval of 1 to 2 time units of receiving instructions from the controller.

4.2 Events and Streams for Problem Specification

In [6], a formal design of the railroad problem is given. It uses ESMs to formalize the behavior of train-gate-controller objects. The formal object-oriented model thus obtained is linked with PVS to formally verify the safety property in the modelled system. We use the approach outlined in Section 3 to formally represent the above design in Lucx and prove that our design satisfies the safety property. As we show in Section 4.4 below, the safety property can be written purely in terms of the times of occurrences of observable events in the system. So we skip the details of state streams, and discuss below the specification of event streams and their constraints.

The events *Lower* and *Raise* are sent by the controller to the gate. The events *Near*? and *Exit*? are received by the controller from a train. The gate closes using the event *Down* and opens using the event *Up*. The events *In* and *Out* are used by trains respectively to indicate that they are inside the crossing and outside the crossing respectively. A *period* is the interval of time between two successive instants when the gate opens. Hence the duration of $k-th$ period is $\overline{Up_{k+1}} - \overline{Up_k}$. Within a period, several trains may come, and hence the events *Near*, *In*, *Out*, and *Exit* may occur several times. However, within a period, the controller events and gate events occur just once. We represent the events by the following streams:

1. The streams *Lower* and *Raise* are shared representations for the synchronous occurrences of *Lower*!, *Lower*?, and *Raise*!, *Raise*?. Thus, $\overline{Lower_k}$ and $\overline{Raise_k}$ give the times of occurrences of the events Lower and Raise in the $k-th$ period, because in each period they happen just once.
2. The streams *Down* and *Up* represent the events Down and Up. That is, in the $k-th$ period, the events Down and Up occur at times $\overline{Down_k}$, and $\overline{Up_k}$.
3. We use a 3-dimensional stream σ to represent the events from trains, with the convention that the events *Near*, *In*, *Out*, and *Exit* are denoted by 1,2,3, and 4 respectively. The justification is that for each train in the $k-th$ period, these events are linearly ordered: $TIME(Near)(k) < TIME(In)(k) < TIME(Out)(k) < TIME(Exit)(k)$. We can also represent the events for a train by using *tuples*, which we avoid for clarity of presentation. The stream σ has three significant dimensions, say *TRAIN*, *EVE*, and *PER* with tags $\mathbb{N}$ for *TRAIN* and *PER* and the set $\{1,2,3,4\}$ for *EVE*. The evaluation σ @ $[TRAIN : i, EVE : j, PER : k]$, denoted σ_{ijk}, is the time at which the event j occurred in $i-th$ train in the $k-th$ period. For instance, σ_{243} gives the time

at which the event *Exit* occurred in the second train in the $3rd - period$. Notice that i increases with the arrival of a new train in the system. The 1-dimensional stream $\sigma@[TRAIN : i, EVE : 1]$ gives the times of arrivals of the i-th train in all periods.

4.3 Specification in the Language

For every period, events used by the gate are linearly ordered:

$k \in \mathbb{N}, \overline{Lower_k} < \overline{Down_k} < \overline{Raise_k} < \overline{Up_k}$

Within a period k, the events of each train are linearly ordered: $\sigma_{i1k} < \sigma i2k < \sigma i3k < \sigma_{i4k}$. For every period k, the time constraints $C1, \ldots, C6$ can be formally specified in Lucx as follows:

C1 $\sigma\ @[EVE : 1, PER : k] + 2 < \sigma\ @[EVE : 2, PER : k] < \sigma\ @[EVE : 1, PER : k] + 4$
C2 $\sigma\ @\ [EVE : 1, PER : k] < \sigma\ @\ [EVE : 4, PER : k] < \sigma\ @\ [EVE : 1, PER : k] + 6$
C3 $\sigma_{11k} < \overline{Lower_k} < \sigma_{11k} + 1$
C4 $last_time(\sigma\ @[EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Raise_k} < last_time(\sigma\ @[EVE : 4, PER : k], \overline{Lower_{k+1}}) + 1$
C5 $\overline{Lower_k} < \overline{Down_k} < \overline{Lower_k} + 1$
C6 $\overline{Raise_k} + 1 < \overline{Up_k} < \overline{Raise_k} + 2$

4.4 Verification of Safety Property

Informally, a program that is consistent with the above requirements is *safe*, if in every period k the following property is satisfied by the program: *The gate closes before any train is in the crossing and opens only after the last train in the period has left the crossing.* Using our specification formalism, we formally rewrite the safety property as follows:

$$\overline{Down_k} < \sigma_{12k} < last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Up_k} \qquad (S)$$

To prove the safety property, we use $C1 \ldots C6$, and $A1$ below as axioms and show that the predicate (S) is a consequence of these axioms:

$$\sigma_{14k} \leq last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}}) \qquad A1$$

The proof steps are as follows for any period k - the axioms used in deriving a step are shown at the end of each step:

1. $\overline{Down_k} < \overline{Lower_k} + 1$ [C5]
2. $\overline{Lower_k} + 1 < \sigma_{11k} + 2$ [C3]
3. $\sigma_{11k} + 2 < \sigma_{12k}$ [C1]
4. $\overline{Down_k} < \sigma_{12k}$ [Steps 1,2,3]
5. $\sigma_{12k} < \sigma_{14k}$ [C1,C2]
6. $\sigma_{14k} \leq last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}})$ [A1]
7. $\sigma_{12k} < last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}})$ [Steps 5,6]
8. $last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Raise_k}$ [C4]
9. $last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}}) + 1 < \overline{Raise_k} + 1$ [Step 8]
10. $\overline{Raise_k} + 1 < \overline{Up_k}$ [C6]
11. $last_time(\sigma\ @\ [EVE : 4, PER : k], \overline{Lower_{k+1}}) < \overline{Up_k}$ [Steps 9,10]
12. $\overline{Down_k} < \sigma_{12k} < last_time(\sigma\ @[EVE : 4, PER : k], \overline{Lower_{k+1}} < \overline{Up_k}$ [Steps 4,7,11]

We conclude that any formal model in which the ESMs for train, gate, and controller satisfy the axioms $C1, \ldots, C6, A1$ satisfy the predicate (S).

5 Concluding Remarks

Any implementation of the verified design of a real-time reactive system must faithfully conform to the design. Because of the semantic gap between the language used for the formal design and the programming language that implements the design, it is hard to demonstrate the faithfulness of the implementation to the verified design. Lucid programs, being declarative, can be reasoned about. GIPSY [7] is an implementation platform for Lucid. These features convinced us that in Lucid a semantic continuity exists between a high level program, which is a specification, and its implementation. *Lucx*, being a conservative extension of Lucid, would retain this continuity between a real-time system specification in it and its implementation in GIPSY.

The significant feature that we have introduced in Lucid is the notion of context as a first class object. This notion was originally introduced by MaCarthy, and used by Guha [3] for enriching natural language expressions in AI. We are motivated by this work. However, our notion of context differs significantly from MaCarthy's. In our study context is both *finite* and *concrete*. Guha uses contexts as *infinite*, *rich*, and *generalized* objects. Our goal is to be able to manipulate contexts dynamically and evaluate programming language expressions in different contexts. This contrasts with the work of Guha in which the real meaning of natural language expressions are captured by evaluating them in very rich contexts. Not all contexts studied by Guha can be dealt within our language. However, every context that we can define in *Lucx* is indeed a context in Guha's sense, but restricted to well-formed *Lucx* expressions.

Most of the physical systems exhibit hybrid behavior and behave according to certain scientific principles. The scientific programming constructs introduced by Paquet [8] and the results shown in this paper add the expressive power needed to formally specify as well as program such systems. We have shown a *Lucx* specification for the formal design of the railroad crossing problem given in [6], and have verified its safety property. The verification approach depends only on time-constrained events and does not explicitly require the state information. Our ongoing research includes developing a verification approach based on intensional logic, the basis of Lucid, and extending GIPSY architecture for implementing *Lucx* programs.

References

1. Vasu S. Alagar, Joey Paquet, Kaiyu Wan. *Intensional Programming for Agent Communication* Proceedings of DALT'04, New York, July 2004.(post proceedings to be published by LNCS, Springer-Verlog)
2. P.Caspi, D,Pilaud, N.Halbwachs, J.A.Plaice. *LUSTRE: A declarative language for programming synchronous systems.* P.O.P.L. 1987
3. R. V. Guha. *Contexts: A Formalization and Some Applications.* Ph.d thesis, Stanford University, February 10,1995.
4. C.Heitmeyer and N.Lynch. *The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems* In Proceedings of the 15th IEEE Real-time Systems Symposium, RTSS'94, page 120-131, San Juan, Puerto Rico, Dec 1994.
5. H. Marchand, E.Rutten, M. Le Borgne, M. Samman. *Formal verification of programs specified with signal: application to a power transformer station controller.* Science of Computer Programming 41 (2001) 85-104.

6. D. Muthiayen. *Real-Time Reactive System Development - A Formal Approach Based on UML and PVS.* Phd. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, January 2000
7. J. Paquet, P. Kropf. *The GIPSY Architecture*. DCW 2000, 144-153.
8. Joey Paquet. *Intensional Scientific Programming* Ph.D. Thesis, Département d'Informatique, Universite Laval, Quebec, Canada, 1999
9. Joey Paquet and John Plaice. *Dimensions and functions as values.* Proceedings of the Eleventh International Symposium on Lucid and Intensional Programming, Sun Microsystems, Palo Alto, California, USA, may 1998.
10. John Plaice. *RLucid, a general real-time dataflow language.* Formal Techniques in Real-Time and Fault-Tolerant Systems, pages 363-374, Berlin, 1992.
11. J. Springintveld, F. Vaandrager, P. Dargenio. *Testing Timed Automata.* Theoretical Computer Science, vol. 254, pp.225-257,2001.
12. W.W.Wadge, E.A.Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985

Revision Programs with Explicit Negation

Yisong Wang[1] and Mingyi Zhang[2]

[1] Department of Computer Science, Guizhou University,
Guiyang, Guizhou, P. R. China, 550025
ys_wang168@yahoo.com.cn
[2] Guizhou Academy of Sciences, 40 East Yanan Road,
Guiyang, Guizhou, P. R. China, 550001

Abstract. Revision program [9], which describes the transformation from an initial database to a revised database, is a formalism for specifying revisions on database. In order to deal with the incompleteness of knowledge, the paper provides an extension for both database and revision program by introducing explicit negation. Motivated by "$\sim l \leftarrow \neg l$" [2], we propose an extended P-justified revision semantics for the extension, which is more general than P-justified revision semantics, where we argue that the latest knowledge is preferred to the old one when it conflicts with the old. This extension is not trivial by simply regarding negative literals as positive in annotated revision programs [10]. In the end, we establish the one-to-one correspondence between the extended P-justified revisions of extended revision programs and answer sets of the corresponding extended logic programs.

1 Introduction

Revision program is a collection of the revision rules of the forms,

$$in(a) \leftarrow in(a_1), ..., in(a_n), out(b_1), ..., out(b_m) \text{ or} \tag{1}$$

$$out(a) \leftarrow in(a_1), ..., in(a_n), out(b_1), ..., out(b_m) \ . \tag{2}$$

It was firstly introduced by Marek and Truszczński in 1994 as a formalism to describe and study the process of database updates. The intuitive meaning of revision rules of form (1)(resp, (2)) is that a should be in (resp, not in) the database D if $a_1, ..., a_n$ are all in database D and all $b_1, ..., b_m$ are not in D. Informally, a P-justified revised database R of an initial database I is a set of atoms, which satisfies the revision program P in the sense of model theory and guarantees the minimal changes from I [9, 12].

As stated by John McCarthy that it's permitted to cross railway tracks only if there is no approaching train being guaranteed, it is necessary to introduce explicit negation into knowledge representation for its incompleteness [5]. The next example confirms the case.

Example 1.1. Suppose there are four persons John, Adams, Rose and Jack in one music interest group and, anyone in the group, who did not attend concert,

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 403–414, 2005.
© Springer-Verlag Berlin Heidelberg 2005

will be fined ten thousands dollars. After one concert, from the secretary of the group, we know that John attended the concert, Adams did not attend the concert. That's all we know for the concert about them. A few weeks later, we know that some new information from the head of the group: John didn't attend the concert and Rose attended the concert unless John abnormally attended. In the sense of [9], we have the following representation (Actually, we are not quite sure whether or not they are proper presentations with the last two rules of P, but it's the only way to do so.):

$$I = \{Attend(John)\}, P = \left\{ \begin{array}{l} out(Attend(John)) \leftarrow \\ in(Attend(Rose)) \leftarrow out(Attend(John)) \\ in(Fine(X)) \leftarrow out(Attend(X)) \end{array} \right\}$$

So we have R=*{Attend(Rose), Fine(John), Fine(Adams), Fine(Jack)}* as the unique P-justified revision of I, from which we know that anybody will be fined except Rose. Obviously, it's unacceptable to Jack since there is no evidence for him to be fined at all.

We argue that it is reasonable to eliminate old knowledge if the latest knowledge conflicts with it. Inspired by "$\sim l \leftarrow \neg l$" in [2], our extension is constructed by the introduction of a new constraint C_I to the initial knowledge base *I, i.e.* $\{out(l) \leftarrow in(\neg l) | l \in I\}$. The intended meaning is that l (l is known to be true) should be discarded if its explicit negation was assured. For the extension, we propose an extended P-justified revision semantics, which keeps the intended meaning of P-justified revision. Moreover, we prove that extended revision programs and extended logic programs can be embedded into each other in the sense of their 1-1 correspondent semantics – the extended P-justified revisions and answer sets respectively.

2 Extended Revision Programs

In this section, we present our extension of revision programs and databases. From the perspective of descriptive semantics, it is not so important considering variables in the language since they can be removed by instantiation. For clarity, we just focus on its propositional case.

Let U be a denumerable set of *atoms* (a universe). A *classical literal* (or *literal*) is an atom a or its negation (classical negation) $\neg a$. a and $\neg a$ are *complement* with each other. As in first order logic, we admit $\neg\neg a = a$. By Lit we mean the set of all literals, i.e. $Lit = U \cup \neg U$, where $\neg U = \{\neg a | a \in U\}$.

An expression of the forms of *in(l)* or *out(l)* is called *extended revision literal* (or *revision literal*), where $l \in Lit$. *in(l)* (resp. *out(l)*) is called *extended in-literal* (resp. *extended out-literal*). By $RLit$ we mean the set of all revision literals, namely, $RLit = \{in(l) | l \in Lit\} \cup \{out(l) | l \in Lit\}$. A set of extended revision literals L is *incoherent* if there is at least one l such that $\{in(l), out(l)\} \subseteq L$ or $\{in(l), in(\neg l)\} \subseteq L$. Otherwise L is *coherent*. Let L be a set of extended revision literals, we denote $L^+ = \{l | in(l) \in L\}$, $L^- = \{l | out(l) \in L\}$. Let R be a set of literals, we define $R^c = \{in(l) | l \in R\} \cup \{out(l) | l \notin R\}$.

Definition 2.1. *An extended database (knowledge base) S is a set of literals. We denote $S^t = \{a | a \in S\}$, $S^f = \{a | \neg a \in S\}$. S is consistent if there is not any pair of complements in S, i.e. $S^t \cap S^f = \emptyset$.*

By the above definition, any atom, saying a, should have an unique state with respect to a consistent knowledge base S. a is $true$ (resp. $false$, $unknown$) if $a \in S^t$ (resp. $a \in S^f$, otherwise).

Definition 2.2. *An extended revision program is a collection of the extended revision rules of the forms,*

$$in(l) \leftarrow in(l_1), ..., in(l_m), out(l_{m+1}), ..., out(l_n) \ \ or \tag{3}$$

$$out(l) \leftarrow in(l_1), ..., in(l_m), out(l_{m+1}), ..., out(l_n) \tag{4}$$

where l, $l_i (1 \leq i \leq n)$ are literals. *in(l)* (resp. *out(l)*) is the head of the extended revision rule of (3) (resp. (4)). $\{in(l_1), ..., in(l_m), out(l_{m+1}), ..., out(l_n)\}$ is their body. The head and body of an extended revision rule r are denoted respectively by *head(r)* and *body(r)*. The set of the heads of all rules in revision program P is denoted by *head(P)*. If the head of a rule is of the form *in(l)*, the rule is called an *in-rule*, otherwise, it is called an *out-rule*. The set of all extended revision literals appearing in a program P is denoted by *var(P)*.

Given an initial knowledge base I and a revision program P, we assume that, without further mention, the language is determined by I and P, i.e, the U is the set of all atoms which occurs in I or P, or whose negation does. That is, $U = \{a | a \in I \text{ or } \neg a \in I\} \cup \{a | in(a) \in var(P) \text{ or } in(\neg a) \in var(P) \text{ or } out(a) \in var(P) \text{ or } out(\neg a) \in var(P)\}$.

Definition 2.3. *Let S be a consistent set of literals, the satisfiability of extended revision literal is defined as follows,*

- *S satisfies an extended revision literal in(l) if $l \in S$. S satisfies out(l) if $l \notin S$.*
- *S satisfies an extended revision rule r if either S satisfies head(r) or S doesn't satisfy at least one extended revision literal in body(r). Namely, if S satisfies body(r) then S satisfies head(r).*
- *S satisfies an extended revision program P if S satisfies all of the rules of P. S is a model of P if S satisfies P.*

We write $S \models \alpha$, $S \models r$, $S \models P$ to denote that S satisfies extended revision literal α, extended revision rule r and extended revision program P. Given an extended revision program P, we denote the least model of P by $LM(P)$ when P is treated as a Horn program built of independent propositional atoms of forms *in(l)* and *out(l)*. *LM(P)* states the necessary revision to knowledge base forced by P, and is called the *necessary change* of P, denoted by *NC(P)*. For a knowledge base I, the revision result by a coherent set of revision literals L is defined by

$$I \oplus L = (I \backslash L^-) \cup L^+.$$

The following two lemmas 2.1 and 2.2 generalize the basic properties of the model and the operator $\oplus$ in [9].

Lemma 2.1. *Let L be a set of extended revision literals and B be a consistent knowledge base.*

1. *If $B \models L$, then L is coherent and $B \oplus L = B$.*
2. *Let L be a coherent set of extended revision literals. If $L \subseteq L'$ and $B \oplus L \models L'$, then $B \oplus L = B \oplus L'$.*
3. *Let L be coherent. If $\alpha \in L$, then $B \oplus L \models \alpha$. If $B \oplus L \models \alpha$ and $B \not\models \alpha$, then $\alpha \in L$.* □

For two knowledge bases I and R, we define the *inertial set* $\mathcal{I}(I, R)$ as:

$$\mathcal{I}(I, R) = \{in(l) | l \in I \cap R\} \cup \{out(l) | l \notin I \cup R\}.$$

The intended meaning of $\mathcal{I}(I, R)$ is, being the same with the original meaning of inertial set in [9], the collection of all extended revision literals describing the elements that do not change their status, in transition from I to R. Inertia set $\mathcal{I}(I, R)$ has the following important property.

Lemma 2.2. *Let I, I' and R be consistent knowledge bases. Let L be a set of extended revision literals and α be an extended revision literal.*

1. *$\alpha \in \mathcal{I}(I, R)$ if and only if $I \models \alpha$ and $R \models \alpha$.*
2. *$\mathcal{I}(I, R) \subseteq \mathcal{I}(I', R)$ if and only if $R \div I' \subseteq R \div I$.*
3. *If $\mathcal{I}(I, R) \subseteq \mathcal{I}(I', R)$, L is coherent and $R = I \oplus L$, then $R = I' \oplus L$.* □

$\div$ is the symmetrical difference operator of two sets. Part (1) of the lemma expresses a basic intuition behind the inertia set. It consists of those revision literals that are satisfied by both I and R. Part (2) shows that the larger the inertia set, the "closer" the two knowledge bases are (and conversely). Finally, part (3) states that if R is obtained by revising I by L and if I' is "closer" to R than I, then revising I' by L also results in R. We now present our revision constraint before define the semantics of extended justified revision.

Definition 2.4. *Let I be a consistent knowledge base. The revision constraints of I is the set of extended revision rules,*

$$C_I = \{out(l) \leftarrow in(\neg l) | l \in I\}.$$

The notion of revision constraints is quite intuitive. Firstly, it says that $in(\neg l)$ is stronger (or stricter) than $out(l)$. That is — if $\neg l$ is known to be *true* then we should have "l isn't known to be *true*" hold. Secondly, the revision constraint guarantees the consistency of the extended P-justified revision (see definition 2.5) whenever an item reverses its status from *true* to $false$ or from $false$ to *true*. At the same time, it also means that the new knowledge is preferable to the old one. In other words, for any $l \in I$, l should be removed from the justified revision of I if the complement of l has been proved to be hold.

Definition 2.5. *Let I be a consistent knowledge base and P be an extended revision program. A consistent knowledge base R is an extended P-justified revision of I (P-justified revision), if*

- $NC(P_{I,R} \cup C_I)$ *is coherent, and*
- $R = I \oplus NC(P_{I,R} \cup C_I)$

where $P_{I,R}$ is obtained from P by removing all extended revision literals in $\mathcal{I}(I, R)$ from the body of every rules in P.

By far, we present the extended justified revision semantics for our extension, which is almost the same with the original except for the C_I in the necessary change. Frankly, we confess that the same conclusion can be obtained from the original justified revision by taking the negative literal as positive one and being added the same constraint C_I. However, the essence of the justified revision is different in the view of the incompleteness of knowledge representation.

By the end of this section, we demonstrate a few examples to show our extension for knowledge base, revision program and compare them with some other extensions, starting with our version of the Example 1.1 in introduction section. In the sense of our extended revision, we have the following representation:

Example 2.1. (**continue of example 1.1**)
$I = \{Attend(John), \neg Attend(Adams)\}$,

$$P = \left\{ \begin{array}{l} in(\neg Attend(John)) \leftarrow \\ in(Attend(Rose)) \leftarrow out(Attend(John)) \\ in(Fine(X) \leftarrow in(\neg Attend(X)) \end{array} \right\}.$$

So we have R=*{Attend(Rose), ¬Attend(John), ¬Attend(Adams), Fine(John), Fine(Adams)}* as the unique extended P-justified revision of I. According to R, we know that only John and Adams should be fined.

By the next example, I will show our extension are not trivial by regarding negative atoms as a positive one in the other extensions of revision programs, even in the annotated revision programs.

Example 2.2. Let U=*{a, b, c}*, I=*{a, ¬b}* and R=*{c, ¬a, ¬b}*. The extended revision program P is as below:

$$P = \left\{ \begin{array}{l} in(\neg a) \leftarrow out(b) \\ out(\neg b) \leftarrow out(\neg a) \\ in(c) \leftarrow \end{array} \right\}, \qquad C_I = \left\{ \begin{array}{l} out(a) \leftarrow in(\neg a) \\ out(\neg b) \leftarrow in(b) \end{array} \right\}.$$

So we have C_I as above, and $\mathcal{I}(I, R) = \{out(b), out(\neg c), in(\neg b)\}$;

$$P_{I,R} = \left\{ \begin{array}{l} in(\neg a) \leftarrow \\ out(\neg b) \leftarrow out(\neg a) \\ in(c) \leftarrow \end{array} \right\}, \; NC(P_{I,R} \cup C_I) = \{in(\neg a), in(c), out(a)\},$$

$I \oplus NC(P_{I,R} \cup C_I) = I \backslash \{a\} \cup \{\neg a, c\} = \{c, \neg a, \neg b\} = R$. So R is an extended P-justified revision of I.

It's not true that annotated revision program can obtain the same result by renaming the classical negative literal $\neg a$ to a new positive one, saying a', and then apply the annotated transform [10]. As to the above program, the corresponding annotated revision program P^a is:

$$\left\{\begin{array}{l} in(a') : t \leftarrow out(b) : t \\ out(b') : t \leftarrow out(a') : t \\ in(c) : t \leftarrow \end{array}\right\}.$$

Then we have $I^v(a) = \langle t, f\rangle, I^v(a') = \langle f, t\rangle, I^v(b) = \langle f, t\rangle, I^v(b') = \langle t, f\rangle, I^v(c) = \langle f, t\rangle, I^v(c') = \langle f, t\rangle$. $R^v(a) = \langle f, t\rangle, R^v(a') = \langle t, f\rangle, R^v(b) = \langle f, t\rangle, R^v(b') = \langle t, f\rangle, R^v(c) = \langle t, f\rangle, R^v(c') = \langle f, t\rangle$. And then $P^a_{R^v}|I^v$ is

$$\left\{\begin{array}{l} in(a') : t \leftarrow out(b) : \bot \\ in(c) : t \leftarrow \end{array}\right\}.$$

So the necessary change is $C^v(a) = \langle f, t\rangle, C^v(a') = \langle t, f\rangle, C^v(b) = \langle f, t\rangle, C^v(b') = \langle f, t\rangle, C^v(c) = \langle t, f\rangle, C^v(c') = \langle f, t\rangle$. It's easy to see $-C = C$. So $R' = (I \otimes -C) \oplus C = C \neq R$ because of $C^v(b') = \langle f, t\rangle$ but $R^v(b') = \langle t, f\rangle$. Consequently, R is not an extended P-justified revision of I by taking them as annotated.

In fact, it is doubtful to transform the initial database I to B_I. The latter treats *classical negation* and *negation by default* equivalently. The incompleteness of I was discarded by the definition of $B^v_I(a) = \langle t, f\rangle$ if $a \in B_I$ and $\langle f, t\rangle$ otherwise.

Pivkina had presented an extension called nested expression in revision programming [12]. It cannot deal with the above classical negation in revision literals yet. Let the two consistent databases *I, R* and the revision program P are the same with the above example. We just regard the classical literals (a and $\neg a$) as some new positive literals in mind which are irrelevant to each other. Since there is no unary operator *in* and *out* in P at all, the reduction of P w.r.t *(I,R)* is P. Clearly, we have L=$\{in(\neg a), in(c)\}$ is a minimal set of literals that is closed under P (it is the necessary change of $P_{I,R}$). However, $R' = I \oplus L = \{a, \neg a, c, \neg b\} \neq R$. Consequently, R is not an extended P-justified revision of I under nested expression.

There is one more example which is an extended logic program version of Lifschitz [5] about college X uses the rules for awarding scholarships to its students. We implement it in our extended revision program.

Example 2.3. The corresponding extended revision program P is

$$\left\{\begin{array}{l} in(Eligible(x)) \leftarrow in(HighGPA(x)) \\ in(Eligible(x)) \leftarrow in(Minority(x)), in(FairGPA(x)) \\ in(\neg Eligible(x)) \leftarrow in(\neg FairGPA(x)) \\ in(Interview(x)) \leftarrow out(Eligible(x)), out(\neg Eligible(x)) \end{array}\right\}$$

Let the initial knowledge base about Ann is I=$\{$*FairGPA(Ann),* $\neg$*HighGPA(Ann)*$\}$. It is easy to see that we have the unique extended P-justified revision R=$\{$*FairGPA(Ann),* $\neg HighGPA(Ann)$*, Interview(Ann)*$\}$ of *I*.

3 Basic Properties

In the section, some basic results of our extension are presented.

When the classical literals in extended revision program P are restricted within atoms, P will be reduced to the revision program of [9]; moreover, if the initial extended databases are also restricted within the set of atoms, the extended P-justified revision will be the same with P-justified revision, since C_I has not any contribution to $NC(P_{I,R} \cup C_I)$ (because negative literals don't exist any longer in P). We will formally show this by the following theorem.

Theorem 3.1. *Let I and R be two knowledge bases containing only atoms. P be an extended revision program without negative literals in its extended revision literals. We have that R is an extended P-justified revision of I if and only if R is a P-justified revision of I.*

Proof. It is easy to see $NC(P_{I,R} \cup C_I) = NC(P_{I,R})$; I and R are consistent. Consequently, R is an extended P-justified revision of I if and only if R is the P-justified revision of I. □

As we have stated that extended P-justified revision can be obtained from P-justified revision by adding the same constraints to initial database and regarding negative literal as positive one implicitly. We formally present it by the next theorem.

Theorem 3.2. *Let I and R be two consistent knowledge bases. P be an extended revision program. Then R is an extended P-justified revision of I if and only if R is a $(P \cup C_I)$-justified revision of I by regarding all the negative literals as some positive one implicitly.*

Proof. Since I is consistent, we have $in(\neg l) \notin \mathcal{I}(I, R)$ for any "$out(l) \leftarrow in(\neg l)$" in C_I. So $C_{I_{I,R}} = C_I$. $NC(P_{I,R} \cup C_I) = NC(P_{I,R} \cup C_{I_{I,R}}) = NC((P \cup C_I)_{I,R})$. □

The similarity between our extended P-justified revision and P-justified revision in [9], deduces to some very similar properties. The proofs of the following three theorems are respectively similar to Theorem 2.12, Theorem 2.13 and Theorem 2.14 of [9]. So we omit their proofs here. Firstly, we will show the alternative definition for extended P-justified revision by GL-reduction.

Theorem 3.3. *Let P be an extended revision program and let I and R be two consistent knowledge bases. The following two conditions are equivalent:*

(R1) $NC(P_{I,R} \cup C_I)$ is coherent and $R = I \oplus NC(P_{I,R} \cup C_I)$
(R2) $NC(P_R|I \cup C_I)$ is coherent and $R = I \oplus NC(P_R|I \cup C_I)$

where $P_R|I$ is obtained from P by (1) Removing all the rules whose body is not satisfied by R, denoted by P_R; (2) Removing all revision literals from the body of rules in P_R which are satisfied by I. □

From the alternative definition of extended justified revision, we have the following two theorems, which play important roles in proving the following properties of the our extension. At first, we present the notation P^u. Let P be a propositional Horn program. We define

$$P^u = \{c | c \in P \wedge body(c) \subseteq LM(P)\}.$$

Intuitively, P^u consists of all those rules in P which "fire" (are used) during the construction of the least model of P. In particular,

$$head(P^u) = LM(P).$$

Theorem 3.4. *Let P be an extended revision program and a consistent knowledge base R be an extended P-justified revision of a consistent knowledge base I. Then, $(P_{I,R})^u = P_R|I$ and*

$$\begin{aligned} &NC(P_{I,R} \cup C_I) \\ &= NC(P_R|I \cup C_I) \\ &= head(P_R) \cup \{out(l) | in(\neg l) \in head(P_R) \wedge (l \in I)\}. \end{aligned}$$

□

This theorem implies that the condition of our revision constraint C_I can be enforced further by $in(l) \in Head(P_R)$.

Theorem 3.5. *Let P be an extended revision program and let I and R be two consistent knowledge bases. The following conditions are equivalent:*

1. *A consistent knowledge base R is an extended P-justified revision of a consistent knowledge base I.*
2. $NC(P \cup C_I \cup \{\alpha \leftarrow | \alpha \in \mathcal{I}(I, R)\}) = R^c$.
3. $NC(P_{I,R} \cup C_I) \cup \mathcal{I}(I, R) = R^c$.

□

As for the relationship between this extended revision programs and extended logic programs, we have a strict relation between computation of consistent answer sets of extended logic programs and computation of the extended P-justified revision of extended revision programs, that is a 1-1 correspondence.

Definition 3.1. *The translation of the extended revision program P and the consistent extended initial database I into an extended logic program is defined as the logic program $\mathcal{P}(P, I) = P \cup C_I \cup P_N$ over a language $\mathcal{K}$, $\mathcal{K}$ is a propositional language with the set of propositional letters consisting of $\{in(l) | l \in Lit\} \cup \{out(l) | l \in Lit\}$. P_N is the inertia rules of I: If l is initially in (resp. out) then after revision it remains in (resp. out) unless it is forced out (resp. in) explicitly or implicitly. Namely,*

$$P_N = \{in(l) \leftarrow\sim out(l), \sim in(\neg l) | l \in I\} \cup \{out(l) \leftarrow\sim in(l) | l \notin I\}$$

where $\sim$ is the negation by default operator. We have an important proposition for the inertia rules P_N.

Proposition 3.1. *Let I and R be two consistent knowledge bases. Then $LM(P_N^{R^c}) = \mathcal{I}(I, R)$.*

Proof. "⇒" $\forall \alpha \in LHS$, we have "$\alpha \leftarrow$"$\in P_N^{R^c}$. (1) If $\alpha = in(l)$, that is $in(l) \in LHS$ if and only if $l \in I$, $in(\neg l) \notin R^c$ and $out(l) \notin R^c$. If $in(l) \notin R^c$ then $out(l) \in R^c$. It conflicts with $out(l) \notin R^c$. So $in(l) \in R^c$, $l \in R$. Consequently, $in(l) \in \mathcal{I}(I, R)$. (2) If $\alpha = out(l)$, that is "$out(l) \leftarrow$"$\in LHS$ if and only if $l \notin I$ and $in(l) \notin R^c$. So $l \notin R$, $l \notin I \cup R$. Consequently, $out(l) \in RHS$. The reverse is very similar. □

Note that, we take $\mathcal{P}(P,I)$ as a logic program by regarding the revision literals in $\mathcal{P}(P,I)$ as the literals of logic program. A set of such literals M, which is a set of extended revision literals, is an answer set of $\mathcal{P}(P,I)$ if

$$LM(\mathcal{P}(P, I)^M) = M.$$

M is a coherent answer set if M is a coherent set of extended revision literals and $M = LM(\mathcal{P}(P, I)^M)$. We have the following important proposition for the coherent answer set of logic program $\mathcal{P}(P, I)$.

Theorem 3.6. *Suppose that a coherent set of extended revision literals M is a coherent answer set of $\mathcal{P}(P, I)$. Then $M = M^{+c}$.*

Proof. "⇒" $\forall \alpha \in M$, (1) if $\alpha = in(l)$, $l \in M^+$ then $in(l) \in M^{+c}$; (2) if $\alpha = out(l)$, $l \notin M^+$ then $out(l) \in M^{+c}$. So $LHS \subseteq RHS$.

"⇐" $\forall \alpha \in M^{+c}$, (1) if $\alpha = in(l)$, then $in(l) \in M$; (2) if $\alpha = out(l)$, then $l \notin M^+$ and $in(l) \notin M$. In this case, suppose that $out(l) \notin M$. Since $in(l) \notin M$, then we have $l \in I$, otherwise "$out(l) \leftarrow$"$\in P_N^M$ and then $out(l) \in M$. It conflicts with $out(l) \notin M$; At the same time, since $l \in I$, "$in(l) \leftarrow\sim in(\neg l), \sim out(l)$"$\in P_N$, so we have $in(\neg l) \in M$, otherwise $in(l) \in M$ and then it conflicts with $in(l) \notin M$. Moreover, since $l \in I$, "$out(l) \leftarrow in(\neg l)$"$\in C_I$ and $in(\neg l) \in M$, so $out(l) \in M$, it conflicts with $out(l) \notin M$. From the above discussion, we conclude $out(l) \in M$. Consequently, $RHS \subseteq LHS$. □

From Proposition 3.1, Theorem 3.5 and Theorem 3.6, we clearly have the 1-1 correspondence between extended justified revision and coherent answer set as the next corollary.

Corollary 3.1. *Let R and I be two consistent knowledge bases and let P be an extended revision program. Then R is an extended P-justified revision of I if and only if M is a coherent answer set of $\mathcal{P}(P, I)$, such that $R = M^+$ and $M = R^c$.* □

What the corollary implies is, for an extended revision program P, to compute the extended P-justified revision of a knowledge base I, the existing methods computing answer set of logic program can be applied by regarding $\mathcal{P}(P,I)$ as a logic program over the language $\mathcal{K}$. In this sense, our extended revision program can be embedded into logic program.

The semantics of consistent answer set of extended logic programs can also be easily embedded into our extended justified revision semantics by the next theorem. The proof is similar to Theorem 4.1 of [9].

Theorem 3.7. *A consistent set of literals M is an answer set of extended logic program P if and only if M is an extended $rp(P)$-justified revision of $\emptyset$, where $rp(P)$ is obtained from P by rewriting the rule*

$$l \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n$$

to

$$in(l) \leftarrow in(l_1), ..., in(l_m), out(l_{m+1}), ..., out(l_n).$$

□

In the end of this section, we briefly compare our extension, under a translation *lp*, with nested logic programs [8] and with program updates [6].

Given a consistent extended knowledge base I and an extended revision program P, we define a map *lp* over $\mathcal{P}(P, I)$ by replacing each *in(l)* (resp. *out(l)*) by l (resp. $\sim l$). $lp(\mathcal{P}(P, I))$ is a logic program with nested expression (simply, *nested logic program*).

In order to be self-contained, we abstract the basic notations of nested logic programs here by leaving out the binary connectives "," and ";" for terseness. *Elementary formulas* are classical literals and the symbols $\bot$("false") and $\top$("true"). *Formulas* are built from elementary formulas using the unary connective $\sim$ and the binary connectives ,(conjunction) and ;(disjunction). A *nested logic program* is a set of rules of the form $F \leftarrow G$, which is called *rule*. The formulas, rules, and programs that do not contain the negation as failure operator $\sim$ are called *basic*. A set X of literals *satisfies* an elementary F if $F \in X$, or $F = \top$, or X contains a complementary pair.

Let P be a basic program. A set X of literals is *closed* under P if, for every rule $F \leftarrow G$ in P, $X \models F$ whenever $X \models G$. X is an *answer set* for P if X is a minimal set of literals that is both logically closed and closed under P.

The *reduct* of a formula, rule or program relative to a set X of literals is defined recursively.

- for elementary F, $F^X = F$.
- $(\sim F)^X = \begin{cases} \bot, if \ \ X \models F^X \\ \top, otherwise \end{cases}$
- $(F \leftarrow G)^X = F^X \leftarrow G^X$.
- $P^X = \{(F \leftarrow G)^X | F \leftarrow G \in P\}$

A set X of literals is an answer set of P if it is an answer set for the reduct of P^X. From Corollary 3.1, the next theorem is clear.

Theorem 3.8. *Let I and R be two consistent extended knowledge bases and let P be an extended revision program. If R is an extended P-justified revision of I then R is an consistent answer set of $lp(\mathcal{P}(P,I))$ as nested logic program.* □

However, the reverse of the above theorem is not true. Consider the counterexample, let $I = \{a\}$, P is $\{in(b) \leftarrow out(a)\}$. It is easy to see $R_1 = \{b\}$ is also an answer set of $lp(\mathcal{P}(P, I))$ but $R = \{a\}$ is the unique P-justified revision of I. The more profound connection between revision programs and logic programs needs to be further investigated.

Przymusinski and Turner defined a translation π: $in(p) \rightarrow p$ and $out(p) \rightarrow \neg p$ [13], and then established the 1-1 correspondence between P-justified revision and the extended stable model of $\mathcal{P}^*(P, I)$ (Theorem 5.1). It is easy to see, however, the theorem can not be generalized to our extended revision programs as shown by Example 2.2 if we regard $\neg a$ as a new atom a' for any classically negative literal. In fact, we have $R = \{a', b', c\}$ is a P-justified revision of I but $\mathcal{M}(R)$ is not an extended stable model of $\mathcal{P}^*(P, I)$.

Notably, J. Leite and M. Pereira et al generalized the revision (or update) under inertial principle to program update, i.e. the knowledge bases are represented by logic programs, and they extended the program from normal logic program to extended logic programs [6, 7]. Further, they introduced a sequences of logic programs updates – *Dynamic Programs Updates* [1]. It seems that our extension does fall in their generalization. However, comparing with our extension, their generalization has the following deficiency when we just focus on the update of factual LP:

- Firstly, they need to translate revision programs into extended logic programs, it discriminately translates out's in the head of revision rules into classical negation and out's in the body of the same rules into default negation.
- Secondly, the above translation drastically enlarges the objective language.
- Finally, and more significantly, the translated extended logic programs allow for some pairs of answer-sets, one of which will always be closer than the other to the initial model (knowledge base), as shown by Example 8 of [6] and Example 1.2 of [1].

4 Conclusion and Further Work

In this paper, based on Marek and Truszczyński's P-justified revision, we introduce an extension to deal with explicitly negative information. Some desirable properties of revision program have been generalized to our extension, and we establish the one-to-one correspondence between extended justified revisions of extended revision programs and coherent answer sets of the corresponding extended logic programs.

Actually, many propositions of the new revision programs need to be explored. Firstly, how to deal with the justified revision of inconsistent knowledge base. Secondly, some other semantics for our revision programs, specially, the well-founded semantics like [11], should be studied. Finally, more importantly, the conditions for the existence of extended P-justified revision of a given initial database must be developed, such as auto-compatibility theorem [15] in default logic [14] and so on. We will investigate these issues in the future.

Acknowledgements. We should thank Fangzhen Lin and Jiahuai You for their insightful discussions and suggestions, and the anonymous referees for their comments. This work was partially supported by the National Natural Science Foundation under grant NSF 10161005, and the first author was also partially supported by the Natural Science Foundation of Guizhou university under grant 991010.

References

[1] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska and T. Przymusinski. Dynamic logic programming. In A. Cohn and L. Schubert, editors, KR'98. Morgan Kaufmann, 1998.
[2] C. Baral. Embedding Revision Programs in Logic Programming Situation Calculus. JLP 30(1):83-97, 1997.
[3] T. Eiter, M. Fink, G. Sabbatini and H. Tompits. A Framework for Declarative Update Specifications in Logic Programs. In Proc. IJCAI'01, pp. 649–654.
[4] M. Gelfond and V. Lifschitz. The Stable Semantics for Logic Programs. In R. Kowalski and K. Bowen, editors, Proceedings of the 5th International Symposium on Logic Programming, pages 1070-1080, Cambridge, Ma, 1988. MIT Press.
[5] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing, pages 365-387, 1991.
[6] J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs, In J.Dix, L.M. Pereira and T.C.Przymusinski (eds), Logic Programming and Knowledge Representation, Selected Extended Papers from LPKR'97, pages 224-246, Springer-Verlag, LNAI 1471, 1998
[7] J. A. Leite and L. M. Pereira. Iterated Logic Program Updates, In J. Jaffar (ed.), Procs. of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP'98), Manchester, England, pages 265-278, MIT Press, June 1998.
[8] V. Lifschitz, L. R. Tang and H. Turner. Nested expressions in logic programs. Annals of Mathematics and Artificial Intelligence, Vol. 25, 1999, pp. 369-389.
[9] V. Marek and M. Truszczński. Revision Programming. Theoretical Computer Science, 1998.
[10] V. Marek, I. Pivkina and M. Truszczński. Annotated revision programs. Artificial Intelligence, volume 138(1-2), pp.149-180, 2002.
[11] L. M. Pereira and J. J. Alferes. Well Founded Semantics for Logic Programs with Explicit Negation. In Proc. Of European Conference on AI, 1992.
[12] I. Pivkina. Revision Program: A Knowledge Representation Formalism. Ph.D dissertation at http://cs.engr.uky.edu/~inna/. 2001.
[13] T. Przymusinski and H. Turner. Update by means of inference rules. Journal of Logic Programming, Vol. 30, No. 2, pp. 125-143, 1997.
[14] R. Reiter. A logic for default reasoning. Artif. Intell. 13,Nos. 1-2,81-132. 1980.
[15] Zhang Mingyi. A New Research into Default Logic, Information and Computation. Vol.129, No.2, September 15, 1996.

An Algebraic Approach for Codesign

Marc Aiguier[1], Stefan Béroff[1], and Pierre-Yves Schobbens[2,⋆]

[1] Université d'Évry Val d'Essonne, CNRS UMR 8042, LaMI, F-91025 Évry, France
aiguier@lami.univ-evry.fr
[2] Institut d'Informatique, Facultés Universitaires de Namur,
Rue Grandgagnage 21, B-5000 Namur, Belgium
pys@info.fundp.ac.be

Abstract. This paper gives an answer to Cospecification. Our contribution to cospecification is twofold: allowing both to abstractly specify systems without *a priori* on partitioning step, and achieve *a posteriori* partial correctness proofs of C programs (software parts) and VHDL programs (hardware parts) with respect to specifications.

1 Introduction

The work presented in this paper was performed within the French project ECOS [1]. This project was devoted to hardware / software codesign for telecommunication purposes. The aim of this project was to better answer combined system specifications including both software parts (usually described in C or C++ programs) and hardware parts (described by VHDL programs). Indeed, with traditional designing methodologies, both software and hardware parts designed in an independent way from first designing steps. However, last technological developments allow to define specific components where implementing choices in hardware / software parts can evolve from needs. This is called Codesign. Codesign can then be split into two stages: *cospecification* and *partitioning*. Cospecification deals with the specification of systems, without concern for the division into software and hardware parts. At this stage, we are interested by the abstract behavior of systems: what they are supposed to do. Later, partitioning divides systems into hardware and software parts, looking for an optimal trade-off of cost and performance.

The paper gives an answer to Cospecification. Our contribution to cospecification is twofold:

1. allowing to abstractly specify systems without *a priori* on partitioning step. In practice, this is usually VHDL programs which act as specifications of hardware systems.
2. allowing to achieve *a posteriori* partial correctness proofs of C programs and VHDL programs with respect to specifications.

⋆ This work was partially supported by the CTI project ECOS 941 B 098, by the European Commission under WGs Aspire (22704) and Fireworks (23531).

[1] ECOS is the acronym of *Environnement Générique de Cospécification*.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 415–430, 2005.
© Springer-Verlag Berlin Heidelberg 2005

This first point has been solved by defining a simple extension of algebraic specifications to deal with both dynamic and real-time aspects. The choices of algebraic specifications are many. Algebraic specifications have proven to model elegantly and abstractly the functional aspects of systems. They encourage the natural and powerful method of reasoning by equality. They come with a highly generic theory for both structuring specifications into manageable and reusable modules, and refining abstract specifications into more concrete ones. Finally, they are now in the process of standardization with the language CASL. We firstly proposed to specify hardware telecommunication systems by using an extension of the classical conditional equational logic, the *Exceptions-algebras* [6]. However, it happened that telecommunication systems induce strong dynamic and real-time aspects (mainly, delays and response time constraints), features for which classical algebraic specification methods are not well suited. Our first experiences in the specification of hardware systems ($GMDF_\alpha$: an echo canceler, $AM291$: a small microprocessor, DLX: a RISC processor, etc.) with algebraic formalisms, naturally led us to extend usual algebraic approaches to better deal with both dynamic and real-time aspects. We have thus designed a formalism, called *dynamic real-time specifications*, that extends algebraic specifications as simple as possible. In line with the general spirit of algebraic specifications, the model of time is user-definable, as requested e.g. by the VHDL application detailed below. The dependency of time and state is left implicit in the syntax, as has been proven convenient in modal logics. However, we also allow an explicit handling of time as a data-type, as required by many applications.

The second point has been solved by giving an (axiomatic) semantics to VHDL programs. We have not given meaning of C programs in our formalism because this does not pose difficulties to define it. Indeed, our formalism subsumes Y. Gurevich's evolving-algebras which have already used to give a semantics to C programs. Thus, we can transform any VHDL program into an equivalent dynamic real-time specification. A VHDL program will then be correct with respect to its abstract specifications if its translation into a dynamic real-time specification defines a correct refinement of them (see Section 4 for the complete definitions of these notions). Therefore, this requires first to define a refinement theory within dynamic real-time specifications.

The paper is organized as follows: In Section 2, related works dealing with real-time and dynamic aspects are addressed. In section 3, we introduce the definitions of syntax and semantics of dynamic real-time specifications. Section 4 is devoted to define a refinement theory within dynamic real-time specifications. We also studied structuring within this formalism. For lack of space, this cannot be presented in this paper. However, all results about structuration are available in [1]. In Section 5, we present our (axiomatic) semantics of the hardware description language VHDL.

2 Related Work

The way whose dynamic is dealt with in this paper has taken inspiration from [9] for the use of implicit states. [9] follows the state-as-algebra style where states are

algebras and state transformations are transitions from a state algebra to another state algebra. Various approaches follow the same idea for modeling dynamic data types, e.g. [4], excepted that states are explicit. This approach has been chosen as the basic underlying formalism for defining SB-CASL, an extension of the algebraic specification language CASL to deal with dynamic systems. All these works are mainly influenced by Gurevich's evolving algebra approach. All these works naturally lead to an operational semantics of dynamic systems. This is well-adapted when formalisms are used to provide general semantics for imperative programming languages. However, this may be more restrictive when they are devoted to specify system behaviors because the resulting specifications are concrete, and hence they loose in terseness and legibility.

Concerning real-time aspect, many approaches to real-time specification have been developed in the last decade (see [3] for survey). Most extend some known logic (temporal logic, first-order logic) to deal with real time. They differ in their models of time (real numbers [2], natural numbers [8], specified by properties as here, e.g. [14]), their syntax (with explicit [12] or implicit [2] time as here), their properties (decidability [2, 12], complexity).

Some of these approaches to real time use algebraic concepts, e.g. [14] uses *rewriting* – a technique often used to execute algebraic specifications – and complement it with real time. Process algebras use each a specific algebra, in contrast with the universal algebra approach adopted here. Many process algebras have been extended to deal with real time, e.g. [5]. They can be combined with a classical algebraic approach to define data structures, e.g. [15].

3 Dynamic Real-Time Specifications

This section deals with the specification of real-time dynamic systems. It defines dynamic real-time specifications, dynamic real-time models and their properties.

3.1 Signature and Models

We start by extending the usual notion of algebraic signature in this context. Syntactically, these extensions consist in adding a new sort δ to denote time, as well as operations *date*, $\triangleright$, and ι to explicitly manipulate time. Briefly, *date* returns the current date of the system, ι denotes a zero duration, and $\triangleright$ denotes time addition. Thus, a signature comprises the following user-defined sorts:

Definition 1 (Set of sorts). A *set of sorts* is a set that does not contain the symbol δ. Let us note $\mathcal{S} = S \cup \{\delta\}$.

By considering δ as a simple sort denoting time, we can accommodate several models of time, including $\mathbb{N}$, $\mathbb{Z}^+$, $\mathbb{R}^+$, as well as the δ time of VHDL explained below.

Definition 2 ($\mathcal{S}$-sets). Let S be a set of sorts. A *$\mathcal{S}$-set* is a $\mathcal{S}$-indexed family of sets $A = (A_s)_{s \in \mathcal{S}}$ such that A_δ has a subset A_δ^+ where we distinguish an element

ι_A, and is provided with an internal law $\rhd_A : A_\delta \times A_\delta \to A_\delta$ that satisfies the following conditions:

- upon A_δ^+:
  - $\forall a, b \in A_\delta^+,\ a \rhd_A b \in A_\delta^+$ — closure (Δ_1)
  - $\forall a, b \in A_\delta^+,\ a \rhd_A b = \iota_A \Longrightarrow a = \iota_A \wedge b = \iota_A$ — initiality (Δ_2)
  - $\forall a \in A_\delta^+,\ \iota_A \rhd_A a = a$ — neutral to left (Δ_3)
- upon A_δ:
  - $\forall a, b, c \in A_\delta,\ a \rhd_A (b \rhd_A c) = (a \rhd_A b) \rhd_A c$ — associativity (Δ_4)
  - $\forall a \in A_\delta,\ a \rhd_A \iota_A = a$ — neutral to right (Δ_5)
  - $\forall a, b, c \in A_\delta,\ a \rhd_A b = a \rhd_A c \Longrightarrow b = c$ — cancelable to left (Δ_6)
  - $\forall a, b \in A_\delta,\ \exists c \in A_\delta^+,\ (a \rhd_A c = b) \vee (b \rhd_A c = a)$ — linearity (Δ_7)

Elements of A_δ are dates (or instants) whilst elements of A_δ^+ are durations, or distances between instants. Any duration can be considered as an instant, by considering a conventional origin. The properties given upon A_δ and A_δ^+ are constraints that catch the intuitive view that the time elapses linearly by adding successively durations between them.

Proposition 1. *Let us note $\preceq_A$ the binary relation on A_δ defined as follows: $a \preceq b \Leftrightarrow \exists c \in A_\delta^+,\ b = a \rhd_A c$. Then, $\preceq_A$ is a total order on A_δ.*

Some authors, e.g. [14], add commutativity and Archimedean properties. Commutativity makes intuitive sense for the addition; the Archimedean property excludes Zeno's paradox. However, they are not satisfied by the time used in VHDL, which is an intended application of our formalism. The VHDL time is given by a couple of natural numbers: the first number denotes the real time, the second number denotes the step number in the sequence of computations that must be performed at the same time – but still in a causal order. Such steps are called "δ-steps" in VHDL (and "micro-steps" in StateCharts). The idea is that when simulating a circuit, all independent processes must be simulated sequentially by the simulator. However, the real time (the time of the hardware) must not take these steps into account. Thus, two events e_1, e_2 at dates $(a, 1), (a, 2)$ respectively will be performed sequentially (e_1 before e_2) but at a same real time a. The VHDL addition is defined by the following axioms:

$$(r' \neq 0) \Longrightarrow (r, d) \rhd (r', d') = (r + r', 0)$$
$$(r' = 0) \Longrightarrow (r, d) \rhd (r', d') = (r, d + d')$$

where r, r', d and d' are natural numbers and $+$ denotes the usual addition on natural numbers. Clearly, $\rhd$ is not commutative, nor Archimedean: we may infinitely follow a δ-branch by successively adding δ-times.[2] Thus, in the formalism presented in this paper, we use a weaker version of this property by expressing that from any date we can reach any other date by adding some durations to the first date. This is what is expressed by the condition (Δ_7) of Definition 2. Section 5 gives more details on VHDL's semantics.

[2] This is not the intended use of VHDL time, however: VHDL computations should perform a finite number of δ-steps.

Definition 3 (Signatures). A *signature* Σ consists on a set of sorts S and a set F of operation names f, each one provided with a profile of form $i \to o$ with $i \in \mathcal{S}^*$ and $o \in \mathcal{S} \cup \{\varepsilon\}$ (ε is the empty word in $\mathcal{S}^*$). Let us note $\mathcal{F} = F \cup \{date, \triangleright, \iota\}$ with the profiles $date, \iota :\to \delta$ and $\triangleright : \delta \times \delta \to \delta$.

Example 1. We specify a function *produce* that takes as input a natural number Max and duration d, and produces a list of increasing numbers from 0 to Max in time less than d. Producing a number requires some variable time. Then, the target of the game is to produce as many numbers as possible without exceeding the allowed duration d. Moreover, the produced numbers should be placed as regularly as possible from 0 to Max.

To specify the dynamic operation *produce*, lists of natural numbers are used as implicit states. Thus, the side-effecting version of operations on lists will belong to the signature:

$empty :\to$ (resets the state to the empty list),
$head :\to Nat$ (returns the first element of the list),
$cons : Nat \to$ (add the argument number in front of the list),
$pos : Nat \to Nat$ (the number at the position i in the list, starting with 0),
$length :\to Nat$ (the size of the list).
Nat stands for the usual natural numbers. The profile of *produce* is then $Nat \times \delta \to$.

Notation 1. *Let A be a S-set, and $w = w_1 \dots w_n \in \mathcal{S}^*$. By convention, we let $A_w = A_{w_1} \times \dots \times A_{w_n}$ and $A_\varepsilon = \{\mathbb{1}\}$ where $\mathbb{1}$ is neutral for tuples.*

Definition 4 (Models). Let $\Sigma = (S, F)$ be a signature. A *Σ-model* $\mathcal{A}$ is a $\mathcal{S}$-set A together with a set $\underline{A}$, called the *set of states*, and provided for each operation $f : i \to o \in \mathcal{F}$ with two $\underline{A} \times A_\delta$-indexed families $(f^{\mathcal{A}}_\xi)_{\xi \in \underline{A} \times A_\delta}$ and $(\underline{f}^{\mathcal{A}}_\xi)_{\xi \in \underline{A} \times A_\delta}$ of total functions $f^{\mathcal{A}}_\xi : A_i \to A_o$ and $\underline{f}^{\mathcal{A}}_\xi : A_i \to \underline{A} \times A_\delta$.

Moreover, the supplementary operations *date*, ι, and $\triangleright$ satisfy for any $(\eta, \tau) \in \underline{A} \times A_\delta$ the following conditions:

- $date^{\mathcal{A}}_{(\eta,\tau)} = \tau$; (*date* yields the current date)
- $\underline{date}^{\mathcal{A}}_{(\eta,\tau)} = \underline{\iota}^{\mathcal{A}}_{(\eta,\tau)} = \underline{\triangleright}^{\mathcal{A}}_{(\eta,\tau)} = (\eta, \iota_A)$ (these three functions are instantaneous and without side-effects).

Definition 4 calls for some comments:

- $\underline{A}$ has to be understood as the set of possible states. These states are implicit in this formalism, like in modal logics. They are simply used (together with time) as an implicit environment that modifies dynamic operation semantics. The behavior of dynamic operations can then depend on (implicit) time.
- Our extension is upward-compatible: usual algebraic specifications can be used by declaring any operation f to be environment-independent, that is: $\underline{f}^{\mathcal{A}}_{(\eta,\tau)}(a) = (\eta, \iota_A)$ and $f^{\mathcal{A}}_\xi(a) = f^{\mathcal{A}}_{\xi'}(a)$.

– Elements of $\underline{A} \times A_\delta$ denote *environments* whose the effect is to associate different semantics to operations. Functions of form $f^{\mathcal{A}}_{\xi}$ define the expected result of f under the environment ξ while $\underline{f}^{\mathcal{A}}_{\xi}$ defines the side-effects of f (i.e. both the new state reached and the time used to perform f) when f is started in the environment ξ. Thus, a function f which is defined to be *instantaneous and without side-effects* (i.e. $\underline{f}^{\mathcal{A}}_{(\eta,\tau)} = (\eta, \iota_A)$) does not mean that it is environment-independent.

3.2 Terms and Their Evaluation

Since we consider implicit states, the order in which terms are performed is significant: we consider both sequential and parallel composition of terms.

Definition 5 (Terms). Let $\Sigma = (S, F)$ be a signature. Let V be a $\mathcal{S}$-set where elements will be used as variables. For any $\alpha \in \mathcal{S}^*$, let us note $T_\Sigma(V)_\alpha$ the least set satisfying:

"_", denoting the empty tuple, belongs to $T_\Sigma(V)_\varepsilon$
if $x \in V_s$ then $x \in T_\Sigma(V)_s$;
if $t \in T_\Sigma(V)_\alpha$ and $t' \in T_\Sigma(V)_\beta$, then (t, t') and $(t; t')$ belong to $T_\Sigma(V)_{\alpha\cdot\beta}$; [3]
if $t \in T_\Sigma(V)_\alpha$ and $t' \in T_\Sigma(V)_\beta$, then $[t]; t' \in T_\Sigma(V)_\beta$;
if $(f : i \to o) \in \mathcal{F}$ and $t \in T_\Sigma(V)_i$, then $f(t) \in T_\Sigma(V)_o$.

"_" is neutral for "," and ";". Moreover, "," and ";" are associative.
We note $T_\Sigma(V) = (T_\Sigma(V)_\alpha)_{\alpha\in\mathcal{S}^*}$.

Comments: The comma "," is the parallel evaluation of terms, whilst the semicolon ";" is the sequential evaluation. The result of both (t, t') and $(t; t')$ is the tuple composed by the results of t and t'. $[t]; t'$ performs t, discards its results but keeps its side-effects, and then evaluates t'.

Term Evaluation. To evaluate a term $t \in T_\Sigma(A)_\alpha$, we take an environment $\xi = (\eta, \tau) \in \underline{A} \times A_\delta$ as input, and return a tuple $r \in A_\alpha$, as well as a resulting environment $\xi' = (\eta', \tau') \in \underline{A} \times A_\delta$ as output by following a bottom-up evaluation strategy. However, the evaluation is not deterministic due to side-effects of methods. Indeed, as method semantics relied on environments, results of term evaluations will depend on evaluation order of subterms. Term evaluation is then contingent from environments [4]. Therefore, term evaluation is not a function anymore but rather, is a binary relation on the product (values,environments). Consequently, the evaluation of a term $t \in T_\Sigma(A)_\alpha$ leads to a set of pairs $(r, \xi') \in A_\alpha \times (\underline{A} \times A_\delta)$.

With such an approach, we can see that the evaluation of some terms is deterministic. Among these are terms in $T_\Sigma(A)$ of form $f(a)$ where a is a tuple of values ($a \in A_i$). We call such terms *flat terms*. Their evaluation is defined as follows:

[3] "." is the usual concatenation law on words.
[4] In the literature on modal logics, we say that terms are not *rigid*.

Definition 6 (Flat term evaluation). Given a flat term $t = f(a)$ with $f : i \to o$ and an environment $\xi = (\eta, \tau) \in \underline{A} \times A_\delta$, its *evaluation from* ξ, noted $t_\xi^{\mathcal{A}}$, is $(b, \eta', \tau') \in A_o \times \underline{A} \times A_\delta$ where $b = f_\xi^{\mathcal{A}}(a)$, $(\eta', d) = \underline{f}_\xi^{\mathcal{A}}(a)$ and $\tau' = \tau \rhd_A d$.

To evaluate more general terms, we have to consider all possible evaluation orders. This leads to consider the notion of possible occurrence, which is a place where evaluation could occur.

Definition 7 (Possible occurrence). With notations of Definition 6, a flat term t is a *possible occurrence* of a term t' if and only if t is a subterm of t' such that if t' is of the form $(t_1; t_2)$ or $[t_1]; t_2$ then: if t_1 is already completely evaluated (i.e. a tuple of values) then t is a possible occurrence of t_2 else t is a possible occurrence of t_1.

Definition 8 (Evaluation). We note $\mathcal{E}$ the binary relation on $T_\Sigma(A) \times (\underline{A} \times A_\delta)$ defined by: $(t, \xi)\mathcal{E}(t', \xi')$ if and only if there exists a possible occurrence s of t such that $s_\xi^{\mathcal{A}} = (a, \xi')$ and t' is obtained from t by substituting the occurrence of s in t by a.

An *evaluation of* $t \in T_\Sigma(A)_\alpha$ *from* ξ is then any couple $(r, \xi') \in A_\alpha \times (\underline{A} \times A_\delta)$ where $(t, \xi)\mathcal{E}^*(r, \xi')$.[5]

3.3 Formulas and Their Satisfaction

We have three kinds of atoms to respectively deal with values, states and times. Furthermore, we introduce a supplementary operator to deal with dynamic aspects: **after**.

Definition 9 (Formulas). Let Σ be a signature. A Σ-*formula* is built from:
equational atoms of form $t = t'$ where t and t' are terms with variables of the same sort;
equational atoms of form $t \equiv t'$ where t and t' are terms with variables;
relational atoms of form $t \preceq t'$ where t and t' are terms with variables of the sort δ;
formulas of form **after** $[t]$ (φ) where t is a term with variables and φ is a Σ-formula;
connectives in $\{\neg, \wedge, \vee, \Rightarrow\}$ and quantifiers in $\{\forall, \exists\}$ with their usual rules.

Comments: $t = t'$ denotes equality between values. Since the side-effect may be different, equal values are not replaceable. $t \equiv t'$ denotes equalities between states. Finally, **after** $[t]$ (φ) means that φ is true just after having performed t. Afterwards, $t \cong t'$ will be an abbreviation for $t = t' \wedge t \equiv t' \wedge [t]; date = [t']; date$. In this last case, equal values become replaceable.

Definition 10 (Dynamic real-time specification). A *specification* SP is a couple (Σ, Ax) where Σ is a signature and Ax a set of formulas.

[5] $\mathcal{E}^*$ denotes the reflexive and transitive closure of $\mathcal{E}$.

In many applications, we need to define constraints on the future evolution of systems. For instance, we will see that VHDL instructions have long effect after their execution, although this execution does not take any time. Therefore, we have to go in the future to see instruction side-effects. To describe this evolution, we introduce a dynamic operation $\# : \delta \rightarrow$ that will put us in the environment that we'll reach by letting its argument duration elapsed. Thus it will obey the axioms:

$\#(d); date = date \rhd d$
$\#(d_1); \#(d_2) \cong \#(d_1 \rhd d_2)$
$\#(\iota) \cong _$

Example 2. The axioms to define dynamic lists are the following:

$\mathbf{after}\ [cons(n)]\ (head = n)$
$pos(0) = head$
$pos(i) = m \Rightarrow \mathbf{after}\ [cons(n)]\ (pos(i+1) = m)$
$\mathbf{after}\ [empty]\ (lenght = 0)$
$lenght = m \Rightarrow \mathbf{after}\ [cons(n)]\ (lenght = m+1)$

produce can be specified abstractly as follows:

$$\left\{\begin{matrix} date = h \wedge \\ length = 0 \end{matrix}\right\} \Longrightarrow \mathbf{after}[produce(n,d)] \left\{\begin{matrix} date \preceq h \rhd d \wedge \\ i < j < length \Rightarrow pos(i) < pos(j) \wedge \\ i < length \Rightarrow pos(i) \leq n \end{matrix}\right\} \quad (1)$$

Satisfaction of equations (between states or values) can be interpreted either as *set equality* where two terms are equal if they return the same set of results, or an *element equality* where two terms are equal if they are deterministic (i.e. the set of values is a singleton) and always return the same result [16]. Usually, the set equality of equations is chosen when dealing with specifications which are intrinsically non deterministic (e.g. choice functions). In this logic, the non determinism of the term evaluation is the consequence of evaluation strategies. Therefore, it is a matter for implementation. In line with the general spirit of algebraic frameworks the aim of which is to abstractly specify systems, we choose element equality, which is more implementation-independent.

Definition 11 (Value-deterministic). Let $\mathcal{A}$ be a Σ-model. A term $t \in T_\Sigma(A)_\alpha$ has a *deterministic value* $v \in A_\alpha$ from $\xi \in \underline{A} \times A_\delta$ means that, for any (v', ξ_1) such that $(t, \xi)\mathcal{E}^*(v', \xi_1)$, $v = v'$.

Definition 12 (State-deterministic). With the notations of Definition 11, a term $t \in T_\Sigma(A)$ has a *deterministic state* $\eta \in \underline{A}$ from $\xi \in \underline{A} \times A_\delta$ means that, for any $(v', (\eta', \tau'))$ such that $(t, \xi)\mathcal{E}^*(v', (\eta', \tau'))$, $\eta = \eta'$.

Definition 13 (Formula satisfaction). Let $\mathcal{A}$ be a Σ-model. Let φ be a Σ-formula. Let $\nu : V \rightarrow A$ be an assignment (V covering all the variables of φ). Let ξ be an environment in $\underline{A} \times A_\delta$. $\mathcal{A}$ *satisfies* φ from ξ with ν, noted $\mathcal{A} \models_{\xi,\nu} \varphi$, if and only if:

- if φ is of the form $(t = t')$ (resp. $t \equiv t'$) then $\mathcal{A} \models_{\xi,\nu} t = t'$ (resp. $\mathcal{A} \models_{\xi,\nu} t \equiv t'$) means that, $\nu^{\#}(t)$ and $\nu^{\#}(t')$ have the same deterministic value v (resp. the same deterministic state η) from ξ;[6]
- if φ is of the form $t \preceq t'$ then $\mathcal{A} \models_{\xi,\nu} t \preceq t'$ means that, $\nu^{\#}(t)$ and $\nu^{\#}(t')$ have respectively both deterministic values v and v' from ξ, and $v \preceq_A v'$;
- if φ is of the form **after** $[t]$ (ψ) then $\mathcal{A} \models_{\xi,\nu}$ **after** $[t]$ (ψ) means that for all (v, ξ') such that $(\nu^{\#}(t), \xi)\mathcal{E}^*(v, \xi')$, $\mathcal{A} \models_{\xi',\nu} \psi$;
- propositional connectives and first order quantifiers are handled as usual.

Given a Σ-formula φ, $\mathcal{A}$ *satisfies* φ, noted $\mathcal{A} \models \varphi$, means that for all assignments $\nu : V \to A$ (V covering all the variables of φ) and all environments $\xi \in \underline{A} \times A_\delta$, $\mathcal{A} \models_{\xi,\nu} \varphi$. Finally, $\mathcal{A}$ satisfies a specification SP, noted $\mathcal{A} \models SP$, if and only if $\mathcal{A}$ satisfies all its formulas. $\mathcal{A}$ is then called a *SP-model*.

4 Refinement of Dynamic Real-Time Specifications

Here, we are going to take advantage from showing that the dynamic real-time specification logic is an institution [11] to define a refinement theory for dynamic real-time specifications.

4.1 An Institution for Dynamic Real-Time Specifications

In first, we must define an appropriate morphism notion between models.

Definition 14 ($\mathcal{S}$-morphism). Given a set of sorts S, a *$\mathcal{S}$-morphism* between two $\mathcal{S}$-sets A and B is a $\mathcal{S}$-indexed family $\mu = (\mu_s)_{s \in \mathcal{S}}$ of total functions $\mu_s : A_s \to B_s$ such that $\mu_\delta(A_\delta^+) \subseteq B_\delta^+$. For $w \in \mathcal{S}^*$, μ_w is acting on tuples.

Definition 15 (Σ-morphism). Given a signature Σ, a *Σ-morphism* between two Σ-models $\mathcal{A}$ and $\mathcal{B}$ is a $\mathcal{S}$-morphism from A to B together with a total function $\underline{\mu} : \underline{A} \to \underline{B}$ such that for every operation name $f : i \to o$ of Σ, every tuple $a \in A_i$ and every environment $(\eta, \tau) \in \underline{A} \times A_\delta$, we have: $\mu_s(f^{\mathcal{A}}_{(\eta,\tau)}(a)) = f^{\mathcal{B}}_{(\underline{\mu}(\eta),\mu_\delta(\tau))}(\mu_i(a))$ and $\underline{\mu}(\underline{f}^{\mathcal{A}}_{(\eta,\tau)}(a)) = \underline{f}^{\mathcal{B}}_{(\underline{\mu}(\eta),\mu_\delta(\tau))}(\mu_i(a))$. Moreover, μ is compatible with the predicate $\preceq$: $\forall a, b \in A_\delta,\ a \preceq_A b \Rightarrow \mu_\delta(a) \preceq_B \mu_\delta(b)$).

Given a signature Σ, Σ-models and Σ-morphisms (with the usual composition) clearly form a category: let us note it $Mod(\Sigma)$.

An essential ingredient which is missing is an appropriate notion for dynamic real-time signatures.

Definition 16 (Signature morphism). Let $\Sigma = (S, F)$ and $\Sigma' = (S', F')$ be two signatures. A *signature morphism* $\sigma : \Sigma \to \Sigma'$ is a pair $(\sigma_{sorts}, \sigma_{fun})$

[6] $\nu^{\#}$ is the canonical extension of ν on terms and formulas.

of mapping $\sigma_{sorts} : S \to S'$ and $\sigma_{fun} : F \to F'$ such that for all $f : i \to o$, $\sigma_{fun}(f) : \sigma^{\#}_{sorts}(i) \to \sigma^{\#}_{sorts}(o)$ where $\sigma^{\#}_{sorts}$ is the natural extension of σ_{sorts} to S^*.

Given a signature morphism σ, let us note $\overline{\sigma}$ its canonical extension to Σ-formulas.

Definition 17 (Reduct functor). Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. The *reduct functor* $_{-\restriction_\sigma} : Mod(\Sigma') \to Mod(\Sigma)$ is defined as follows:

- for each Σ'-model $\mathcal{A}$, $\mathcal{A}_{\restriction_\sigma}$ is the Σ-model $\mathcal{B}$ where:
  - $\forall s \in \mathcal{S},\ B_s = A_{\sigma_{sorts}(s)}$ and $\underline{B} = \underline{A}$;
  - $\forall f \in F,\ \forall \xi \in \underline{B} \times B_\delta,\ f^{\mathcal{B}}_\xi = \sigma_{fun}(f)^{\mathcal{A}}_\xi \wedge \underline{f}^{\mathcal{B}}_\xi = \underline{\sigma_{fun}(f)}^{\mathcal{A}}_\xi$;
  - $\forall f \in \{date, \rhd, \iota\},\ \forall \xi \in \underline{B} \times B_\delta,\ f^{\mathcal{B}}_\xi = f^{\mathcal{A}}_\xi \wedge \underline{f}^{\mathcal{B}}_\xi = \underline{f}^{\mathcal{A}}_\xi$;
- for each Σ'-morphism $\mu : \mathcal{A} \to \mathcal{A}'$, $\mu_{\restriction_\sigma} : \mathcal{A}_{\restriction_\sigma} \to \mathcal{A}'_{\restriction_\sigma}$ is the Σ-morphism defined by all the restrictions of μ of the form: $(\mu_{\restriction_\sigma})_s = \mu_s : A_{\sigma_{sorts}(s)} \to A'_{\sigma_{sorts}(s)}$ for every $s \in \mathcal{S}$, and $(\underline{\mu}_{\restriction_\sigma}) = \underline{\mu} : \underline{A} \to \underline{A'}$.

Theorem 1 (Institution of dynamic real-time specifications) *The quadruple $INS_{DRS} = (Sig_{DRS}, Mod_{DRS}, Sen_{DRS}, \models_{DRS})$ is an institution where Sig_{DRS} is the category of dynamic real-time signatures, $Sen_{DRS} : Sig_{DRS} \to Set$ is the functor which maps each Σ to the set of Σ-formulas, $Mod_{DRS} : Sig_{DRS} \to Cat$ is the contravariant functor which maps each Σ to $Mod(\Sigma)$, and $\models_{DRS} = (\models_\Sigma)_{\Sigma \in |Sig_{DRS}|}$ where $\models_\Sigma$ is the satisfaction relation defined in Definition 13.*

4.2 Syntax of Refinement

Specification refinement consists on adding new operations and removing axioms of specifications to replace them by more concrete ones.

Notation 2. *Let $SP = (\Sigma, Ax)$ be a specification. Let us note $Sig[SP] = \Sigma$. A signature morphism $\sigma : \Sigma = (S, F) \to \Sigma' = (S', F')$ is an* inclusion signature morphism *if $S \subseteq S'$ and $F \subseteq F'$.*

Definition 18 (Specification refinement). A specification SP_{imp} is a *refinement* of a specification SP if and only if there is an inclusion signature morphism $\sigma : Sig[SP] \to Sig[SP_{imp}]$. Let us note $SP \rightsquigarrow_\sigma SP_{imp}$ such a refinement.

Example 3. Let us add to the signature of Example 1, the following operations:
$timesub : \delta \times \delta \to \delta$ (subtraction of durations)
$timediv : \delta \times \delta \to Nat$ (division of durations, truncated)
$generate : \delta \times \delta \to$ (dynamic: generate a list before its deadline)

and the following axioms:

$\tau_1 \preceq \tau_2 \Longrightarrow \tau_1 \rhd timesub(\tau_2, \tau_1) = \tau_2$
$\neg(\tau_1 \preceq \tau_2) \Longrightarrow timesub(\tau_2, \tau_1) = 0$
$\neg(\tau_1 \preceq \tau_2) \Longrightarrow timediv(\tau_2, \tau_1) = 0$
$\tau_1 \preceq \tau_2 \Longrightarrow timediv(\tau_2, \tau_1) = succ(timediv(timesub(\tau_2, \tau_1), \tau_1))$

We will produce the list in decreasing order since we add elements with *cons*, so that the last element added is found in the first place. Thus we start with n (if time allows):

$$date \rhd d = e \wedge (cons(n); date) \preceq e \wedge cons(n); date = date \rhd p \Longrightarrow produce(n, d) \cong cons(n); generate(p, e) \quad (2)$$

Otherwise, we just do nothing:
$date \rhd d = e \wedge \neg(cons(n); date) \preceq e \Longrightarrow produce(n, d) \cong _$ (3)

The operation *generate* is recursively defined: the induction step computes a n' that should be evenly spaced if all later steps take the same time:

$$\left\{ \begin{array}{l} (n = head) \wedge (n > 0) \\ \wedge n' = n - (n \ div \ timediv(timesub(e, date), p)) \\ \wedge date \rhd p' = cons(n'); date \wedge date \rhd p \preceq e \end{array} \right\} \Longrightarrow generate(p, e) \cong cons(n'); generate(p', e) \quad (4)$$

Above, *div* is the integer division rounded above, so that n' is less than n.
Otherwise, when either the deadline is reached or the bottom value has been generated, we do nothing (the omitted condition is similar to the previous one):
$\ldots \Longrightarrow generate(p, e) \cong _$ (5)

Obviously, this specification is a syntactic refinement of the specification given in Example 1 and Example 2.

4.3 Semantics of Refinement

A refinement will be correct provided that the behavior of the implementation is indistinguishable from the behavior of the higher level specification under consideration. When dealing with loose semantics, this consists on cutting down in model classes.

Notation 3. *Given a specification $SP = (\Sigma, Ax)$, the category of Σ-models $\mathcal{M}$ such that $\mathcal{M} \models_\Sigma Ax$ is noted $Mod(SP)$. A Σ-sentence φ is a* semantic consequence *of a specification SP if and only if for every $\mathcal{M} \in |Mod(SP)|$, $\mathcal{M} \models \varphi$. The set of semantic consequences of SP is noted $SP^\bullet$.*

Definition 19 (Semantic refinement). Let $SP \rightsquigarrow_\sigma SP'$. SP' is a *semantic refinement* of SP, written $SP \Vdash_\sigma SP'$, if and only if $Mod(SP')_{\restriction_\sigma} \subseteq Mod(SP)$.

Example 4. The specification given in Example 3 is a semantic refinement of the specification of Example 1 and Example 2. Indeed, we can easily show that Axiom (1) given in Example 2 can be deduced from Axioms (2), ..., (5) given in Example 3.

Semantic refinement well expresses refinement correctness, that is the implementation satisfies all the properties of the implemented specification.

Proposition 2. *Let $SP \Vdash_\sigma SP'$ be a semantic refinement. Then, we have: $SP^\bullet \subseteq SP'^\bullet$.*

4.4 Refinement Composition

Of course, it is not reasonable to refine a specification as a whole in a single step. Large softwares usually require many refinement steps before obtaining efficient programs. This leads to the notion of sequential composition of refinements steps. Usually, composition of enrichment is mainly divided into two concepts: horizontal composition and vertical composition. Horizontal composition deals with refinement of subparts of system specifications when they are structured into specification "blocks". In [1], we have shown that all classical modularity results are preserved in the dynamic real-time specification framework when specifications are structured through the basic set of specification building operations in $\{union, translate, derive\}$. As the institution proof system of [7] defined from $\{union, translate, derive\}$ has been shown sound and complete w.r.t. semantic refinement (see Theorem 4.5 in [7]), we directly have that the horizontal refinement correctness holds in our refinement theory.

On the contrary, vertical composition deals with many refinement steps. It corresponds to the transitive closure of refinement relations $\rightsquigarrow_\Sigma$. Its correctness is expressed by the following result:

Theorem 2. *The following rule is sound:* $\frac{SP \rightsquigarrow_\sigma SP' \quad SP' \rightsquigarrow_{\sigma'} SP''}{SP \rightsquigarrow_{\sigma' \circ \sigma} SP''}$

5 VHDL Semantics

In this section, we provide the hardware description language VHDL with an algebraic semantics. This semantics is rather descriptive in contrast to existing formal descriptions of VHDL which are more operational [13]. We have adopted a description of VHDL according to software engineering principles. These principles are twofold:

1. Abstraction and functional description of the programming language, that is a formula only means a property linking inputs and outputs of functions.
2. Semantics must reflect user's view. Therefore, it must avoid, as much as possible, to exploit internal mechanisms which are not directly manipulated by users.

This leads us to not explicitly describe the simulator of VHDL. It is simply observed.

For lack of space, this presentation is rather succinct. Here, we will only give meaning to instructions devoted to simulation, that is suspension instructions

and signal assignments. Other instructions are usual ones that we find in any imperative language. Their semantics is not problematic. A complete algebraic semantics of VHDL from dynamic real-time specifications is given in second author's PhD.

5.1 The VHDL Language

VHDL is a language for describing, in an executable way, hardware systems, mainly integrated circuits. It can be seen both as a programming language (VHDL is based on the programming language ADA) and as a simulation language (VHDL includes some structures allowing us to define and model events, time and signals). To describe the behavior of an hardware system, we define a set of modules. Each module is divided into two parts: an external view, called *entity*, that describes its connexion with the external world (its interface) and an internal view, called *architecture*, that describes its realization. In entities, we mainly give input/output ports and their types. These ports carry *signals*.

Architectures implement entities. An architecture is composed of processes. Each process runs cyclically. A process can suspend its execution, to wait for some event [7]. Therefore, in addition of usual instructions found in any imperative language, VHDL is equipped with two supplementary kinds of instructions which are specific of hardware description: *suspension instructions* and *signal assignments*. Suspension instructions are of the form: **wait** {**on** sequences of signals} {**until** condition} {**for** duration}. And, signal assignments are of the form: signal identifier $\Leftarrow$ value {**after** duration}.

A VHDL description of a hardware system mainly describes a concurrent world whose active objects are processes. Each process is performed sequentially, but this sequencing is only there for simulating, and takes no time when implemented on silicon. To represent this, time is defined as couples of natural numbers. The first element is the real time, the time of the circuit. This time is discrete, with as unit the femtosecond ($10^{-15}s$). The second element, the δ *time*, is used to sequence instructions which have to be performed at a same real time. Therefore, the performance of a VHDL description results on a sequence of signal affectations, usually called *transactions*, distributed on a time line.

5.2 Our VHDL Semantics

Suspension Instructions. *suspension* instructions stop their process for a given duration, or until a condition is satisfied. Our description simply states that a suspended process cannot be act on its signals during its suspension. Here, we only detail suspension instructions of the form: **wait** signal **for** durations. The other cases (**wait on** and **wait until**) are handled in the same way.

Each VHDL instruction (e.g. **wait for** 10 ns) will be represented by an instantaneous side-effecting function, e.g. $wait_for : Process \times \delta \times \delta \rightarrow$. The argument $Process$ will be the identity of the process in which this instruction

[7] An *event* is a change on a signal.

is found. The second argument is simply the duration found in the instruction. The third argument is the time after which the instruction will be executed, depending on the instructions found before. It allows to take into account both some suspension and/or signal assignment instructions which may have taken place before the suspension instruction under consideration (see Paragraph on the semantics of processes below). These instructions must be performed before it. Therefore, the execution of the suspension instruction under consideration is delayed till the end of all the suspension durations of these instructions.

$$date = t_c \Longrightarrow \mathbf{after}[Wait_for(P, time, delay)]$$

$$\mathbf{after}[\#(delay)]\,\forall\alpha\mathbf{after}[\#(\alpha)]\left(\begin{array}{c}\left(\begin{array}{c}t' \preceq time \wedge \\ signal_out(P,s) = true\end{array}\right) \\ \Downarrow \\ \left(trans(s, expr, t_c \rhd delay \rhd t') = false\right)\end{array}\right)$$

$signal_out(P, s) = true$ means that s is an output signal for the process P, and $trans(s, expr, t) = true$ means that a transaction has been required at time t to assign the value $expr$ to the signal s. The date of request cannot be in a suspended interval.

Therefore, the axiom above means that P cannot require transaction on any of its output signals s during all the duration $time$.

Signal Assignments. VHDL signal assignments do not assign a new value immediately, but rather posts a transactions on it at a later date. Some of these transactions may be preempted by transactions posted later: the idea is that a change that does not last long enough will not be performed by the hardware. Roughly, the VHDL rule is: *A signal assignment will actually occur iff there are no transaction on the same signal with a different value which have been created after its performance and before the actual date of this affectation.*

A VHDL assignment "signal $\Leftarrow$ expr **after** time" will again be represented by a function symbol $Assign_sig : Type \times Process \times Type \times \delta \times \delta \rightarrow$. For instance, $Assign_sig(s, P, 0, 5, 10)$ corresponds to the instruction $s \Leftarrow 0$ **after** 5 ns executed by the process P at time 10 ns, e.g., because the previous instruction was of the form **wait for** 10 ns (see Paragraph just below).

Now, the formal rule simply expresses the preemption rule:

$$\left(\begin{array}{c}signal_out(P,s) = true \wedge \\ date = t_c\end{array}\right) \Longrightarrow \mathbf{after}[Assign_sig(s, P, expr, time, delay)]$$

$$\left(\mathbf{after}[\#(delay)]\left(\begin{array}{c}\left(\begin{array}{c}pre_before(s, expr, tc \rhd time) = false \wedge \\ pre_after(s, expr, tc \rhd time) = false\end{array}\right) \\ \Updownarrow \\ \mathbf{after}[\#(time)](trans(s, expr, t_c \rhd delay) = true)\end{array}\right)\right)$$

The predicates pre_after and pre_before (definition not shown here but given in second author's PhD) express the preemption rules defined by the VHDL norm. Briefly, $pre_before(s, expr, tc \rhd time)$ is false if there are no transaction on s which have been produced after the date t_c and will take effect before $tc \rhd time$.

In the same way, $pre_after(s, expr, tc \triangleright time)$ is false if there are no transaction on s with a different value which have been produced after the date t_c and will take effect after $tc \triangleright time$. Therefore, the axiom above means that if s is an output signal of P and the preemption rules are satisfied on s (according to pre_before and pre_after specifications) then there is a transaction on s at time $tc \triangleright delay$ on the time line.

An algorithm translating VHDL programs towards dynamic real-time specifications has been defined along the lines sketched above in second author's PhD thesis. Our way to express temporal properties is general: by using quantification, ordering on time and the operator **after**$[\#(time)]$, we can express any property expressible in temporal logic. It is in fact not obvious for a temporal logic to reach the level of first-order expressiveness [10]. It is also more expressive than known real-time extensions of temporal logic [3]. This is, of course, at the cost of decidability.

6 Conclusion

In this paper, we have introduced a new algebraic-like formalism to describe – and semantically reason about – dynamic real-time systems. The modifications are simple and rather minimal: we introduce notations for state, duration and date. Due to the power of first-order logic, we can immediately express all usual real-time constraints, such as delays, timeouts or response-time, and every property of temporal logic. A refinement theory adequate for this formalism has been defined. This enables us to introduce implementation details progressively. Finally, we have sketched an application, which uses our framework to give a semantics to VHDL.

References

1. M. Aiguier, S. Béroff, and P.-Y. Schobbens. An algebraic approach to real-time specification. Technical report, University of Evry val-d'Essonne, 2001.
2. R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.
3. R. Alur and T. Henzinger. Logics and models of real time: a survey. In J. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, number 600 in LNCS, pages 74–106. Springer, 1992.
4. E. Astesiano and E. Zucca. D-oids: A model for dynamic data types. *Math. Struct. Comp. Sci.*, 5(2):257–282, 1995.
5. J. Baeten and J. Bergstra. Real-time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.
6. G. Bernot, P. L. Gall, and M. Aiguier. Label algebras and exception handling. *Science of Computer Programming*, 23:227–286, 1994.
7. T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 2002. paraitre.

8. S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proceedings of the 15th Annual Real-time Systems Symposium*. IEEE Computer Society Press, 1994.
9. P. Dauchy and M.-C. Gaudel. Algebraic specifications with implicit state. Technical Report LRI-887-1994, Universit de Paris-Sud, 1994.
10. D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic*. Clarendon Press, 1994.
11. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, Jan. 1992.
12. E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, 1990.
13. C. D. Kloos and P.-T. Breuer, editors. *Formal Semantics for VHDL*. Kluwer Academics Publishers, 1995.
14. P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2–3):225–246, 1997.
15. L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29(3):271–292, Feb. 1997.
16. M. Walicki and S. Meldal. Algebraic approaches to nondeterminism: An overview. *ACMCS*, 29(1):30–81, Mar. 1997.

Duration Calculus: A Real-Time Semantic for B

Samuel Colin[1,2], Georges Mariano[1], and Vincent Poirriez[2]

[1] INRETS*, 20, rue Elisée RECLUS, BP 317 F-59666 Villeneuve d'Ascq Cedex, France
[2] LAMIH**, UMR CNRS 5830, Le Mont Houy, 59313 Valenciennes Cedex 9, France

Abstract. Among the possible approaches for expressing real-time problems with the ***B*** method, two are dominant : the use of the usual ***B*** mechanisms to define real-time constraints on the one hand, and extending ***B*** through another formalism more adapted to the real-time context on the other hand.

We define here a possible real-time semantic for ***B***, by using a real-time logic (the duration calculus), and we illustrate how this extension affects the proof mechanism used to show the soundness of abstract machines.

1 Introduction

For several years, feedback on the use of formal methods in the industrial field has been, in majority, positive. Indeed, for instance, the ***B*** method showed its strength in helping the conception of safety-critical systems (for instance, the famous example of line number 14 of the Parisian subway [BBFM99]).

However, the possibilities offered by the formal methods have to evolve at the same time as industrial needs do. Viable methods for the validation of non-functional constraints appear gradually : among them are the time-constrained problems. Indeed, the field of embedded devices and embedded software has great need of methods allowing the designing of solutions including time management, and also, in the case of critical systems, the checking of the *validity* of these solutions.

Such methods already exist, but often show drawbacks that can make the study of some cases difficult. On the one hand, model-checking methods are suitable for the validation of little problems, but when composing those problems to have them interact, the number of cases increase dramatically. There are methods to avoid this problem partially, but they involve often abstract interpretation, which is actually a way of giving semantics to languages. On the other hand, methods using timed automatas have good compositionality properties, but do not allow the description of the step from the abstract modelisation to computer code.

Then the idea of merging these design methods with formal languages and/or methods ensues, so one can benefit from both sides : easy validation of timed constraints from the temporal formalism part, properties of modularity, compositionality and proximity with computer code from the language part.

* Institut National de REcherche sur les Transports et leur Sécurité.
** Laboratoire d'Automatique, de Mécanique, et d'Informatique industrielles et Humaines.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 431–446, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In the next sections, we present a method to obtain a formal tool, allowing the checking of both functional and timed properties of a given problem, by defining a real-time semantic for the ***B*** method. We first describe the formalism used to express this semantic, the duration calculus[1] then we remind the reader of the properties of the ***B*** method. Later on, we describe more in detail a real-time semantic for ***B***, and end up with the possibilities brought by this approach.

2 Duration Calculus

From the site ([DCa]) : "The duration calculus is a modal logic for describing and reasoning about the real-time behaviour of dynamic systems, where states change over time and are represented by functions from time (reals) to the Boolean values (0 and 1). It is an extension of Interval Temporal Logic[2] [Dut95], but with continuous time, and uses integrated durations of states as interval temporal variables. Assuming finite variability of state functions, the axioms and rules of the DC constitute a complete logic (relative to Interval Temporal Logic)."

2.1 History

Research on a real-time logic more powerful than "classical" interval temporal logic (see [Dut95]) was initiated by the ProCos[3] project of the ESPRIT[4] program, in the BRA[5] 3104 and 7071 working groups.

This initiative led in 1990 to the paper entitled "A calculus of durations" [ZHR91], which established the foundations of DC. Then, more advanced studies followed, treating such topics as completeness or decidability (see e.g. [HZ04]), then extensions to DC, like DC with infinite time intervals, or higher-order DC, for instance (see e.g. [DCb]).

There are also examples of the use of DC ([Nai99]) through the design of real-time software, as well as proof assistants for DC. Nowadays, the most active institution in this field is the IIST[6]. Its site [DCa] gathers many links in relationship with DC.

2.2 Classical Modelling of Real-Time Problems

[SH01] presents an example of the study of the watertank problem. This case study interests us because it describes well the required steps in any other study of a real-time problem with DC. These steps are :

1. Problem variables are defined
2. Specifications of the problem are translated into a DC formula we call *Req*

[1] abbreviated as DC from now on.
[2] abbreviated IL from now on.
[3] Provably correct systems.
[4] European Strategic Program for Research in Information Technology.
[5] Basic Research Action.
[6] International Institute for Software Technology, affiliated with the United Nations University.

3. Design decisions are taken and also translated into a DC formula we call *Des*, such that $Des \Rightarrow Req$
4. Design takes place in a dense-time context, thus one may need to make it discreet. In that case, a formula *Cont* must be found, such that $\mathcal{A} \vdash Cont \Rightarrow Des$, $\mathcal{A}$ being a formula stating the behaviour of the environment and the relations between discrete and dense variables.
5. Finally, a program verifying the discrete constraints of *Cont* is written.

Notice that here, the programming step is the last one, and the language used in [SH01] is simple.

Therefore, our idea is to exploit the fact that, in the ***B*** method, the programming step is strongly connected to the proof step through refinement, so that we obtain a simplification of all these steps. Here follows a brief survey of DC with examples.

2.3 Syntax

Let X_i be a *propositional temporal letter* (interpreted as a boolean function over time intervals), P_i a state variable (interpreted as a boolean-valued function over time), $x, y, \ldots$ global variables (interpreted as real numbers), f_i functions and R_i relation symbols. Usually, the functions are the standard arithmetic ones $(+, *)$ and the relations are also the usual ones $(=, \leq)$. The syntax of DC formulas is :

formula ::= Atom | $\neg$ formula | formula $\vee$ formula | formula$^\frown$formula | $\exists x$.formula
Atom ::= **true** | X | R(term,..., term)
term ::= x | ℓ | $\int$state | f(term,..., term)
state ::= 0 | 1 | P | state $\vee$ state | $\neg$ state

Let us mention the fact as functions and relations might be noted with prefix or infix notation, as syntax is not our main concern.

In section 4.1 one can see that predicates are actually used as state variables : this way we can specify changes of some particular states of variables over time (e.g. watch when some variable reaches a critical level). This technique is implicitly used in [SH01].

2.4 Semantics

The most direct way to interpret DC formulas is to do so over time intervals. For space reasons, we will only mention the most noticeable traits of DC semantic : let I be an interpretation of a DC formula over a time interval, $\mathcal{V}$ a valuation of terms over a time interval, and $\mathcal{T}$ a function from time to $\{0,1\}$, we have :

$I(\neg\phi)([b,e]) = \neg I(\phi)([b,e])$ (the semantic of predicate calculus' connectors is as usual)
$I(\phi_1{}^\frown\phi_2)([b,e]) = \exists m.(I(\phi_1)([b,m]) \wedge I(\phi_2)([m,e]))$ for $m \in [b,e]$
$\mathcal{V}(\ell)([b,e]) = e - b$
$\mathcal{V}(\int S)([b,e]) = \int_b^e \mathcal{T}(S)(t)dt$
$\mathcal{T}(0)(t) = 0$
$\mathcal{T}(\neg S)(t) = 1 - \mathcal{T}(S)(t)$

What can we then observe ? The definition of $\frown$ and ℓ are linked, as ℓ has different values among subformulas where several $\frown$ connectors are nested. The introduction of the $\int$ operator over state formulas thus allows the modelling of very fine-grained timed formulas, where it can be reasoned on the duration of events (represented as states), and the relationships between them (some examples are found in sec. 2.5).

A proviso is added for the state variables, which are interpreted as functions over time : for the functions to be integrable, they need to be *finitely variable* over the considered time interval. For example, the following function is not finitely variable over an interval of real numbers :

$$f(t) = \begin{cases} 0 \text{ if } t \text{ is irrational} \\ 1 \text{ otherwise} \end{cases}$$

2.5 DC Examples

Some examples are inspired from [HZ04]:

1. Let the state variables *Gas* and $Flame$ be the expressions of the event "gas is produced" and "flame exists", respectively. Then, this DC formula states that during the non-zero time interval, each time gas is produced, the flame must be present :
$$\int(Gas \Rightarrow Flame) = \ell \wedge \ell > 0$$
2. The formula $\ell = 10 \frown \ell = 5$ states that the first part of the time interval is 10 time units long, and the second part is 5 time units long.
3. $\textbf{true} \frown (\phi \frown \textbf{true})$ states that the ϕ formula is valid in some sub-interval. This special construction is also noted $\Diamond\phi$, and is comparable with the $\Diamond$ one can find in other temporal logics.
4. Similarly, the formula $\neg\Diamond(\neg\phi)$ is noted $\Box\phi$, and is interpreted as : "for any time sub-interval, the ϕ formula is valid".
5. $\lceil\!\lceil S \rceil\!\rceil$ is a convenient notation for $\int S = \ell \wedge \ell > 0$, meaning that the state S holds for a non-zero time interval. This notation is often used in DC formulas.

2.6 WDC*

Numerous papers have described numerous possibilities to extend DC. We present here two extensions whose purpose is to allow to reason about an arbitrary number of time intervals, and to reason on events happening at a time point.

DC* A new logical connector *, which chops an interval an arbitrary (possibly zero) number of times. Hence this connector extends the $\frown$ connector to an arbitrary number of intervals. The axioms for * reflect this behaviour adequately. Its semantic is defined as follows :

$$I(\phi^*)([b,e]) = \begin{cases} \text{true if } b = e, or \\ \exists t_1, \ldots, t_n \in \mathbb{R}. \\ \quad (b = t_1 < \ldots < t_n = e \wedge \bigwedge_{i=1}^{n} I(\phi)([t_i, t_{i+1}]) \end{cases}$$

We also introduce several notations : $\phi^+ \mathrel{\widehat{=}} \phi^\frown(\phi^*)$, $\phi^0 \mathrel{\widehat{=}} \ell = 0$, $\phi^k \mathrel{\widehat{=}} \underbrace{\phi^\frown \ldots ^\frown \phi}_{k \text{ times}}$ for $k > 0$

DC* is preferred over DC when hybridising with a programming language, due to the iteration connector, which allows a more accurate representation of the loops (such as the *while* control structure).

WDC* The weakly monotonic time extension of DC*, adds a new level of time, so that it can be reasoned about succession of events, possibly at a single time point. The time domain is defined to be $(\mathbb{T}, \omega)$ (where $\mathbb{T}$ is the time domain chosen for time intervals, thus usually interval of real numbers). Then each time point can be compared to each other with an order compatible with the lexicographic order. This definition results in the splitting of macro-time points of DC* into one or several micro-time points of WDC*. The syntax changes are as follows :

formula ::= Atom | ¬ formula | formula ∨ formula | formula$^\frown$formula | $\exists x$.formula | formula* | $\lceil P \rceil^0$
Atom ::= **true** | X | R(term,..., term)
realterm ::= x_r | ℓ | $\int$state | f_r(term,..., term)
integerterm ::= x_i | η | f_i(term,..., term)
state ::= 0 | 1 | P | state ∨ state | ¬ state

where x_r is a real variable, f_r a real function (such as addition, for instance), x_i an integer variable, f_i an integer function, and P a state. η represents the micro-time level of WDC*, and the semantic of $\lceil P \rceil^0$ is defined as (with θ a function interpreting the value of states over time) :

$$I(\lceil P \rceil^0)([b,e]) = \text{true if } b = e \text{ and } \theta(P)(b) = 1$$

We also have the following notation : $\lceil P \rceil \equiv \neg(\neg\lceil 1 \rceil^0{}^\frown \neg\lceil P \rceil^0{}^\frown \neg\lceil 1 \rceil^0)$ (meaning there is no non-zero micro-time interval where P is false).

By the means of these new connectors, one can now reason about states at precise time points. We encourage the reader to refer to the appendix of [SH01] where a complete presentation of WDC* is made.

2.7 Proof System and Support

Without going much into detail, it can be stated that the proof system of duration calculus is particular, due to the particular nature of ℓ (see the second example of sec. 2.5 where ℓ seems to have two differents values) and the side-conditions of some of its axioms and inference rules. Indeed, some axioms and side-conditions require that the formula has a certain shape (it must not contain $^\frown$, for instance) so it can be applied.

Several proof tools have overcome these problems, through different approaches : *PC/DC* implements DC through its semantics (deep-embedding), *Isabelle/DC* uses shallow-embedding[7], [Ras02] describes an implementation of Signed Interval Logic

[7] also referred to as external embedding, because Isabelle is a meta-engine.

with Isabelle, from which results for DC can be deduced. There exists also DC libraries for Coq (presented in [CPM03]) illustrating both the shallow-embedding and deep-embedding approaches.

There are also tools rather based on model-checking methods ([Pan01] is such a tool). Many of these tools can be found at [DCa].

3 B Method

We will suppose the reader is familiar with the B method or formal methods of the same kind (Z, for instance). We will simply recall its greatest characteristics here.

To prove the correctness of an operation of a B machine, a proof obligation for this operation must be proved : this proof obligation is generated from the operation itself and the invariant of the machine. We are particularly interested in this building rule, as it is based on Hoare's triples and weakest precondition calculus : the Hoare's triple $\{pre\}[S]\{post\}$ is valid if $pre \Rightarrow wpc(S, post)$, where *pre* and *post* are pre- and post-condition respectively, and S a substitution. This formula is read as : "the precondition establishes the weakest precondition of *S* w.r.t. its postcondition". In the B method, this weakest-precondition calculus is noted $[S]post$ and follows the rules mentioned in fig. 1, the machine invariant plays the role of both *pre* and *post*, and *S* is the operation whose correctness we want to prove. The reader can refer to [Abr96] for more details. Note that we might use "substitution" or "GSL" equally in the next paragraphs, for the sake of simplicity.

3.1 Real-Time Constrained Problems with "Classical" B

The use of the set theory allows the developer to adapt the proof to its framework. Therefore it is possible to express models with real-time needs. In [Lan98], a commu-

GSL	$[GSL]P$	description
$skip$	P	"Do nothing" substitution
$x := E$	$P[E/x]$	All the occurrences of x are replaced by E
$g\|S$	$g \wedge [S]P$	Precondition
$g \Longrightarrow S$	$g \Rightarrow [S]P$	Guard
$S;T$	$[S]([T]P)$	Sequence
$S[]T$	$[S]P \wedge [T]P$	Bounded choice
$@x.S$	$\forall x[S]P$	Unbounded choice
$S\|\|T$	simplified with rewriting rules	Non-deterministic substitution
WHILE C DO S VARIANT V INVARIANT I	$\forall x(I \wedge C \Rightarrow [S]I)$ $\forall x(I \Rightarrow V \in \mathbb{N})$ $\forall x(I \wedge C \Rightarrow [n := V][S](V < n))$ $\forall x(I \wedge \neg C \Rightarrow P)$	While loop (these formulas allow to ensure that the loop terminates)

Fig. 1. Calculus of the weakest precondition

nication protocol between a SmartCard and its reader is modelled in B. A component with real-time constraints is also described in [TS99]. Unfortunately, there are still some limitations.

In [Lan98], temporal constraints do not include hard real-time needs, i.e. quantified time intervals, but rather constraints on the order of the steps of the protocol. And in [TS99], the model calls a clocks it updates itself, and does not allow the “triggering” of the different operations, although it allows to check the operations function correctly if triggered during specific time intervals. Moreover, with this approach, proof obligations can become complex, even more if they are composed together so they communicate, but function according to different clocks.

Hence, even if it is possible to model time-constrained problems in "classical" B, the complexity of proof obligations potentially generated, as well as the external modelling of time (the problem is not subject to time, but handles it through a machine acting as a clock), limit the class of problems that can be addressed.

3.2 *Event B* and Real-Time Constraints

Event B (see e.g. [Abr00]) is an extension of B allowing abstract specification of reactive systems. Thus, it is easier to model protocols requiring concurrency, or systems described by events that can happen in it.

In [HJMO03], timed automatas are used in conjunction with ***event B*** to model timed event systems. The presented example is a railroad crossing, in which the train can take several states (modelled by a set), and to each event is associated a transition having the train go from one state to the next. We have here a correspondence between the events’ system of the ***B*** machine and a timed automata representing the different states the system can be in, as well as associated transitions.

This approach has two advantages : many clocks can be defined to improve the number of real-time properties of the model, and the refinement of timed automata is intuitive, giving the ability to check that many properties of the model are kept at the refinement step. But there are also subtle points one has to take care of : the events can not be triggered explicitly, thus one can only act on variables of the model to ensure the triggering of the guard of the awaited event. This way of modelling keeps ***event B*** away from an immediate implementation.[8] Additionally, The more clocks there are in the model, the more constraints on them may appear in events’ guards, hence the harder the proof obligations can be.

4 Timed Extension for "Classical" B

The extension presented here is based on [SH01], in which the presented method rather follows the design steps presented in section 2.2. So, instead of having logical specifications as a basis to validate a solution to a time constrained problem, we use the

[8] This is half a problem, though, as ***event B*** has been designed for abstract modelisation, and there are systems allowing to explicit the operational semantic of events ([BF03]).

GSL	$dur([\text{GSL}], P)$
skip	$[\![ ]\!]$
$x := E$	$[\![ ]\!]$
delay d	$(\ell = d) \wedge [\![P]\!]$
$g \mid S$	$dur([S], P)$
$g \Longrightarrow S$	$dur([S], P)$
$S;T$	$dur([S], ([T]P)) \frown dur([T], P)$
$S[\!]T$	$dur([S], P) \vee dur([T], P)$
$@x.S$	$dur([S], P)$
$S\|\|T$	transformed through rewriting rules
WHILE C DO S VARIANT V INVARIANT I	$dur([S], I)^*$

Fig. 2. Calculus of a duration formula from a generalised substitution

substitutions of ***B*** to describe the dynamical behaviour by giving them a real-time semantic, so as to extract time informations we are interested in validating.

We define this semantic under the usual ***B*** design hypotheses, that is to say that there is no concurrency, and the substitutions terminate. Moreover, we suppose the true synchrony hypothesis, stating that affectations take zero time. This hypothesis is made because, in practice, the delays used to synchronise the different components of a model are very big w.r.t. the execution time of little instructions (affectations, additions,etc), hence the latter do not play a big role in the core of the modelled problem. Let us notice though, that the possibility remains to add delay statements if one needs to take into account the duration (even little) of particular substitutions.

4.1 Timed Semantic of Substitutions

We present in fig.2 a possible semantic for B substitutions. The formula $dur([\text{GSL}], P)$ means "the duration of the substitution GSL, assuming the postcondition P holds after the substitution has been executed". Indeed, this postcondition is to be checked in the normal development cycle of a ***B*** model (when checking the generated proof obligations with a prover), thus we can safely assume the postcondition holds.

Let us first notice that the calculus of a duration formula bases itself on a predicate : this predicate represents the state whose evolution we want to watch during the "execution" of the substitution (more on this in section 4.2).

The formulas of fig. 2 are obtained by first giving the substitutions a WDC* semantics, and then projecting the obtained formulas into DC*. As an example, let us make the proof of the rule for the *While*, as made in [SH01]. Let us introduce some notations : SCH_i represents the state when a process i (composed of substitutions) is waiting and not running, i.e. is ready to be scheduled. Indeed, this WDC* semantics of substitutions takes into account a possible parallel execution of programs, although we do not use it so far. $\mathcal{M}_{fin}(S)$ represents the semantics of substitution S under the hypothesis that S

terminates (in the timed meaning, not the B meaning). More details about this notation are in [SH01].

The inference rule for the *While* can be written as :

$$\frac{\{[S]I\}[S, dur([S],I)]\{I\} \qquad I \wedge C \Rightarrow [S]I \qquad I \wedge \neg C \Rightarrow P}{\{I\} \begin{bmatrix} \text{WHILE } C \\ \text{DO } S \\ \text{INVARIANT } I \end{bmatrix} , dur([S],I)^* \{P\}}$$

Please note that we omit the side-conditions for the *VARIANT*, as it is not used in the proof. We keep the definition of real-time rules for the *while* :

$$\mathcal{M}_{fin}(WHILE) \equiv (\lceil C \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^* \frown \lceil \neg C \rceil^0$$

Now let us prove that $\lceil I \rceil^0 \frown \mathcal{M}_{fin}(WHILE) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(WHILE) \frown \lceil P \rceil^0$

1. ASSUME: $\{[S]I\}[S, dur([S],I)]\{I\}$
2. ASSUME: $I \wedge C \Rightarrow [S]I$
3. ASSUME: $I \wedge \neg C \Rightarrow P$
4. $\lceil [S]I \rceil^0 \frown \mathcal{M}_{fin}(S) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(S) \frown \lceil I \rceil^0$ (1)
5. $\lceil I \rceil^0 \frown \mathcal{M}_{fin}(WHILE) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(WHILE) \frown \lceil P \rceil^0$

PROOF:

5.1. ASSUME: $\lceil I \rceil^0 \frown \mathcal{M}_{fin}(WHILE)$
5.2. $\lceil I \rceil^0 \frown (\lceil C \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^* \frown \lceil \neg C \rceil^0$ (5.1,definition of *while*)
5.3. $\lceil I \wedge \neg C \rceil^0 \vee (\lceil I \wedge C \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^+ \frown \lceil \neg C \rceil^0$ (5.2,WDC^*)
5.4. $\lceil I \wedge \neg C \rceil^0 \vee (\lceil [S]I \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^+ \frown \lceil \neg C \rceil^0$ (2,5.3)
5.5. $\lceil I \wedge \neg C \rceil^0 \vee (\mathcal{M}_{fin}(S) \frown \lceil I \rceil^0 \frown SCH_i)^+ \frown \lceil \neg C \rceil^0$ (4,5.4)
5.6. $\lceil I \wedge \neg C \rceil^0 \vee (\mathcal{M}_{fin}(S) \frown SCH_i \frown \lceil I \rceil^0)^+ \frown \lceil \neg C \rceil^0$ (5.5,*COND2*)
5.7. $\lceil I \wedge \neg C \rceil^0 \vee (\mathcal{M}_{fin}(S) \frown SCH_i)^+ \frown \lceil I \wedge \neg C \rceil^0$ (5.6,WDC^*)
5.8. $(\mathcal{M}_{fin}(S) \frown SCH_i)^* \frown \lceil I \wedge \neg C \rceil^0$ (5.7,WDC^*)
5.9. $(\lceil C \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^* \frown \lceil I \wedge \neg C \rceil^0$ (5.8,WDC^*)
5.10. $(\lceil C \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^* \frown \lceil \neg C \rceil^0 \frown \lceil I \wedge \neg C \rceil^0$ (5.9,WDC^*)
5.11. $\mathcal{M}_{fin}(WHILE) \frown \lceil I \wedge \neg C \rceil^0$ (5.10,definition of *while*)
5.12. $\mathcal{M}_{fin}(WHILE) \frown \lceil P \rceil^0$ (3,5.11,WDC^*)

□

Then, we prove the duration formula (the proof is similar to the one in [SH01]) :

1. ASSUME: $\{[S]I\}[S, dur([S],I)]\{I\}$
2. $\prod(\lceil [S]I \rceil^0 \frown \mathcal{M}_{fin}(S)) \Rightarrow dur([S],I)$ (1)
3. $\prod(\lceil I \rceil^0 \frown \mathcal{M}_{fin}(WHILE)) \Rightarrow dur([S],I)^*$

PROOF:

3.1. ASSUME: $\prod(\lceil I \rceil^0 \frown \mathcal{M}_{fin}(WHILE))$
3.2. $\prod(\lceil I \wedge \neg C \rceil^0) \vee$
$\prod((\lceil [S]I \rceil^0 \frown \mathcal{M}_{fin}(S) \frown SCH_i)^+) \frown \prod(\lceil \neg C \rceil^0)$ (3.1, definition of *while*, monotony of $\prod$)
3.3. $\ell = 0 \vee (dur([S],I) \frown \ell = 0)^+ \frown \ell = 0$ (3.2,2, definition of $\prod$)

```
S =
  BEGIN
    delay 3;
    IF
      x ≥0
    THEN
      delay 1; skip;
    ELSE
      delay 2; x:=-x;
    END;
  END
```

```
S =
  BEGIN
    delay1;
    ANY
      y
    WHERE
      y ∈ { a | ∃b b >0 ∧a
      =2 ∗b }
    THEN
      x:=y
    END;
  END
```

Fig. 3. Some examples of operations

3.4. $\ell = 0 \vee dur([S],I)^{+}$ (3.3,DC^{*})
3.5. $dur([S],I)^{*}$ (3.4,DC^{*})
□

Thus we have proved that the rules for the *while* are correct w.r.t. its definition in WDC*. The proofs for other substitutions are similar : for instance, the proofs for the bounded and unbounded choice happen to be compatible with the definition of the rules for the *if then else* structure of [SH01]. That is the least one could have expected from two formalisms originally based on Hoare's triple.

Examples. The following examples are not meant to reflect actual time-constrained problems, but rather to give the reader an intuition of the mechanism of the generation of duration formulas from the operations.

Example 1. The duration formula associated with the substitution in figure 3 on the left, knowing that, after execution, we have $x \geq 0$, is :

$$\begin{aligned}
\mathrm{dur}([S], x \geq 0) &= \mathrm{dur}([delay 3], [IF...](x \geq 0) \frown \mathrm{dur}([IF...], x \geq 0) \\
&\quad \mathrm{dur}([delay\ 3], (x \geq 0 \Rightarrow x \geq 0) \wedge (\neg x \geq 0 \Rightarrow -x \geq 0)) \\
&= \frown (\ \mathrm{dur}([x \geq 0 \Longrightarrow delay\ 1; skip], x \geq 0) \\
&\quad \vee \mathrm{dur}([\neg x \geq 0 \Longrightarrow delay\ 2; x := -x], x \geq 0)) \\
&= \ell = 3 \wedge [\![(x \geq 0 \Rightarrow x \geq 0) \wedge (\neg x \geq 0 \Rightarrow -x \geq 0)]\!] \\
&\quad \frown (\ell = 1 \wedge [\![x \geq 0]\!]) \vee (\ell = 2 \wedge [\![-x \geq 0]\!]) \\
&= \ell = 3 \wedge [\![\mathrm{true}]\!] \frown (\ell = 1 \wedge [\![x \geq 0]\!]) \vee (\ell = 2 \wedge [\![-x \geq 0]\!])
\end{aligned}$$

Example 2. The duration formula associated with the substitution in figure 3 on the right, knowing that after execution we have $x \geq 0$, is :

$$\begin{aligned}
\mathrm{dur}([S], x \geq 0) &= \ell = 1 \wedge [\![\forall y, y \in \{a | \exists b, b > 0 \wedge a = 2 * b\} \Rightarrow y > 0]\!] \\
&= \ell = 1 \wedge [\![\mathrm{true}]\!]
\end{aligned}$$

4.2 Use in Abstract Machines

Timed Correctness of Operations. Now that we have timed operations, how do we prove their correctness ? The invariant as a timed requirement seems interesting, but fails on two points : the invariant represent a requirement that must be true between

executions of operations, thus at a time point between operations, and the invariant can be understood as the common subset what all the operations of the machine establish. Hence the invariant will in general be too imprecise w.r.t. the states of variables after an operation. Thus we need to give the developer a way of expressing more precise requirements for each operation. These requirements will be predicates we can base the duration formulas' generation on. This remark relates to those made in [TS99–section 6.4], where to each operation corresponds a timed operation, in which timed constraints are expressed in the precondition.

Then, all we have to do is associate to each substitution of the system a real-time constraint it must establish. To this end, we propose to add a new substitution we name TIMING, whose role is to state the real-time constraint of the substitution it guards. Finally, we have to provide the predicate whose evolution we will watch in the substitution (the P of figure 2). Then, we have two possibilities : use the invariant of the machine as a basis for the calculation of the duration formula, but as stated above the invariant might be too imprecise and contain useless predicates making the resulting formula unwieldy. The other possibility is to use a predicate representing what the operation will have realized, while staying consistent with the invariant of the machine and not containing irrelevant predicates : this solution is presented in [Pet03], in the form of postconditions. Postconditions possess all the characteristics mentioned above : they contain only predicates relevant for the associated operation, and they are informative enough to represent the result of the operation.

TIMING
Substitution
POST
Postcondition (in the form of a predicate)
REQUIRES
Real-time constraint
END

Thus a substitution with a real-time constraint will have the shape indicated in the table above. Then, in order to prove the timed correctness of the substitution, it suffices to generate the corresponding trace with the provided postcondition, and check the constraint is verified, thus :

$$dur([\text{Substitution}], \text{Postcondition}) \Rightarrow \text{Real-time constraint}$$

Example 1. The generated formula for the example of figure 4 on the left side is : $\ell = 1 \wedge \lceil\!\lceil x - 1 \geq 0 \rceil\!\rceil \Rightarrow \Box(\lceil\!\lceil x \geq 0 \rceil\!\rceil)$, which is easy to prove. Note also that the postcondition is intuitively verified.

Example 2. In this example (figure 4, on the right side), two substitutions with real-time constraints are nested. We then have two possibilities : either prove the internal substitution, then not take into account the internal postcondition and time constraint to prove the external substitution, or prove the internal substitution, and, instead of recalculating the part corresponding to the internal substitution, use the time constraint it establishes.

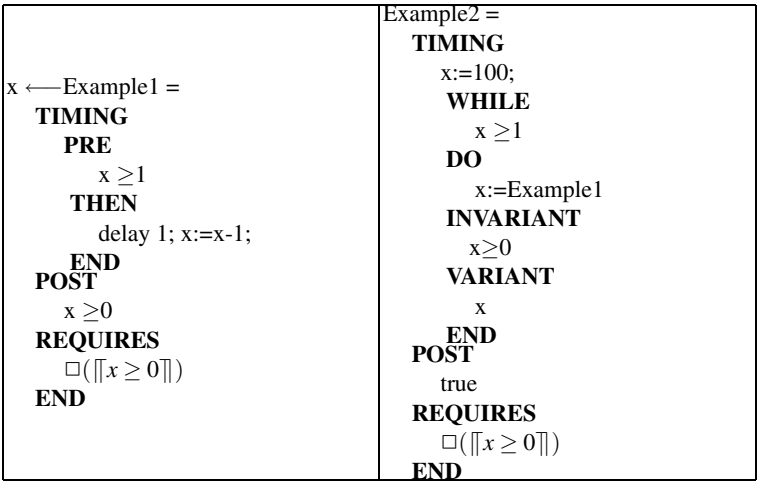


Fig. 4. An example of specification

The generated duration formulas is : $(\Box(\lceil\lceil x \geq 0\rceil\rceil))^* \Rightarrow \Box(\lceil\lceil x \geq 0\rceil\rceil)$. We can easily see that the formula does not hold in the case the loop is not even entered, i.e. in the case the iteration reduces to $\ell = 0$, producing the formula $\ell = 0 \Rightarrow \Box(\lceil\lceil x \geq 0\rceil\rceil)$. The interval is reduced to a time point, thus we can not deduce anything about the state $x \geq 0$. A possible way to make this specification correct, would be to add a delay statement before the loop is entered, in order to ensure that interval is not a time point. We have also replaced the calculus of $\text{dur}([Example1], x \geq 0)$ with the real-time specification established by *Example1*. Indeed, it allows us better precision than using the only invariant, as the postcondition of *Example1* guarantees the invariant is respected (the proof is done at the usual step of verification of the correctness of the proof obligations).

Modularity. Section 4.2 illustrates the way we can validate nested substitutions with real-time constraints. Now, this is the way proof obligations for operations in "classical" ***B*** are generated. We do not want, as stated in [Pet03], to depend on the code of the called operations to keep a *software component* view of ***B*** machines. To this end, what do we need ? First, we need *code independence* towards the operations called in the included machines, i.e. the called operation must satisfy a contract with the help of which we will be able to validate the current operation, without knowing the code of the called operation. Second, we need *limited scope* of the variables, i.e. an operation need not export a contract containing variables from an included machine.

The first constraint is achieved, when generating the duration formula, by using the duration formula of the called operation in the same manner as the nested substitutions of section 4.2. It is up to the developer to achieve the second constraint, as he is the one who determines what each operation must guarantee. Then, the verification of duration formula is done as in section 4.2.

MACHINE	REFINEMENT
...	...
INVARIANT	**INVARIANT**
y ∈ IF(NAT1)	z = max(y ∪{0})
OPERATIONS	**OPERATIONS**
lire(n) =	lire(n) =
PRE	**PRE**
n ∈ NAT1	n ∈ NAT1
THEN	**THEN**
y :=y ∪{n}	z :=max(z,n)
END;	**END**;

Fig. 5. Machine LittleExample and its refinement

Refinement. Refinement in ***B*** allows us the checking of that the result of an operation is not inconsistent with the one of the operation it refines. Thanks to this, we can know that what is calculated by the operation keeps certain properties. It allows also the use of new variables, more concrete ones (in the programming sense), to realise these calculi. The verification is then made by expressing the relation between the new variables and those of the refined machine through a so-called *gluing* invariant.

However, timed verification is about the *way* the operation unfolds, which may cause problems : thus, a refined operation can have a more precise timed trace, but is not allowed to redefine its working steps. So, we have several cases :

Direct Demonstration. If ϕ is the duration formula of the operation, and ϕ' the duration formula of its refinement, check $\phi' \Rightarrow \phi$. In that case, if new, more concrete, variables are introduced, replacing those from the refined machine, the formula is generally not provable. For instance, in figure 5, the variable of the machine, a set, is refined by an integer variable. This means that, although the new variable corresponds functionally to the refined one (proved by the ***B*** proof obligations), the way they are are calculated is different, and then the different states the variable can have during the calculus can differ in the abstraction and in the refinement.

The timed trace must also be strongly similar to the one of the abstraction. For instance, in figure 6, the operation $oper2$ can not refine $oper1$, because the formula to prove would be $\ell = 5 \Rightarrow \ell = 0$, which is false. This way of designing would force us to have a great time precision at the beginning of the modelisation, and that is not what we wish for the designer.

The Contract Approach. The contract approach is another approach to ensure that a refinement will not contradict the operations that might use it : the new operation also fulfils the real-time constraints of the operation it refines. This corresponds to an *operation refinement* approach.

In the example figure 6, we achieve this by removing the real-time constraints from $oper2$, calculating its timed trace with the postcondition of $oper1$, and checking this trace validates the real-time constraints of $oper1$. We have chosen to adopt this more flexible approach, for the timed validation of a refinement. We have just proposed a

oper1 =	oper2 =
TIMING	**TIMING**
skip	delay 5;
POST	**POST**
$x = x$	$x = x$
REQUIRES	**REQUIRES**
$\ell \leq 10$	$\ell \leq 10$
END	**END**

Fig. 6. Example of differently timed refinements

formal description of postconditions and their refinement in [CMP04]. Then, the only remaining problem would be the addition of new variables refining the ones from the abstract machine (which corresponds to *data refinement*).

5 Conclusion

After having presented DC and one of its extension suitable for the verification of real-time programs, as well as some of its properties, we have reminded the reader of the foundations of the ***B*** method, and some of the ways used to express real-time problems with it. We have seen that it is possible to use either the bare formalism, but then we have to face difficulties in the designing and the proof steps, or ***event B*** completed with known methods, coming from the real-time community, but in that case we have to face a lack of tools.

Hence we have showed that another way is possible : extending the most used formalism with a real-time logic. This allowed us to define a real-time semantic for B substitutions without modifying the foundations of the semantic, based on set theory. In fact, the opposite effect was achieved : DC, being defined partly on top of the predicate calculus, allowed us to use the substitutions to find their real-time trace with regard to a postcondition that we know to be correct (the correctness proof was made during the classical ***B*** design stage).

Moreover, the real-time validation step of the operations justifies the need for the modular validation of B machines, by requiring the use of postconditions, and by using the same mechanism as in the call of operations of included machines.

6 Perspectives

Now that we are able to specify and verify a problem with real-time constraints in ***B*** under the usual hypotheses (no concurrency, termination), what is left is to remove these hypotheses in order to benefit from the expressiveness of DC, in order to check, for instance, the railroad crossing problem (see [CBR93] for a general description). With this aim in view, we can express problems with more subtle real-time requirements (for example replacing the real-time logic used in [LFD96] by DC), with an easier treatment of non-terminating (for instance, the validation of a mutual exclusion protocol, as in [SH01]).

It can also be interesting to use DC as a way of expressing and validating fairness and liveness constraints in ***event B***, by expressing time quantifications more naturally (i.e. without using clocks or machines manipulating time).

Another interest of DC is the ability to express timed automatas (see [JH00]), and vice versa, provided certain constraints are respected : it then allows the use of several methods at the same time to model a real-time problem, i.e. a timed automata to specify the real-time behaviour of the model, and use the ***B*** method to build the implementation step by step, with the help of refinement.

In the end, ***B*** is also used to validate UML models (see [ML02]). A real-time extension to B will allow the checking of OCL models with real-time constraints, those being inherited from the corresponding UML model, since UML 2.0 will include notions of time.

References

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[Abr00] Jean-Raymond Abrial. Event driven sequential program construction. École Jeunes chercheurs en programmation, March 2000.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR : A successful application of B in a large project. In Wing et al. [WWD99], pages 369–387.

[BF03] Jean-Paul Bodeveix and Mamoun Filali. Machines virtuelles pour le B événementiel. In *AFADL'2003* [IRI03], pages 227–242.

[CBR93] Heitmeyer C.L, Labaw B.G, and Jeffords R.D. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*. IEEE Computer Society Press, 1993. http://chacs.nrl.navy.mil/personnel/heitmeyer.html.

[CMP04] Samuel Colin, Georges Mariano, and Vincent Poirriez. A natural extension of B substitutions : postconditions. Technical report, LAMIH/ROI, 2004. http://www.univ-valenciennes.fr/ROAD/WP/.

[CPM03] Samuel Colin, Vincent Poirriez, and Georges Mariano. Thoughts about the implementation of the duration calculus with coq. In *4th International Workshop on the Implementation of Logics*, volume Technical report ULCS-03-018. University of Liverpool, september 2003. http://www.csc.liv.ac.uk/research/techreports/.

[DCa] http://www.iist.unu.edu/dc/.

[DCb] http://www.iist.unu.edu/home/Unuiist/newrh/III/1/page.html.

[Dut95] Bruno Dutertre. Complete proof systems for first order interval temporal logic. In *Logic in Computer Science*, pages 36–43, 1995.

[HJMO03] A. Hammad, Jacques Julliand, H. Mountassir, and D. Okalas Ossami. Expression en B et raffinement des sytèmes réactifs temps réel. In *AFADL'2003* [IRI03], pages 211–226.

[HZ04] M.R. Hansen and C.C. Zhou. *Duration Calculus, a formal approach to real-time systems*. Number ISBN: 3-540-40823-1. Springer-Verlag, 2004. Series : Monographs in Theoretical Computer Science. An EATCS Series.

[IRI03] IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

[JH00] Zhao Jianhua and Dang Van Hung. Checking timed automata for some discretisable duration properties. In *Journal of Computer Science and Technology*, volume 15, pages 423–429. September 2000.

[Lan98] Jean-Louis Lanet. Using the B method to model protocols. In *AFADL'98* [LIS98], pages 79–90.

[LFD96] Kevin Lano, J. Fiadeiro, and Jeremy Dick. Extending B AMN with concurrency. Technical report, Dept. of Computing, Imperial College, 1996.

[LIS98] LISI/ENSMA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, Téléport2 - Avenue 1 - BP109 - 86960 FUTUROSCOPE Cedex, October 1998. LISI/ENSMA.

[ML02] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In *Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*, Dresden, Germany, September 2002.

[Nai99] Zhan Naijun. Another formal proof for deadline driven scheduler. Technical Report 169, UNU/IIST, P.O. Box 3058, Macau, august 1999.

[Pan01] P.K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *RT-TOOLS'2001*, Aalborg, August 2001. (affiliated with CONCUR 2001). Technical report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai, 2000.

[Pet03] Dorian Petit. *Génération automatique de composants logiciels sûrs à partir de spécifications formelles B*. Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambrésis, December 2003.

[Ras02] Thomas Marthedal Rasmussen. *Interval Logic - Proof Theory and Theorem Proving*. PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, january 2002.

[SH01] François Siewe and Dan Van Hung. Deriving real-time programs from duration calculus specifications. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume LNCS 2144, pages 92–97, Livingston-Edinburgh, Scotland, september 2001. Springer-Verlag. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[TS99] Helen Treharne and Steve Schneider. Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey TW20 0EX, England, June 1999.

[WWD99] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *Proceedings of FM'99: World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag). Springer Verlag, September 1999.

[ZHR91] C.C. Zhou, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. In *Information Processing Letters*, volume 10(5), pages 269–276. Dezember 1991.

An Algebra of Petri Nets with Arc-Based Time Restrictions

Apostolos Niaouris

School of Computing Science, University of Newcastle,
Newcastle upon Tyne NE1 7RU, U.K.
Apostolos.Niaouris@ncl.ac.uk

Abstract. In this paper we present two algebras, one based on term re-writing and the other on Petri nets, aimed at the specification and analysis of concurrent systems with timing information. The former is based on process expressions (at-expressions) and employs a set of SOS rules providing their operational semantics. The latter is based on a class of Petri nets with time restrictions associated with their arcs, called at-boxes, and the corresponding transition firing rule. We relate the two algebras through a compositionally defined mapping which for a given at-expression returns an at-box with behaviourally equivalent transition system. The resulting model, called the Arc Time Petri Box Calculus (atPBC), extends the existing approach of the Petri Box Calculus (PBC).

Keywords: Net-based algebraic calculi; arc based time Petri nets; process algebras; box algebra; SOS semantics.

1 Introduction

Process algebras, e.g., ACP [2], CCS [15] and CSP[9], provide a formal framework for dealing with large and complex concurrent computing systems by employing specific operators corresponding to commonly used programming constructs. The way of representing system's structure is given through suitably defined set of process expressions, and their behaviour is typically captured by a (structured) set of sequences of executed actions. Another way of modelling concurrent systems is provided by Petri nets [16, 20], which support a graphical representation of concurrent systems and, through their being based on a theory of partial orders (capturing explicit asynchrony), an additional means of verifying their correctness efficiently, and a way of expressing properties related to causality and concurrency in system behaviour.

These two kinds of formalisms treat the structure and semantics of concurrent systems in different ways, which in the past meant that it was almost impossible to take full advantage of their relative advantages (i.e., *compositionality* and *explicit asynchrony*) when used in isolation. To a significant extent, this problem was addressed by the Box Algebra [5, 6] and its precursor, the Petri Box Calculus (PBC) [4]. Both models provided a framework where both Petri nets and

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 447–462, 2005.
© Springer-Verlag Berlin Heidelberg 2005

process algebras could co-exist, and thus established a bridge between these two approaches.

Since their conception, the PBC has been extended to cover, in particular, concurrent systems with timing restrictions [12, 13]. In both cases, the timing restrictions were associated with transitions, effectively specifying for how long an enabled action (or transition) can delay/prolong its execution as well as what is a minimum delay or execution time. Another way in which timing assumption could be introduced is to associate clocks (or age) with the resources (or tokens). More precisely, one can specify how old/young a given resource consumed by an action must be. This approach has been extensively studied in the past, see, e.g., [1, 8, 17], both as a model for dealing with complex concurrent systems such as communication protocols, and as a framework for verifying their properties. It is precisely this kind of time modelling which is our concern in this paper.

We introduce and investigate two different models for the specifications of concurrent systems including explicit timing information of the latter kind. Both models have an algebraic structure based on operators present in the standard PBC.

The first algebra is based on process expressions, called at-expressions, and a system of rewriting rules providing structural operational semantics of at-expressions in the style of [18]. The second algebra is based on a class of Petri nets with arc-based timing restrictions, called at-boxes, and their execution rules. This means, in particular, that: (i) each arc from a place p to a transition is given two time bounds, e and l, representing the *earliest consuming time* and the *latest consuming time*, respectively, for a token which has arrived at place p; (ii) the local clock of a token is started at the very moment it has been created; and (iii) time is discrete. It is important to point out that property (i) suits particularly well the intended compositional setting we are aiming at since the handshake synchronisation of two transitions basically amounts to gluing them together, and no special consideration of their timing restrictions is needed. On the other hand, gluing two transitions in the other time frameworks we mentioned requires combining their timing intervals which can be done in several different ways.

The two algebras are related through a compositionally defined mapping which, for at-expression returns a corresponding at-box (its denotational semantics). The main result is that the denotational and operational semantics of an at-expression are behaviourally equivalent. The resulting framework consisting of two consistent algebras is called the Arc-Based Time Petri Box Calculus, or simply *atPBC*.

The paper is organised as follows. Section 2 recalls some basic notions used throughout the paper, section 3 introduces at-boxes, section 4 provides the syntax and semantics of at-expressions, and section 5 develops a compositional net model based on at-boxes.

Throughout the paper, we assume that the reader is familiar with the basic concepts of PBC and the Box Algebra [5] on which the compositional treatment of nets is based.

2 Basic Notions

Throughout the paper, $\mathbb{N}$ denotes the set of non-negative integers and $\mathbb{N}^\infty \stackrel{\text{df}}{=} \mathbb{N} \cup \{\infty\}$. A *multiset* over a set X is a function $\mu : X \to \mathbb{N}$. We will write $\mu \leq \mu'$ if the domain X of μ is included in that of the multiset μ', and $\mu(x) \leq \mu'(x)$, for all $x \in X$. An element $x \in X$ belongs to μ, denoted $x \in \mu$, if $\mu(x) > 0$. The sum and difference of multisets, and the multiplication by a non-negative integer are respectively denoted by $+$, $-$ and $\cdot$ (the difference will only be applied when the second argument is smaller or equal to the first one). A subset of X may be treated as a multiset over X, by identifying it with its characteristic function, and a singleton set can be identified with its sole element. A multiset μ over X may be denoted as $\sum_{x \in X} \mu(x) \cdot \{x\}$, as well as written in extended set notation, e.g., $\{a, a, b\}$ denotes a multiset μ such that $\mu(a) = 2$, $\mu(b) = 1$ and $\mu(x) = 0$ for all $x \in X \setminus \{a, b\}$.

A tuple $N = (P, T, F, M)$ is a *Place/Transition net* (or PT-net) if: (i) P is a finite set of *places*, T is a finite set of *transitions* disjoint from P, and $F : (T \times P) \cup (P \times T) \to \{0, 1\}$ is a *flow function*; and (ii) $M : P \to \mathbb{N}$ is the *initial marking* (in general, any mapping from P to $\mathbb{N}$ is a marking of N). In what follows, for every $x \in P \cup T$, ${}^\bullet x = \{y \mid F(y, x) = 1\}$ is the *preset* of x and $x^\bullet = \{y \mid F(x, y) = 1\}$ is the *postset* of x; we assume that both sets are always non-empty.

A finite set of transitions U, called a *step*, is *enabled* at a marking M if, for all $p \in P$, $M(p) \geq \sum_{t \in U} F(p, t) \cdot U(t)$. Such a step may *fire* leading to a *follower* marking M' given, for every place $p \in P$, by $M'(p) \stackrel{\text{df}}{=} M(p) - \sum_{t \in U} F(p, t) \cdot U(t) + \sum_{t \in U} F(t, p) \cdot U(t)$. We denote this by $M[U\rangle M'$, and call M' *reachable* from M (in general, a marking can be reachable through a possibly empty sequence of intermediate markings). The net is *safe* if for every marking M reachable from the initial one, it is the case that $M(p) \leq 1$, for all $p \in P$.

To label transitions in nets considered in this paper, we use a fixed set of *communication* actions $\mathcal{A}$ such that for every $a \in \mathcal{A}$, there exists its *conjugate*, $\widehat{a} \in \mathcal{A}$, satisfying $a \neq \widehat{a}$ and $\widehat{\widehat{a}} = a$. Also, there is a *silent* (or internal) action $\imath \notin \mathcal{A}$. In the algebra of nets (as well as in the process algebra), it will be assumed that a *synchronisation* of two conjugate communication actions gives rise to the silent action $\imath$.

3 Boxes with Arc-Based Time Restrictions

An *arc-time box* (or at-box) is a tuple $\Theta = (P, T, F, \lambda, \mu)$ such that: (i) P, T and F are as in the definition of a PT-net; (ii) λ is a mapping with the domain $P \cup T \cup ((P \times T) \cap F^{-1}(\{1\}))$; and (iii) $\mu : P \to \mathbb{N} \cup \{\bot\}$ is the *initial state* of Θ (in general, any such mapping is a state of Θ). For every place $p \in P$ and transition $t \in T$, we have the following: $\lambda(p)$ is a symbol in $\{\mathsf{e}, \mathsf{i}, \mathsf{x}\}$; $\lambda(t)$ is an action in $\mathcal{A} \cup \{\imath\}$; and if $F(p, t) = 1$ then $\lambda(p, t) = (e, l) \in \mathbb{N} \times \mathbb{N}^\infty$, where $e \leq l$. We adopt the standard rules concerning the drawing of diagrams representing Petri nets.

The 'time-less' version of an at-box Θ which is defined as a PT-net $\langle\Theta\rangle \stackrel{\text{df}}{=} (P, T, F, \langle\mu\rangle)$, such that, for every $p \in P$ $\langle\mu\rangle(p) \stackrel{\text{df}}{=} 1$ if $\mu(p) \in \mathbb{N}$ and $\langle\mu\rangle(p) \stackrel{\text{df}}{=} 0$ if $\mu(p) = \perp$. In what follows, $\langle\Theta\rangle$ will be called the *underlying* net of Θ, and we will assume that it is always safe.

Note that states of at-boxes are interpreted differently from markings of PT-nets, namely, $\mu(p) = k$ means that p holds a single token which is k units of time old, and $\mu(p) = \perp$ means that p is empty.

In the at-box model, time restrictions are associated with the arcs incoming to transitions. For example, if $\lambda(p,t) = (e,l)$, then the interval (e,l) gives the waiting time for the tokens flowing from place p to transition t. This interval identifies the time for which a token has to wait in place p before it can be used to fire transition t. The left bound, e, is called the *minimum* waiting time and the right bound, l, the *maximum* waiting time. A token on p cannot be used to fire t when it is younger than the minimum waiting time and must be used to fire an enabled transition before the maximum waiting time has finished (unless the transition has been disabled in the meantime). If t is not enabled and the maximum waiting time has passed, the token can no longer be used to fire transition t. The age of tokens is represented through a state mapping which returns, for each place containing a token, its age ($\perp$ is returned if a given place is empty). When a token arrives to a place, its age is set to zero. After that the age can be increased due to the passage of time. It should be emphasized that a token does not need to enable any transition in order for its clock to start 'ticking'.

A finite set of transitions U, called a *step*, is *enabled* at a state μ if it is enabled at the marking $\langle\mu\rangle$ in the underlying PT-net and, moreover, if $t \in U$ and $p \in {}^\bullet t$ then $e \leq \mu(p) \leq l$ where $(e,l) = \lambda(p,t)$. Such a step may *fire* leading to a *follower* marking μ' given, for every place $p \in P$, by $\perp$ if $p \in {}^\bullet U \setminus U^\bullet$, 0 if $p \in U^\bullet$, and $\mu(p)$ otherwise. We denote this by $\mu[U\rangle\mu'$.

Another kind of dynamic changes is effected by time moves. A state μ can change into state μ' by the passage of one time unit if, for transition t enabled at μ and and every place $p \in {}^\bullet t$ we have $\mu(p) < l$, where $(e,l) = \lambda(p,t)$. The change results in a new state μ' given by $\mu'(p) \stackrel{\text{df}}{=} \mu(p) + 1$ if $\mu(p) \in \mathbb{N}$, and $\mu'(p) \stackrel{\text{df}}{=} \mu(p)$ otherwise. We denote this by $\mu[\surd\rangle\mu'$. Intuitively, at-boxes' time deadlines are defined to be *hard*, i.e., when a transition is ready to fire and even if only one of the tokens that will be consumed has reached its maximum waiting time then this transition must fire (or become disabled) before further passage of time.

The overall behaviour of Θ is captured by its *reachability tree* with nodes labelled by states and arcs labelled by moves, denoted by RT_Θ. In this tree, the root node is labelled by the initial state and, if a node is labelled by a state μ, then for every move $\mu[x\rangle\mu'$ there is a unique descendant labelled by μ'; the arc leading to it is labelled by $\surd$ if $x = \surd$, and by $\lambda(U) \stackrel{\text{df}}{=} \sum_{t\in U} U(t) \cdot \{\lambda(t)\}$ if $x = U$ is an executed step. Figure 1 shows an at-box Θ_1 and the corresponding reachability tree RT_{Θ_1}. The use of reachability trees instead of reachability graphs is quite surprising at the moment but will be explained later in this paper together with the considerations that led to this decision.

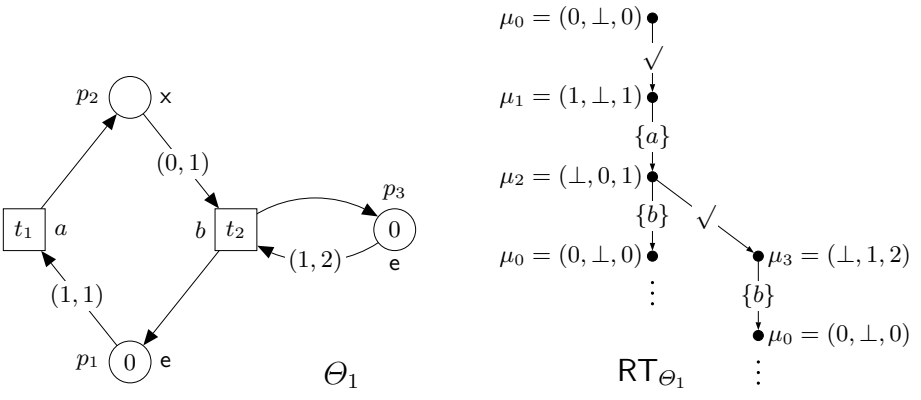


Fig. 1. An at-box Θ_1 and its reachability tree RT_{Θ_1}

4 An Algebra of Process Expressions

The following is the syntax for *static arc-based time box expressions* (or static at-expressions), E, which correspond to at-boxes without tokens.

$$E \ ::= \ \alpha\langle el\rangle \mid E \ \mathsf{sy}\ A \mid E \ \mathsf{rs}\ A \mid E \,[]\, E \mid E \| E \mid E; E \mid [E \circledast E \circledast E] \ .$$

The only modification, when compared with the standard PBC syntax, is that a different type of constant expression is used, viz. $\alpha\langle el\rangle$ where: $\alpha \in \mathcal{A} \cup \{\imath\}$ is a basic action; e is a non-negative integer; and $l \geq e$ is a non-negative integer or ∞. Moreover, the actions employed by the syntax allow two-way rather than multi-way synchronisation. Similarly as in the case of at-boxes, e denotes the *minimum*, and l the *maximum* waiting time.

Sequence $E; F$ and choice $E \,[]\, F$ compositions are standard; the $[]$ is used to denote what is essentially the + in CCS [15] and the comma (,) in COSY [10]. The iterative construct $[D \circledast E \circledast F]$ means 'perform D once, then perform zero or more repetitions of E, then perform F once'. The basic expression $\alpha\langle el\rangle$ means 'upon its activation, execute a single action with communication capabilities α and terminate, waiting at least e units of time and no more than l units of time to do so'. The concurrent composition operator is basically a disjoint union and hence differs from its counterparts in CCS and COSY, and is similar to the $\|_\emptyset$ in TCSP [21]. For instance, $a\langle 00\rangle \| \widehat{a}\langle 00\rangle$ can perform the $\{a\}$ and $\{\widehat{a}\}$ actions individually (as well as a two-action step $\{a, \widehat{a}\}$), but no synchronised action (in contrast to $a.nil | \widehat{a}.nil$ in CCS). Synchronisation can only be achieved through the explicit synchronisation, $E \ \mathsf{sy}\ A$. Essentially, it applies the CCS synchronisation mechanism over all the concurrently enabled pairs $(a, \widehat{a})$ of conjugate action names. Finally, the restriction $E \ \mathsf{rs}\ A$ prevents all the actions a and $\widehat{a}$, where $a \in A$, from occurring.

Static expressions describe structural characteristics of concurrent systems. Their behaviour will be modelled using dynamic at-expressions, introduced next.

The syntax of (standard) dynamic PBC expressions is changed by adding time related annotations to the over- and underbars. Each such annotation is a pair of two non-negative integers that correspond to the *age* of the 'youngest' and 'oldest' token that might be consumed. For example, $\overline{E}^{00}$ is an expression E which is in its initial state and all tokens present are zero time units old. Another example, $\underline{E}_{35}; F$, is a sequential composition where the first component has terminated, and produced some tokens. The exact number (and clock values) of these tokens is not represented by the annotation, but what is represented is the age of the youngest is 3 time units, and of the oldest 5 time units. Effectively, this means that the annotation gives an *age range* for the tokens in the state which is represented by the expression. This, in general, provides less information than that conveyed by the state mappings provided by at-boxes. However, it will turned out that this reduced (or abstracted) view is sufficient to reason about the behaviour. We will re-visit this issue later on.

The *dynamic at-expressions*, G, are defined below, where E denotes a static at-expression, and $\mathbb{E}, \mathbb{L} \in \mathbb{N}$ are such that $\mathbb{E} \leq \mathbb{L}$.

$$G ::= \overline{E}^{\mathbb{E}\mathbb{L}} \mid \underline{E}_{\mathbb{E}\mathbb{L}} \mid G \,\mathsf{sy}\, A \mid \quad G \,\mathsf{rs}\, A \quad \mid \quad G \,\square\, E \quad \mid \quad E \,\square\, G \quad \mid$$
$$G \| G \mid G; E \mid \quad E; G \quad \mid [G \circledast E \circledast E] \mid [E \circledast G \circledast E] \mid [E \circledast E \circledast G] \,.$$

Given that we are primarily interested in at-expressions that can be derived from expressions of the form $\overline{E}^{00}$, the above syntax may appear to be too permissive. For example, it admits an expression $\overline{\alpha\langle 03\rangle}^{55}$ which has an inconsistent timing information (the enabled action cannot wait for more than 3 time units before being executed, yet the age of the enabling tokens is already 5). However, such an expression may be a part of another, fully consistent expression, e.g., $(\overline{\alpha\langle 03\rangle}^{55})\ \mathsf{rs}\ \{a\}$, and thus cannot be excluded.

Operational Semantics of At-Expressions. We follow the way through which the semantics of PBC was defined, with appropriate modifications in order to address timing issues. We first define a structural equivalence relation on at-expressions aims to capture the most fundamental correspondence between expressions. For example, $\underline{E}_{\mathbb{E}\mathbb{L}}; F \equiv E; \overline{F}^{\mathbb{E}\mathbb{L}}$ states that a sequential system in which its first component has terminated is the same as the system in which the second component is ready to begin its operation. The time annotations are not changed since the entire state produced by the first component is passed to the second one. Formally, $\equiv$ is the least equivalence relation on dynamic at-expressions such that the rules in table 1 are satisfied. Note that we do not give any rule for $\overline{E}^{\mathbb{E}\mathbb{L}} \| \overline{F}^{\mathbb{E}'\mathbb{L}'}$ with $\mathbb{E}\mathbb{L} \neq \mathbb{E}'\mathbb{L}'$ as such an expression can never be derived from initially marked static expressions, which are of our primary interest.

Proposition 1. *Assuming that we treat the rules in table 1 as term rewriting rules, if $G \equiv H$ and G is an at-expression, then so is H.*

Table 1. Rules of the structural equivalence for at-expressions

$$\overline{E \| F}^{\mathbb{EL}} \equiv \overline{E}^{\mathbb{EL}} \| \overline{F}^{\mathbb{EL}} \qquad \underline{E}_{\mathbb{EL}} \| \underline{F}_{\mathbb{E'L'}} \equiv \underline{E \| F}_{\min\{\mathbb{E},\mathbb{E'}\}\,\max\{\mathbb{L},\mathbb{L'}\}}$$

$$\overline{E \,[]\, F}^{\mathbb{EL}} \equiv \overline{E}^{\mathbb{EL}} \,[]\, F \qquad \underline{E}_{\mathbb{EL}} \,[]\, F \equiv \underline{E \,[]\, F}_{\mathbb{EL}}$$

$$\overline{E \,[]\, F}^{\mathbb{EL}} \equiv E \,[]\, \overline{F}^{\mathbb{EL}} \qquad E \,[]\, \underline{F}_{\mathbb{EL}} \equiv \underline{E \,[]\, F}_{\mathbb{EL}}$$

$$\overline{E \text{ rs } A}^{\mathbb{EL}} \equiv \overline{E}^{\mathbb{EL}} \text{ rs } A \qquad \underline{E}_{\mathbb{EL}} \text{ rs } A \equiv \underline{E \text{ rs } A}_{\mathbb{EL}}$$

$$\overline{E \text{ sy } A}^{\mathbb{EL}} \equiv \overline{E}^{\mathbb{EL}} \text{ sy } A \qquad \underline{E}_{\mathbb{EL}} \text{ sy } A \equiv \underline{E \text{ sy } A}_{\mathbb{EL}}$$

$$\overline{E; F}^{\mathbb{EL}} \equiv \overline{E}^{\mathbb{EL}}; F \qquad E; \underline{F}_{\mathbb{EL}} \equiv \underline{E; F}_{\mathbb{EL}}$$

$$\underline{E}_{\mathbb{EL}}; F \equiv E; \overline{F}^{\mathbb{EL}} \qquad \overline{[D \circledast E \circledast F]}^{\mathbb{EL}} \equiv [\overline{D}^{\mathbb{EL}} \circledast E \circledast F]$$

$$[\underline{D}_{\mathbb{EL}} \circledast E \circledast F] \equiv [D \circledast \overline{E}^{\mathbb{EL}} \circledast F] \qquad [D \circledast \underline{E}_{\mathbb{EL}} \circledast F] \equiv [D \circledast E \circledast \overline{F}^{\mathbb{EL}}]$$

$$[D \circledast \overline{E}^{\mathbb{EL}} \circledast F] \equiv [D \circledast \underline{E}_{\mathbb{EL}} \circledast F] \qquad [D \circledast F \circledast \underline{F}_{\mathbb{EL}}] \equiv \underline{[D \circledast E \circledast F]}_{\mathbb{EL}}$$

Similarly as at-boxes, at-expressions can perform two kinds of operational semantics moves, namely *action* moves and *time* moves. A time move has the form $G \xrightarrow{\surd} H$ and an action move has the form $G \xrightarrow{\Gamma} H$ where $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$ is a finite multiset ($k \geq 0$). Each γ_i is an *action occurrence* of the form α^δ where: $\alpha \in \mathcal{A} \cup \{\imath\}$ is a communication or silent action, and $\delta \in \{0, 1\}$ indicates whether this particular occurrence can be delayed or not ($\delta = 1$ means that we can still delay executing α without violating the hard time deadline rule, while $\delta = 0$ means that time cannot yet progress and so this occurrence can be considered as *urgent*). We now define various types of moves of the structural operational semantics of dynamic at-expressions.

Empty Moves. The following rules deal with the empty action moves.

$$\frac{G \equiv H}{G \xrightarrow{\varnothing} H} \qquad \frac{G \xrightarrow{\varnothing} J \xrightarrow{\Gamma} H}{G \xrightarrow{\Gamma} H} \qquad \frac{G \xrightarrow{\Gamma} J \xrightarrow{\varnothing} H}{G \xrightarrow{\Gamma} H}$$

Basic Action. A basic action can occur if its timing restrictions are satisfied by the age range of its overbar:

$$\overline{\alpha\langle el \rangle}^{\mathbb{EL}} \xrightarrow{\{\alpha^\delta\}} \underline{\alpha\langle el \rangle}_{00} \qquad \text{where } e \leq \mathbb{E} \text{ and } \mathbb{L} \leq l \text{ and } \delta = 0 \Leftrightarrow \mathbb{L} = l$$

Note that the age range of a newly created underbar is always set to (00).

Restriction and Synchronisation. There is a single rule for restriction:

$$\frac{G \xrightarrow{\Gamma} H}{G \mathsf{\ rs\ } A \xrightarrow{\Gamma} H \mathsf{\ rs\ } A} \quad \text{where } \alpha \notin A \cup \widehat{A} \text{ for every } \alpha^{\delta} \in \Gamma$$

and two synchronisation rules. The first one states that a synchronised expression can do all that the original one could, while the second captures the essential meaning of the standard handshake synchronisation:

$$\frac{G \xrightarrow{\Gamma} H}{G \mathsf{\ sy\ } A \xrightarrow{\Gamma} H \mathsf{\ sy\ } A} \qquad \frac{G \mathsf{\ sy\ } A \xrightarrow{\Gamma + \{\, a^{\delta}, \widehat{a}^{\delta'} \,\}} H \mathsf{\ sy\ } A}{G \mathsf{\ sy\ } A \xrightarrow{\Gamma + \{\iota^{\delta \cdot \delta'}\}} H \mathsf{\ sy\ } A} \quad \text{where } a \in A$$

The second rule essentially means that two conjugate action occurrences, a^{δ} and $\widehat{a}^{\delta'}$, can always be synchronised. Moreover, such a synchronisation is urgent iff at least one of its participant was.

Other Operators. There is no real difference in the rules for the remaining operators when compared with the standard PBC [5, 6].

$$\frac{G \xrightarrow{\Gamma} G' \ ,\ H \xrightarrow{\Gamma'} H'}{G \| H \xrightarrow{\Gamma + \Gamma'} G' \| H'} \qquad \frac{G \xrightarrow{\Gamma} H}{\begin{array}{l}[G \circledast E \circledast F] \xrightarrow{\Gamma} [H \circledast E \circledast F] \\ [E \circledast G \circledast F] \xrightarrow{\Gamma} [E \circledast H \circledast F] \\ [E \circledast F \circledast G] \xrightarrow{\Gamma} [E \circledast F \circledast H]\end{array}}$$

$$\frac{G \xrightarrow{\Gamma} H}{\begin{array}{l}E \,\square\, G \xrightarrow{\Gamma} E \,\square\, H \\ G \,\square\, E \xrightarrow{\Gamma} H \,\square\, E\end{array}} \qquad \frac{G \xrightarrow{\Gamma} H}{\begin{array}{l}G; E \xrightarrow{\Gamma} H; E \\ E; G \xrightarrow{\Gamma} E; H\end{array}}$$

Time Moves. There is a single time rule:

$$\frac{\neg\exists\ G \xrightarrow{\{\alpha^0\}} H}{G \xrightarrow{\surd} G \oplus 1}$$

where $G \oplus 1$ is G with each time annotation $\mathbb{EL}$ at an over- or underbar changed to $(\mathbb{E}+1)(\mathbb{L}+1)$. Although the above rule is a rule with negative premise, the inference rule system is well defined since time moves are not used in the premise of any rule for the action moves. Notice that a time move can only be applied at the topmost level of an expression as it cannot be 'propagated up' through the expression using action rules.

$$
\begin{array}{lll}
(1) & \overline{(a\langle 02\rangle; c\langle 44\rangle) \parallel (b\langle 11\rangle; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\}}^{\,00} & \equiv \\
(2) & ((\overline{a\langle 02\rangle}^{\,00}; c\langle 44\rangle) \parallel (\overline{b\langle 11\rangle}^{\,00}; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\} & \xrightarrow{\{a^1\}} \\
(3) & ((\underline{a\langle 02\rangle}_{00}; c\langle 44\rangle) \parallel (\overline{b\langle 11\rangle}^{\,00}; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\} & \xrightarrow{\surd} \\
(4) & ((\underline{a\langle 02\rangle}_{11}; c\langle 44\rangle) \parallel (\overline{b\langle 11\rangle}^{\,11}; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\} & \xrightarrow{\{b^0\}} \\
(5) & ((\underline{a\langle 02\rangle}_{11}; c\langle 44\rangle) \parallel (\underline{b\langle 11\rangle}_{00}; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\} & \equiv \\
(6) & ((a\langle 02\rangle; \overline{c\langle 44\rangle}^{\,11}) \parallel (b\langle 11\rangle; \overline{\widehat{c}\langle 14\rangle}^{\,00}))\ \mathsf{sy}\ \{c\} & \xrightarrow{\surd\surd\surd} \\
(7) & ((a\langle 02\rangle; \overline{c\langle 44\rangle}^{\,44}) \parallel (b\langle 11\rangle; \overline{\widehat{c}\langle 14\rangle}^{\,33}))\ \mathsf{sy}\ \{c\} & \xrightarrow{\{\imath^0\}} \\
(8) & ((a\langle 02\rangle; \underline{c\langle 44\rangle}_{00}) \parallel (b\langle 11\rangle; \underline{\widehat{c}\langle 14\rangle}_{00}))\ \mathsf{sy}\ \{c\} & \equiv \\
(9) & \underline{(a\langle 02\rangle; c\langle 44\rangle) \parallel (b\langle 11\rangle; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\}}_{00} &
\end{array}
$$

Fig. 2. An evolution of the expression $\overline{(a\langle 02\rangle; c\langle 44\rangle) \parallel (b\langle 11\rangle; \widehat{c}\langle 14\rangle))\ \mathsf{sy}\ \{c\}}^{\,00}$

Proposition 2. *Assuming that we treat the rules of the operational semantics as term rewriting rules, and H has been derived from an at-expression, then H is also an at-expression.*

We are interested in at-expressions of the form $G = \overline{E}^{\,00}$, and to capture its behaviour we will use a *reachability tree*, denoted by RT_G. Its nodes are labelled by equivalence classes of dynamic expressions reachable from G, and arcs are labelled by multisets over $\mathcal{A} \cup \{\imath\}$ or the $\surd$ symbol. The root node is labelled by $[G]_{\equiv}$ and, if a node is labelled by $[H]_{\equiv}$, then: for every move $H \xrightarrow{\Gamma} J$, there is a unique descendant labelled by $[J]_{\equiv}$ and the arc leading to it is labelled by the multiset $\sum_{\alpha^\delta \in \Gamma} \Gamma(\alpha^\delta) \cdot \{\alpha\}$, and if the time move is possible for H then there is a unique descendant labelled by $[H \oplus 1]_{\equiv}$ and the arc leading to it is labelled by $\surd$.

Examples. Our first example, in figure 2, shows an at-expression with two sequential actions *a,c* in parallel with two other sequential actions *b,* $\widehat{c}$ and a possible synchronisation in action *c*. Different execution scenarios can be followed. We choose, in line (2), to execute action a followed by a time move in line (3) which is the only possible move at this stage. Action b becomes urgent and in line (4) b is executed. After three time moves, an interesting point in line (6) is that both c and $\widehat{c}$ are executable but only action c is urgent. No time move is allowed. Synchronisation will take place in line (7) by executing the silent synchronisation action $\imath$. Even though only one of the synchronised actions is urgent, the silent action is also marked as urgent. The second example, in figure 3, shows an at-expression consisting of an action a in parallel with two sequential actions b, $\widehat{a}$ and a possible synchronisation of actions a and $\widehat{a}$. In

$$
\begin{array}{lll}
(1) & \overline{(a\langle 00\rangle \parallel (b\langle 11\rangle; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\}}^{00} & \equiv \\
(2) & (\overline{a\langle 00\rangle}^{00} \parallel (\overline{b\langle 11\rangle}^{00}; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\} & \xrightarrow{\{a^0\}} \\
(3) & (\underline{a\langle 00\rangle}_{00} \parallel (\overline{b\langle 11\rangle}^{00}; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\} & \xrightarrow{\checkmark} \\
(4) & (\underline{a\langle 00\rangle}_{11} \parallel (\overline{b\langle 11\rangle}^{11}; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\} & \xrightarrow{\{b^0\}} \\
(5) & (\underline{a\langle 00\rangle}_{11} \parallel (\underline{b\langle 11\rangle}_{00}; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\} & \equiv \\
(6) & (\underline{a\langle 00\rangle}_{11} \parallel (b\langle 11\rangle; \overline{\widehat{a}\langle 01\rangle}^{00}))\ \mathsf{sy}\ \{a\} & \xrightarrow{\{\widehat{a}^1\}} \\
(7) & \underline{(a\langle 00\rangle \parallel (b\langle 11\rangle; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\}}_{01} &
\end{array}
$$

Fig. 3. An evolution of the expression $\overline{(a\langle 00\rangle \parallel (b\langle 11\rangle; \widehat{a}\langle 01\rangle))\ \mathsf{sy}\ \{a\}}^{00}$

line (2), action a is urgent and must be executed immediately but $\widehat{a}$ is not yet enabled. In this example, the synchronisation between a and $\widehat{a}$ was not allowed by the time restrictions.

5 An Algebra of Arc-Time Boxes

We now extend the box algebra to at-boxes, by defining compositionally a mapping box which, for static at-expressions, returns at-boxes. The net algebra employs operators directly corresponding to (and denoted as) those used in the algebra of static at-expressions. All the net operators are similarly as in the standard PBC with two important modifications: (i) changing the definition of the basic net corresponding to a single action, and (ii) taking care of the time restrictions associated with transition input arcs. Essentially, that latter means that if p and t are a place and transition which are 'carried forward' by a net operator, then the associated time constraint $\lambda(p,t)$ is also carried forward. Moreover, in the synchronisation operation, if t and t' are fused together to yield a $\imath$-labelled synchronisation transition u, then we assume that ${}^\bullet t \cap {}^\bullet t' = \varnothing$ and $t^\bullet \cap t'^\bullet = \varnothing$. We omit here a full definition of the composition operators, and instead provide in figure 4 a number of examples involving the operators used in the algebra of at-boxes.

We introduce a denotational semantics of at-expressions through the semantical mapping box from static at-expressions to at-boxes so that $\mathsf{box}(\alpha\langle el\rangle) = \mathsf{N}_{\alpha\langle el\rangle}$, where $\mathsf{N}_{\alpha\langle el\rangle}$ is shown in figure 4, and for other static at-expressions: $\mathsf{box}(E\ \mathsf{sy}\ A) = \mathsf{box}(E)\ \mathsf{sy}\ A$, $\mathsf{box}(E\ \mathsf{rs}\ A) = \mathsf{box}(E)\ \mathsf{rs}\ A$, $\mathsf{box}(E \,\square\, F) = \mathsf{box}(E) \,\square\, \mathsf{box}(F)$, $\mathsf{box}(E \| F) = \mathsf{box}(E) \| \mathsf{box}(F)$, $\mathsf{box}(E; F) = \mathsf{box}(E); \mathsf{box}(F)$ and $\mathsf{box}([D \circledast E \circledast F]) = [\mathsf{box}(D) \circledast \mathsf{box}(E) \circledast \mathsf{box}(F)]$. Since we are interested in the behaviour of systems starting from their initial state, we also need to describe $\mathsf{box}(G)$, for any dynamic at-expression of the form $G = \overline{E}^{00}$. The appropriate

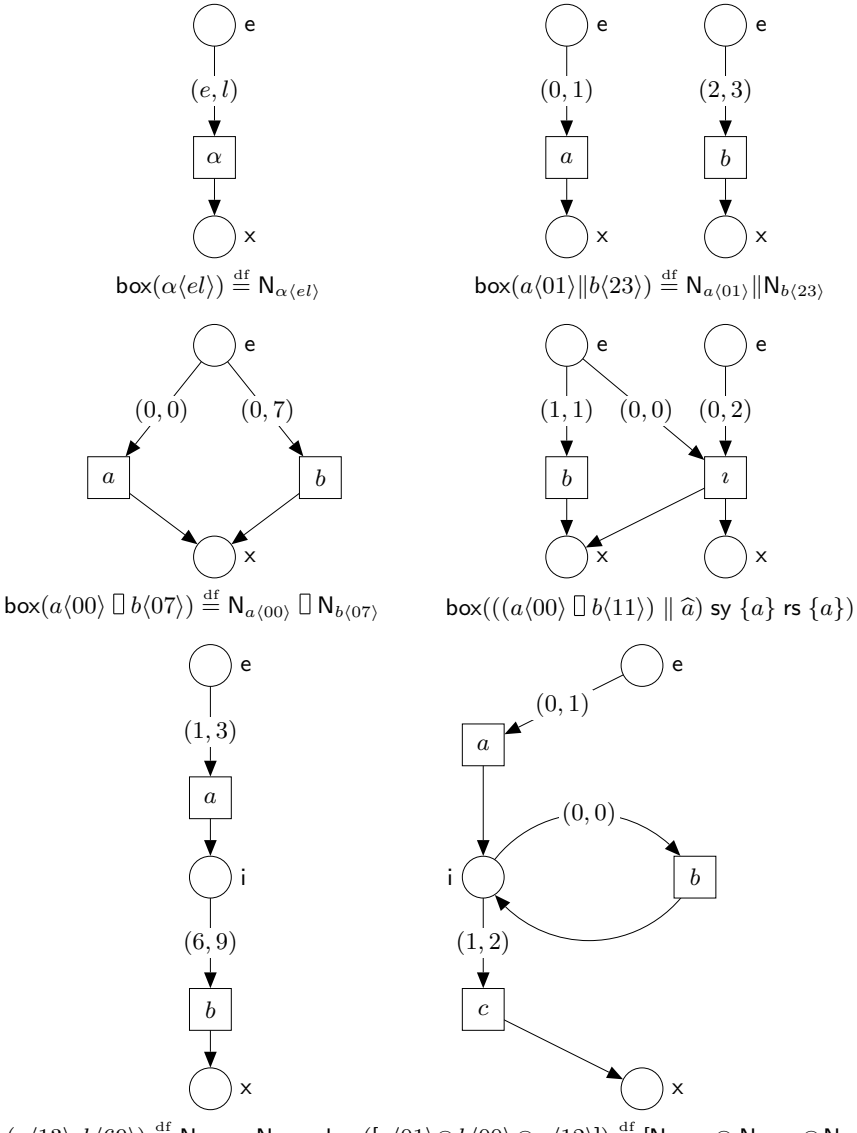


Fig. 4. Examples of nets defined in the algebra of at-boxes

at-box is defined as box(G) with $\mu(p)$ changed to 0, for every entry place p. In order to guarantee the safeness of the underlying PT-net, we follow the standard treatment of the PBC, by restricting slightly the syntax of the second component of the iterative construct $[D \circledast E \circledast F]$, by stipulating that each application of parallel composition is within the scope of some sequential composition. It is then not difficult to see that.

Proposition 3. *For every dynamic at-expression* $G = \overline{E}^{00}$*, the mapping* box *returns an at-box.*

Consistency Between Denotational and Operational Semantics. We now formulate the central result of this paper which states that the two semantics of at-expressions are equivalent. The following result extends that for the standard PBC where the transition systems of corresponding expressions and boxes are isomorphic [6].

Theorem 1. *For every dynamic at-expression* $G = \overline{E}^{00}$*, the reachability trees* RT_G *and* $\mathsf{RT}_{\mathsf{box}(G)}$ *are isomorphic.*

The first comment about the above theorem is that the result is not formulated in terms of *reachability graphs* of G and $\mathsf{box}(G)$, as in the standard PBC, but rather in terms of their reachability trees. The reason is that the latter are not isomorphic (though they are strongly bisimilar). Isomorphism of reachability graphs fails to hold because, in general, there is no one-to-one correspondence between the expressions reachable from G and the states reachable from the initial marking of $\mathsf{box}(G)$. To illustrate this, we consider the at-expression $G = \overline{((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); d\langle 01\rangle}^{00}$ for the the corresponding at-box $\mathsf{box}(G)$ is shown in figure 6. It may be easily checked that this net allows the following two sequences of moves, both starting from the initial state:

	scenario1		*scenario2*	
(1)	$(0,0,0,\perp,\perp,\perp,\perp)$	$[\{t_1,t_2\}\rangle$	$(0,0,0,\perp,\perp,\perp,\perp)$	$[\{t_1\}\rangle$
(2)	$(\perp,\perp,0,0,0,\perp,\perp)$	$[\surd\rangle$	$(\perp,0,0,0,\perp,\perp,\perp)$	$[\surd\rangle$
(3)	$(\perp,\perp,1,1,1,\perp,\perp)$	$[\{t_3\}\rangle$	$(\perp,1,1,1,\perp,\perp,\perp)$	$[\{t_2,t_3\}\rangle$
(4)	$(\perp,\perp,\perp,1,1,0,\perp)$	$[\{t_4\}\rangle$	$(\perp,\perp,\perp,1,0,0,\perp)$	$[\{t_4\}\rangle$
(5)	$(\perp,\perp,\perp,\perp,\perp,\perp,0)$		$(\perp,\perp,\perp,\perp,\perp,\perp,0)$	

The two corresponding execution sequences for the expression G are shown in figure 5. One may further observe that the left marking in line (4) above corresponds to the expressions in lines $(4')$ and $(4a')$, and that the right marking in line (4) above corresponds to the expressions in lines $(4'')$ and $(4a'')$. However, the two markings are different yet we have $(4') \equiv (4a') = (4a'') \equiv (4'')$, which indicates that the expressions in lines $(4', 4a', 4'', 4a'')$ represent the same state of the system. It is therefore impossible to show that the reachability graphs of G and $\mathsf{box}(G)$ are isomorphic. This should not be treated as a cause for concern since theorem 1 above still provides a strong relationship between the behaviours of the at-expressions and the corresponding at-boxes. The above discussions also shows that, in general, there can be no direct translation from dynamic at-expressions to at-boxes since, informally, there are fewer of the former than of the latter. In a way, as we already mentioned it, at-expressions are more *abstract* than the corresponding at-boxes. This, as we expect, can be used

to improve model-checking of behaviours specified by at-expressions, by providing an equivalence relation between reachable states of at-boxes which could be used to improve the efficiency of the unfolding of at-boxes (with the resulting unfoldings being smaller). This hypothesis is at the present moment investigated in the context of the general scheme for generating net unfoldings in [11] and the corresponding tool support.

The above discussion also means that a proof of theorem 1, cannot be obtained by a simple adaptation of that used in [6] since dynamic at-expressions cannot be unambiguously mapped to at-boxes. To explain how we cope with this problem, assume that we have an at-expression $G = \overline{E}^{00}$ not involving action restriction nor synchronisation, like that considered above. One can then make a crucial observation that for each transition t in $\mathsf{box}(G)$, the annotations of its input arcs are *exactly the same*, say (e, l) (this is, clearly, not true of at-boxes in general). This specific property implies that to check the enabledness of t it suffices to check that each input place to t has a token, and that the age of the oldest and the youngest token in such places lies between e and l. This is strictly less information than we require in the general case.

For every dynamic at-expression G, we call *clusters* $CL(G) = \{cl_1, \ldots, cl_n\}$ sets of places of $\mathsf{box}(G)$ which are compositionally defined and correspond to the entry/exit interfaces of $\mathsf{box}(G)$ as well as the input places of all individual transitions. This allows one to express the evolutions of $\mathsf{box}(G)$ in terms of changing the 'state' of clusters rather than the state of individual places. More precisely, the *cluster filling fc* of $\mathsf{box}(G)$ is a mapping which associates with each cluster either $\perp$ (meaning the cluster is empty), or $\mathbb{EL}0$ (meaning the cluster is partially filled, the age of the youngest token in it is $\mathbb{E}$, and the age of the oldest is $\mathbb{L}$), or $\mathbb{EL}1$ (meaning the cluster is completely filled, the age of the youngest token in it is $\mathbb{E}$, and the age of the oldest is $\mathbb{L}$). We then define enabledness of steps and dynamic changes of the net w.r.t. cluster based states, in a way quite similar to that used in the usual semantics, including the notion of a reachability tree. It can then be proven that *cluster-based at-boxes* are behaviorally equivalent to normal at-boxes (more precisely, their reachability trees are isomorphic).

Following the cluster definition, in the at-expression considered above, there are five clusters: $cl_1 \stackrel{\text{df}}{=} \{p_1\}$, $cl_2 \stackrel{\text{df}}{=} \{p_2\}$, $cl_3 \stackrel{\text{df}}{=} \{p_3\}$, $cl_4 \stackrel{\text{df}}{=} \{p_4, p_5, p_6\}$ and $cl_5 \stackrel{\text{df}}{=} \{p_7\}$. Assuming this ordering of clusters, our two scenarios can be rewritten as follows:

	scenario1		*scenario2*	
$(1''')$	$(001, 001, 001, \perp, \perp)$	$[\{t_1, t_2\}\rangle$	$(001, 001, 001, \perp, \perp)$	$[\{t_1\}\rangle$
$(2''')$	$(\perp, \perp, 001, 000, \perp)$	$[\surd\rangle$	$(\perp, 001, 001, 000, \perp)$	$[\surd\rangle$
$(3''')$	$(\perp, \perp, 111, 110, \perp)$	$[\{t_3\}\rangle$	$(\perp, 111, 111, 110, \perp)$	$[\{t_2, t_3\}\rangle$
$(4''')$	$(\perp, \perp, \perp, 011, \perp)$	$[\{t_4\}\rangle$	$(\perp, \perp, \perp, 011, \perp)$	$[\{t_4\}\rangle$
$(5''')$	$(\perp, \perp, \perp, \perp, 001)$		$(\perp, \perp, \perp, \perp, 001)$	

scenario1

$$
\begin{array}{lll}
(1') & \overline{((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); d\langle 01\rangle}^{\,00} & \xrightarrow{\{a^0,b^1\}} \\
(2') & ((\underline{a\langle 00\rangle}_{00} \parallel \underline{b\langle 01\rangle}_{00}) \parallel \overline{c\langle 11\rangle}^{\,00}); d\langle 01\rangle & \xrightarrow{\checkmark} \\
(3') & ((\underline{a\langle 00\rangle}_{11} \parallel \underline{b\langle 01\rangle}_{11}) \parallel \overline{c\langle 11\rangle}^{\,11}); d\langle 01\rangle & \xrightarrow{\{c^1\}} \\
(4') & ((\underline{a\langle 00\rangle}_{11} \parallel \underline{b\langle 01\rangle}_{11}) \parallel \underline{c\langle 11\rangle}_{00}); d\langle 01\rangle & \equiv \\
(4a') & ((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); \overline{d\langle 01\rangle}^{\,01} & \xrightarrow{\{d^0\}} \\
(5') & \underline{((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); d\langle 01\rangle}_{00} &
\end{array}
$$

scenario2

$$
\begin{array}{lll}
(1'') & \overline{((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); d\langle 01\rangle}^{\,00} & \xrightarrow{\{a^0\}} \\
(2'') & ((\underline{a\langle 00\rangle}_{00} \parallel \overline{b\langle 01\rangle}^{\,00}) \parallel \overline{c\langle 11\rangle}^{\,00}); d\langle 01\rangle & \xrightarrow{\checkmark} \\
(3'') & ((\underline{a\langle 00\rangle}_{11} \parallel \overline{b\langle 01\rangle}^{\,11}) \parallel \overline{c\langle 11\rangle}^{\,11}); d\langle 01\rangle & \xrightarrow{\{b^1,c^1\}} \\
(4'') & ((\underline{a\langle 00\rangle}_{11} \parallel \underline{b\langle 01\rangle}_{00}) \parallel \underline{c\langle 11\rangle}_{00}); d\langle 01\rangle & \equiv \\
(4a'') & ((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); \overline{d\langle 01\rangle}^{\,01} & \xrightarrow{\{d^0\}} \\
(5'') & \underline{((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); d\langle 01\rangle}_{00} &
\end{array}
$$

Fig. 5. Two execution sequences corresponding to scenario 1 and 2

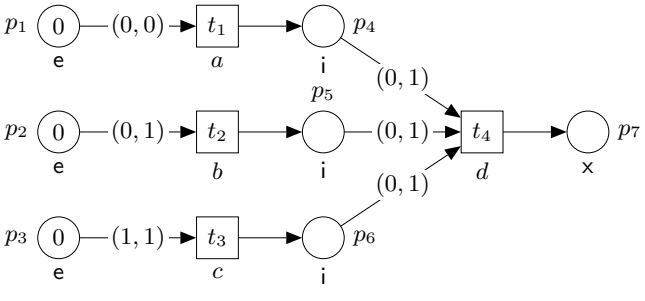


Fig. 6. An at-box corresponding to the expression $\overline{((a\langle 00\rangle \parallel b\langle 01\rangle) \parallel c\langle 11\rangle); d\langle 01\rangle}^{\,00}$

Note that the problem encountered before with line (4) is no longer present in line (4‴). Effectively, this means that we can suitably adopt the proof technique used in, e.g., [6], to justify theorem 1. Finally, synchronised transitions will have two clusters which are responsible for their enabling, and restriction will remove transitions, but the corresponding clusters will still be retained.

6 Concluding Remarks

In this paper, we introduced a new compositional model of arc-based time Petri nets, and a corresponding process algebra of time expressions. We have explained the nature of the correspondence between the two algebras, in terms of their respective reachability trees, and outlined an intermediate (cluster based) representation used in the proof of this correspondence. In particular, these results make it possible to combine the verification techniques developed independently for process algebra and Petri nets with timing, and to give a syntax oriented semantics of real-time specification languages. Finally, we also plan to explore more efficient model checking technique based on the observations made in this paper.

References

1. P. A. Abdulla and N. Aletta: Timed Petri Nets and BQOs. Proc. of *ICATPN'01*, J.-M. Colom, M. Koutny (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2075 (2001) 53-70.
2. J. Baeten and W. P. Weijland: *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press (1990).
3. B. Berthomieu and M. Diaz: Modelling and verification of Time Dependent Systems Using Time Petri Nets. *IEEE Trans. on Soft. Eng.* 17 (1991) 259-273.
4. E. Best, R. Devillers and J. Hall: The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. In: *Advances in Petri Nets*, G. Rozenberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 609 (1992) 21-69.
5. E. Best, R. Devillers and M. Koutny: A Unified Model for Nets and Process Algebras. In: *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, S. A. Smolka, (Eds.). Elsevier (2001) 873–944.
6. E. Best, R. Devillers and M. Koutny: *Petri Net Algebra*. EATCS Monographs on TCS, Springer (2001).
7. B. Bieber and H. Fleischhack: Model Checking of Time Petri Nets Based on Partial Order Semantics. Proc. of *CONCUR'99*, J. Baeten and S. Mauw (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1664 (1999) 210-225.
8. T. Bolognesi, F. Lucidi and S. Trigila: From Timed Petri Nets to timed LOTOS. Proc. of *PSTV'90*, L. Logrippo et al. (Eds.). North-Holland (1990) 395-408.
9. C. A. R. Hoare: *Communicating Sequential Processes*. Prentice Hall (1985).
10. R. Janicki and P. E. Lauer: *Specification and Analysis of Concurrent Systems - the COSY Approach*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1992).
11. V. Khomenko, M. Koutny and W. Vogler: Canonical Prefixes of Petri Net Unfoldings. *Acta Informatica* 40 (2003) 95-118.
12. M. Koutny: A Compositional Model of Time Petri Nets. Proc. of *ICATPN'00*, M. Nielsen and D. Simpson (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1825 (2000) 303-322.
13. O. Marroquin Alonso and D. de Frutos-Escrig: Computing a Finite Prefix of a Time Petri Net. Proc. of *ICATPN'01*, J.-M. Colom and M. Koutny (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2075 (2001) 303-322.
14. P. Merlin and D. Farber: Recoverability of Communication Protocols - Implication of a Theoretical Study. *IEEE Trans. on Soft. Comm.* 24 (1976) 1036-1043.

15. R. Milner: *Communication and Concurrency*. Prentice Hall (1989).
16. T. Murata: Petri Nets: Properties, Analysis and Applications. *Proc. of IEEE* 77 (1989) 541-580.
17. M. Nielsen, V. Sassone and J. Srba: Properties of Distributed Timed-Arc Petri Nets. Proc. of *FSTTCS'01*, R. Hariharan, M. Mukund, V. Vinay (Eds.). Springer-Verlag, Lecture Notes in Computer Science 2245 (2001) 280-285.
18. G. D. Plotkin: A Structural Approach to Operational Semantics. Technical Report FN-19, Computer Science Department, University of Aarhus (1981).
19. L. Popova: Time Petri Nets. *Journal of Information Processing and Cybernetics EIK* 27 (1991) 227-244.
20. W. Reisig: *Petri Nets. An Introduction*. EATCS Monographs on Theoretical Computer Science, Springer-Verlag (1985).
21. A. W. Roscoe: *The Theory and practice of Concurrency*. Prentice-Hall (1998).

A Calculus for Shapes in Time and Space

Andreas Schäfer

Department of Computing Science,
University of Oldenburg, 26111 Oldenburg, Germany
schaefer@informatik.uni-oldenburg.de

Abstract. We present a spatial and temporal logic based on Duration Calculus for the specification and verification of mobile real-time systems. We demonstrate the use of the formalism and apply it to a case study. We extend a pure Duration Calculus specification for the controller by spatial assumptions to reason about spatial system properties. We prove that the formalism is undecidable in general for discrete and continuous domains and present a decidable fragment.

Keywords: Real-time systems, mobile systems, spatial logic, temporal logic, Duration Calculus.

1 Introduction

There are many well understood formal techniques for the specification and verification of real-time systems, among them the interval temporal logic Duration Calculus (DC) [ZHR91, HZ04] or Timed Automata [AD94]. But often problems of safety critical real-time systems have a spatial nature. Consider for example two trams driving on the same track. This situation itself is not dangerous as long as they do not "overlap" or as long as the distance between two trams is sufficiently large. Another example is a robot moving around in a restricted area. A desirable safety property might be that the robot moves at most 5 cm outside its area so that it does not endanger staff.

These properties cannot be expressed concisely and nicely in standard temporal logics. This led us to the idea to extend a well known formalism for real-time systems to be able to describe spatial properties when needed. The use of the formalism should be similar to the use of pure temporal logics when no spatial reasoning is required.

In this paper we describe how to extend the temporal interval logic Duration Calculus [ZHR91] to a spatio-temporal interval logic. Instead of having one dimension for time, we allow an arbitrary number of dimensions from which a subset is considered to be spatial and a subset to be temporal. From the point of view of our formalism, it does not matter whether a dimension is temporal or spatial. If we just chose one (temporal) dimension, we get the normal Duration Calculus.

In section 2 we introduce the formalism and show in section 3 how it can be used to describe spatial properties such as movement of objects in space. In

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 463–477, 2005.
© Springer-Verlag Berlin Heidelberg 2005

section 4 we show how a controller design done in pure Duration Calculus can be extended to reason about the spatial properties of the whole system.

In section 5 we prove that the formalism is undecidable for dense and discrete time/space-domains before giving a decidable fragment in section 6.

2 Shape Calculus

In this section we define the shape calculus and show for each feature that the extension is conservative and we do not add expressive power to Duration Calculus if we only consider one dimension.

In Duration Calculus the behaviour of a system is modelled by a set of time-dependant variables whose value change in time. The natural extension for a spatial and temporal logic is to choose the dimension for space and for time and to use flexible variables whose value depend on the point in time and space. According to DC terminology we will call these variables *observables* and let *Obs* denote the set of all observables. The semantics of an observable X is given by a function $\mathcal{I}$

$$\mathcal{I}[\![X]\!] : \mathbb{R}^n_{\geq 0} \to \{d_0, \ldots, d_m\}$$

where $\{d_0, \ldots, d_m\}$ denotes the range of the observable. To guarantee that the integral exisits for this function, we require $\mathcal{I}[\![X]\!]$ to be continuous almost everywhere. If we choose $n = 1$ we end up with the Duration Calculus definition.

State Expressions. Properties of a point in space and in time are expressed by *state formulae* denoted by π and build from comparison of observables and boolean connectors.

$$\pi ::= X = d \,|\, \neg\pi_1 \,|\, \pi_1 \wedge \pi_2$$

where d is in the range of the observable X. The semantics is then given by a function

$$\mathcal{I}[\![\pi]\!] : \mathbb{R}^n_{\geq 0} \to \{0, 1\}$$

which is defined in the expected way. The definition is exactly like in Duration Calculus. If the observable X is of boolean type, we will write X for $X = 1$.

Terms. Interval temporal logic assigns a real value to every interval and Duration Calculus introduces the integral operator to measure the duration of a certain state in a given interval.

Instead of intervals we use bounded polyhedra for the n-dimensional case. Alternatively we could have used hypercubes, but as we shall see in the examples, specifications can be done much more conveniently with polyhedra. The set of bounded polyhedra will be denoted by *Poly*. One dimensional polyhedra are intervals.

In many cases, we are not interested in the spatio-temporal volume, but only in a spatial or a temporal measure. For example, it might be important how much of a vehicle is outside its working area. In this case we need an integral

for the spatial part. On the other hand we might be interested in the amount of time the system is moving, which is a temporal integral.

To this end, we allow linear transformations T of the function $\mathcal{I}[\![X]\!]$ before applying the integral. With these transformations we can for example achieve projections on all axes or on hyperplanes. We define the set of *terms* as follows:

$$\theta ::= \int \boldsymbol{T}\pi \mid x \mid \boldsymbol{d}.i \mid f(\theta_1, \ldots, \theta_k)$$

where π is a state assertion, $\boldsymbol{T}$ is a $m \times n$ matrix of real numbers, x a rigid variable of type real, $\boldsymbol{d}$ a variable whose type is a n-dimensional vector of reals, and f a function symbol, which can be an arithmetic function like $+$. As usual, the value of the rigid variable is determined by a valuation $\mathcal{V}$ and the set of valuations is denoted by Val. The value of a vector $\boldsymbol{d}$ is also given by the valuation and the components of $\boldsymbol{d}$ are accessed by indexing. So $\boldsymbol{d}.i$ returns the ith component of $\boldsymbol{d}$.

The semantics of terms is a function

$$I[\![\theta]\!] : Poly \times Val \to \mathbb{R}$$

and in particular the semantics of the integral of a state assertion π after a transformation T is defined using the characteristic function χ. This idea is sketched in figure 1 (a)-(c).

$$I[\![\textstyle\int \boldsymbol{T}\pi]\!](\mathcal{M}, \mathcal{V}) = \displaystyle\int_{\mathcal{M}} \chi_{\boldsymbol{T}(\mathcal{M} \cap \mathcal{I}[\![\pi]\!]^{-1}(1))}$$

where

$$\chi_{\boldsymbol{T}(\mathcal{M} \cap \mathcal{I}[\![\pi]\!]^{-1}(1))} = \begin{cases} \mathbb{R}^m \to \mathbb{B} \\ \boldsymbol{x} \mapsto \begin{cases} 1 & \text{if } \exists \boldsymbol{x}' \in \mathcal{M} : \boldsymbol{x} = \boldsymbol{T}\boldsymbol{x}' \wedge \mathcal{I}[\![\pi]\!](\boldsymbol{x}') = 1 \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

For the 1-dimensional case, the linear transformation $\boldsymbol{T}$ is only scaling and $\int \boldsymbol{T}\pi = \boldsymbol{T} \int \pi$ holds. Thus in this case the transformation does not add expressive power and we still have the Duration Calculus.

Formulae. In temporal interval logic, the intervals can be "chopped" into a leftmost and a rightmost subinterval. In our case we introduce the chop operator $\langle \boldsymbol{d} \rangle$ to split the polyhedron along a given hyperplane, which results in two new sub-polyhedra. This is illustrated in figure 1 (d).

The set of *formulae* is given by

$$F ::= F_1 \langle \boldsymbol{d} \rangle F_2 \mid p(\theta_1, \ldots, \theta_k) \mid \neg F_1 \mid F_1 \wedge F_2 \mid \exists x : F \mid \exists \boldsymbol{d} : F$$

where p is a predicate symbol, x a rigid variable and $\boldsymbol{d}$ a rigid vector. The other boolean connectives can be defined as the usual abbreviations.

The semantics is a function

$$\mathcal{I}[\![F]\!] : Poly \times Val \to \mathbb{B}.$$

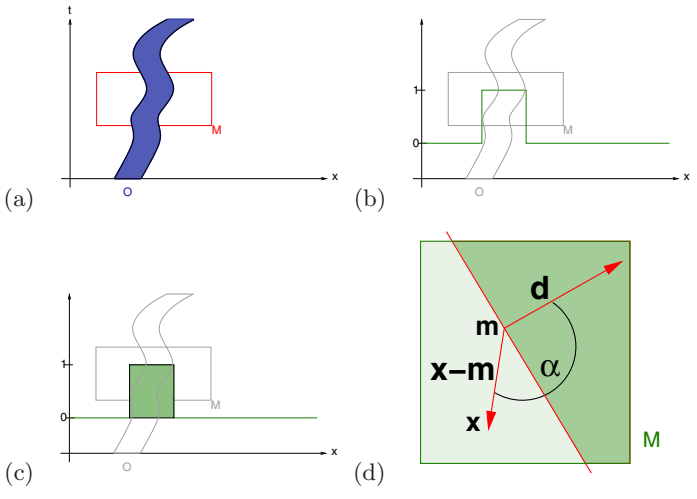


Fig. 1. (a) Function $\mathcal{I}[\![O]\!]$ with polyhedron $\mathcal{M}$, (b) the characteristic function $\chi_{\boldsymbol{T}(\mathcal{M}\cap\mathcal{I}[\![O]\!]^{-1}(1))}$ where $\boldsymbol{T} = e_x^T = (1,0)$ is the transposed unit vector for the spatial dimension. This is the projection onto the x-axis and (c) the integral $\int \chi_{\boldsymbol{T}(\mathcal{M}\cap\mathcal{I}[\![O]\!]^{-1}(1))}$. (d) illustrates the chop-operation

A formula $F_1 \langle \boldsymbol{d} \rangle F_2$ is valid if and only if there is a hyperplane orthogonal to the vector $\boldsymbol{d}$ such that the polyhedron $\mathcal{M}$ is split by this hyperplane into two polyhedra $\mathcal{M}_1$ and $\mathcal{M}_2$ which fulfil F_1 and F_2, respectively. So

$$\mathcal{I}[\![F_1 \langle \boldsymbol{d} \rangle F_2]\!](\mathcal{M}, \mathcal{V}) = true$$

iff there exists a $\boldsymbol{m} \in \mathcal{M}$ such that with $\mathcal{M}_1 \stackrel{df}{=} \{\boldsymbol{x} \in \mathcal{M} | \langle \boldsymbol{x} - \boldsymbol{m}, \boldsymbol{d} \rangle \leq 0\}$ and $\mathcal{M}_2 \stackrel{df}{=} \{\boldsymbol{x} \in \mathcal{M} | \langle \boldsymbol{x} - \boldsymbol{m}, \boldsymbol{d} \rangle \geq 0\}$ the following holds:

$$\mathcal{I}[\![F_1]\!](\mathcal{M}_1, \mathcal{V}) = true \text{ and } \mathcal{I}[\![F_2]\!](\mathcal{M}_2, \mathcal{V}) = true$$

Here $\langle \boldsymbol{x} - \boldsymbol{m}, \boldsymbol{d} \rangle$ denotes the scalar product of $\boldsymbol{x} - \boldsymbol{m}$ and $\boldsymbol{d}$ which is proportional to the cosine of the angle α between these vectors. Thus, it is negative iff α is greater than 180 degrees, i.e., the point $\boldsymbol{x}$ is below and positive otherwise. Note that the scalar product is bilinear. So scaling the vector $\boldsymbol{d}$ with positive reals does not change $\mathcal{M}_1$ or $\mathcal{M}_2$.

In the 1-dimensional case, when the vector used for the chop operation has only one dimension, there are only three different cases possible. All of them can be modelled in Duration Calculus using the DC chop operator ";" and conjunction.

$$\begin{aligned} F \langle 1 \rangle G &\iff F; G \\ F \langle 0 \rangle G &\iff F \wedge G \\ F \langle -1 \rangle G &\iff G; F \end{aligned}$$

So we have the conclusion.

Corollary 1. *For $n = 1$ the shape calculus and the Duration Calculus coincide.*

Note 1. The chop operation is associative for the same vector $\boldsymbol{d}$ but not for different vectors

$$(F \langle \boldsymbol{d} \rangle G) \langle \boldsymbol{d} \rangle H \iff F \langle \boldsymbol{d} \rangle (G \langle \boldsymbol{d} \rangle H)$$

but in general if $\boldsymbol{d_1}$ is not a multiple of $\boldsymbol{d_2}$

$$(F \langle \boldsymbol{d_1} \rangle G) \langle \boldsymbol{d_2} \rangle H \not\iff F \langle \boldsymbol{d_1} \rangle (G \langle \boldsymbol{d_2} \rangle H).$$

2.1 Abbreviations

Duration Calculus defines a lot of abbreviations to ease the handling of specifications. We adopt them directly.

$$\ell \stackrel{df}{=} \int 1 \qquad \ell_{\boldsymbol{T}} \stackrel{df}{=} \int \boldsymbol{T} 1 \qquad \ell_{\boldsymbol{d}} \stackrel{df}{=} \int (\frac{1}{\|\boldsymbol{d}\|} \boldsymbol{d}) 1$$

can be used to measures the size or diameter of the polyhedron. We denote by e_x and e_t the unit vectors for the x- respectively time dimension, by e_x^T respectively e_t^T their transposition, and we write $\ell_t \stackrel{df}{=} \int e_t^T 1$ to measure the size along the time axis and $\ell_x \stackrel{df}{=} \int e_x^T 1$ for the size along the x-axis.

Like in Duration Calculus, the chop along the time axis is written

$$F; G \stackrel{df}{=} F \langle \boldsymbol{e_t} \rangle G.$$

The everywhere operator $\lceil \pi \rceil$ expresses that a state assertion π holds almost everywhere in the polyhedron. It can be augmented by a transformation $\boldsymbol{T}$.

$$\lceil \pi \rceil \stackrel{df}{=} \int \pi = \ell \wedge \ell > 0 \qquad \lceil \pi \rceil_{\boldsymbol{T}} \stackrel{df}{=} \int \boldsymbol{T} \pi = \ell_{\boldsymbol{T}} \wedge \ell_{\boldsymbol{T}} > 0.$$

A polyhedron of length zero is also denoted by

$$\lceil \rceil_{\boldsymbol{T}} \stackrel{df}{=} \ell_{\boldsymbol{T}} = 0.$$

The somewhere operator $\Diamond F$ allows the polyhedron to be chopped twice in the same direction such that in the middle polyhedron F holds.

$$\Diamond_{\boldsymbol{d}} F \stackrel{df}{=} true \langle \boldsymbol{d} \rangle F \langle \boldsymbol{d} \rangle true$$

with the dual globally operator $\Box$

$$\Box_{\boldsymbol{d}} \stackrel{df}{=} \neg \Diamond_{\boldsymbol{d}} \neg F.$$

Note 2. Although the chop operation is not associative for different directions, the $\Diamond$ operation commutes

$$\Diamond_{\boldsymbol{d_1}} \Diamond_{\boldsymbol{d_2}} F = \Diamond_{\boldsymbol{d_2}} \Diamond_{\boldsymbol{d_1}} F$$

and because of duality also the $\Box$ operation does

$$\Box_{\boldsymbol{d_1}} \Box_{\boldsymbol{d_2}} F = \Box_{\boldsymbol{d_2}} \Box_{\boldsymbol{d_1}} F.$$

3 Application

Using the proposed framework, we can easily describe movement and connectivity of objects in space. Apart from that, we can describe and reason about the control of these objects in this framework, using the Duration Calculus part, as this is just a conservative extention. At first we present some examples describing how objects move in space. Assume one spatial dimension x and one temporal dimension t. We consider an object O which is modelled by an observable O such that the observable has value 1 for a point in time and space iff the object occupies this point in space for this point in time.

3.1 STOP

The simplest expression is that an object O with size s does not move in space.

$$\mathsf{STOP}(O,s) \stackrel{df}{=} (\lceil\neg O\rceil \vee \lceil\rceil_{e_x})\ \langle e_x\rangle\ (\lceil O\rceil \wedge \ell_x = s)\ \langle e_x\rangle\ (\lceil\neg O\rceil \vee \lceil\rceil_{e_x})$$

The idea of this formula is sketched in figure 2 and it reads as follows. The polyhedron (here it is a square) can be partitioned into three parts along the x-direction. In the first part, everywhere is not object O or it has size zero in x-direction. In the second part, O is true everywhere and this part has size s in x-direction. So O has constant size s over time. And in the third part, there is again no object O or it has size zero. As we chop the square orthogonal to x-axis the position of O must be constant.

3.2 Continuous Movement

The STOP formula can be generalised to describe continuous movement with velocity v. Here we chop orthogonal to the velocity vector and obtain three sub-polyhedra such that only in the middle one O is true everywhere. This is sketched in figure 2 (b).

$$\mathsf{MOTION}(O,v) \stackrel{df}{=} (\lceil\neg O\rceil \vee \lceil\rceil_{\binom{1}{-v}})\ \langle\tbinom{1}{-v}\rangle\ \lceil O\rceil\ \langle\tbinom{1}{-v}\rangle\ (\lceil\neg O\rceil \vee \lceil\rceil_{\binom{1}{-v}})$$

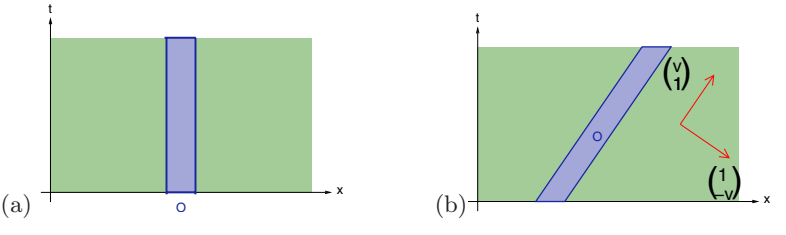


Fig. 2. (a) The object O remains immobile (b) The object O moves continously, according to the pattern MOTION

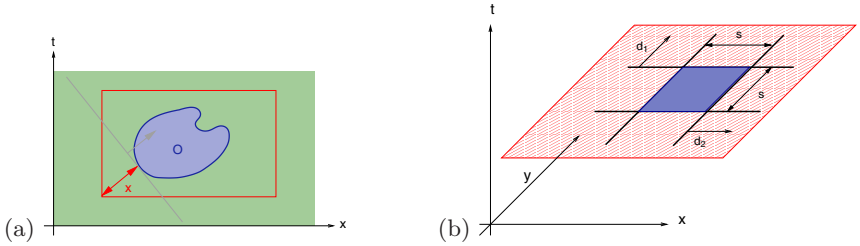


Fig. 3. (a) The object O has distance x in direction $\boldsymbol{d}$ from beginning of the polyhedron. (b) The object is a square

This does not specify where the object O starts its movement. A starting position can be defined by adding an additional constraint to MOTION, for example

$$\mathsf{MOTION}_0(O, v) \stackrel{df}{=} \mathsf{MOTION}(O, v) \wedge \neg(\lceil\neg O\rceil_x \langle e_x\rangle \mathit{true})$$

requires the object to start at point 0 on the x-axis.

3.3 Position of Objects

By taking the position of object O at a certain point of time, it is possible to define arbitrary movements. This is captured by

$$\mathsf{POS}_{\boldsymbol{d}}(O, x) \stackrel{df}{=} ((\lceil\neg O\rceil \vee \lceil\rceil_{\boldsymbol{d}}) \wedge \ell_{\boldsymbol{d}} = x) \langle \boldsymbol{d}\rangle \lceil O\rceil_{\boldsymbol{d}} \langle \boldsymbol{d}\rangle \mathit{true}.$$

In this formula we define x to be the maximal size in direction $\boldsymbol{d}$ of a sub-polyhedron in which O is not true. This is sketched in figure 3 (a).

3.4 Generalisation to More Than One Dimension

The MOTION formula given above easily generalises to more than one dimension. Assume that the movement of the object is defined by the vector $\boldsymbol{v} = (v_x, v_y, 1)^T$. Then

$$\mathsf{MOTION}_{3d}(O, \boldsymbol{v}) \stackrel{df}{=} \forall \boldsymbol{d}.\langle \boldsymbol{d}, \boldsymbol{v}\rangle = 0 \Rightarrow (\lceil\neg O\rceil \vee \lceil\rceil_{\boldsymbol{d}}) \langle \boldsymbol{d}\rangle (\neg \Diamond_{\boldsymbol{d}} \lceil\neg O\rceil) \langle \boldsymbol{d}\rangle (\lceil\neg O\rceil \vee \lceil\rceil_{\boldsymbol{d}})$$

describes the movement of an object in the plane, where $\langle \boldsymbol{d}, \boldsymbol{v}\rangle$ denotes the scalar product of $\boldsymbol{d}$ and $\boldsymbol{v}$.

3.5 Shape of Objects

The shape of objects at points in time can also be grasped. For example

$$\mathsf{Circle}(O, s) \stackrel{df}{=} \Box_t\big(\lceil\rceil_t \Rightarrow (\forall \boldsymbol{d}.(\lceil\neg O\rceil_{\boldsymbol{d}} \langle \boldsymbol{d}\rangle \lceil O\rceil_{\boldsymbol{d}} \wedge \ell_{\boldsymbol{d}} = s \langle \boldsymbol{d}\rangle \lceil\neg O\rceil_{\boldsymbol{d}}))\big)$$

requires that for every point in time, in the hyperplane which belongs to this point the object has size s in every possible direction. This is only true if the

object is a circle of diameter s and it does neither disappear nor does appear a second object of this kind.

The property that the object is a square with size s can be defined by the following formula. In figure 3 (b) we depict the idea of this definition.

$$\begin{aligned}\mathsf{Square}(O,s) \stackrel{df}{=} \Box_t\big(\lceil\rceil_t \Rightarrow (\exists \boldsymbol{d_1},\boldsymbol{d_2}.(\langle \boldsymbol{d_1},\boldsymbol{d_2}\rangle = \langle \boldsymbol{d_1},\boldsymbol{e}_t\rangle = \langle \boldsymbol{d_2},\boldsymbol{e}_t\rangle = 0 \wedge \\ (\lceil\neg O\rceil_{(e_x,e_y)^T} \\ \langle \boldsymbol{d_1}\rangle \\ \quad (\lceil\neg O\rceil_{(e_x,e_y)^T} \\ \quad \langle \boldsymbol{d_2}\rangle \lceil O\rceil_{(e_x,e_y)^T} \wedge \ell_{\boldsymbol{d_1}} = \ell_{\boldsymbol{d_2}} = s \\ \quad \langle \boldsymbol{d_2}\rangle \lceil\neg O\rceil_{(e_x,e_y)^T}) \\ \langle \boldsymbol{d_1}\rangle \\ (\lceil\neg O\rceil_{(e_x,e_y)^T}))))\big).\end{aligned}$$

4 Case Study: The Moving Robot (Roadrunner)

Now we apply the ideas of the previous section and specify a system using two spatial and one temporal dimensions. We consider a robot moving on an rectangular plateau. Assume that the robot is equipped with sensors all around to detect the edges. We do not consider the special task of the robot, but only model its movement and edge detection system. It is to be proven that the robot does not fall off the plateau.

Controller. A simple controller for the vehicle may consist only of two states and is sketched in figure 4. One state is *run*, in which the robot is moving forward. When the robot detects an edge, after a response time of δ_R ms, the robots starts to turn around for δ_T ms to rotate by angle π, before returning to state *run*.

In order to derive an implementation from the specification, a subset of DC called the DC-Implementables introduced by Ravn [Rav95] is important. We use the implementables to specify the controller and thus need to introduce the following abbreviation:

$$F \longrightarrow \lceil\pi\rceil \stackrel{df}{=} \neg\Diamond_t(F;\lceil\neg\pi\rceil)$$

With this abbreviation we can define the controller in the following way. Initially, the controller is in state *run*.

$$\lceil run\rceil \vee \lceil\rceil_t \qquad \text{(init-control)}$$

From state *run*, the controller can evolve to state *turn*.

$$\lceil run\rceil \rightarrow \lceil run \vee turn\rceil \qquad \text{(successor-run)}$$

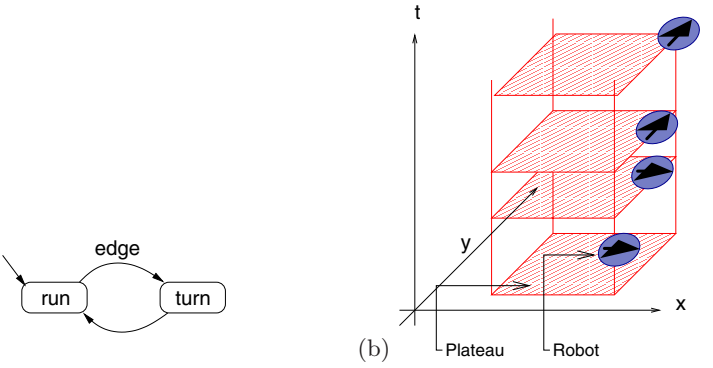


Fig. 4. (a) A simple controller for the roadrunner. (b) A possible unsafe run

From state *turn* it can evolve to state *run*.

$$\lceil turn \rceil \rightarrow \lceil run \vee turn \rceil \qquad \text{(successor-run)}$$

State *run* must be left if an edge is detected after at most δ_R time units.

$$\lceil run \wedge edge \rceil \wedge \ell = \delta_R \rightarrow \lceil \neg run \rceil \qquad \text{(progress-run)}$$

State *turn* must be left after δ_T time units.

$$\lceil turn \rceil \wedge \ell = \delta_T \rightarrow \lceil \neg turn \rceil \qquad \text{(progress-turn)}$$

Spatial Behaviour. To specify the spatial behaviour of the robot we introduce two boolean oberservables RR – for roadrunner – and P – for plateau –. The assumptions are defined using the abbreviations given in the last paragraph.

1. In state *run*, the robot moves with constant velocity v in one direction:

$$\Box_t(\lceil run \rceil \Rightarrow \exists \boldsymbol{d}.(\mathsf{MOTION}_{3d}(RR, d) \wedge \boldsymbol{d}.1^2 + \boldsymbol{d}.2^2 = v \wedge \boldsymbol{d}.3 = 1))$$

2. In state *turn* the robot turns around its axis. We assume that the function f computes the new direction $\boldsymbol{d_2}$ given the old direction $\boldsymbol{d_1}$ and the time which is spent in state *turn*.

$$\Box_t((\lceil run \rceil \wedge \mathsf{Motion}_{3d}(RR, \boldsymbol{d_1}); \lceil turn \rceil \wedge \ell_t = t; \lceil run \rceil \wedge \mathsf{Motion}_{3d}(RR, \boldsymbol{d_2})) \Rightarrow \boldsymbol{d_2} = f(\boldsymbol{d_2}, t))$$

3. The shape of the robot is a circle with unit diameter.

$$\mathsf{Circle}(RR, 1)$$

4. The edge detection is activated as soon as robot leaves the plateau.

$$\Box_t \lceil RR \wedge \neg P \rceil \rightarrow \lceil edge \rceil$$

4.1 Environment

We need to specify that the plateau P is rectangular and does neither change its shape nor moves beneath the robot.

$$\lceil \neg P \rceil \langle \boldsymbol{e_x} \rangle (\lceil \neg P \rceil \langle \boldsymbol{e}_y \rangle \lceil P \rceil \langle \boldsymbol{e}_y \rangle \lceil \neg P \rceil) \langle \boldsymbol{e}_x \rangle \lceil \neg P \rceil$$

Initially, the roadrunner is in the centre of the plateau and the length of the plateau is 21 units. For better readability we omitted the parentheses.

$$
\begin{aligned}
&\lceil \rceil_t \wedge \\
&\lceil \neg(P \wedge RR) \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \\
&\langle \boldsymbol{e_x} \rangle \\
&\quad \lceil \neg(P \wedge RR) \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \\
&\quad \langle \boldsymbol{e_y} \rangle \\
&\qquad \lceil P \wedge \neg RR \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \wedge \ell_x = 10 \\
&\qquad \langle \boldsymbol{e_x} \rangle \\
&\qquad\quad \lceil P \wedge \neg RR \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \wedge \ell_y = 10 \\
&\qquad\quad \langle \boldsymbol{e_y} \rangle \\
&\qquad\quad \lceil P \wedge RR \rceil_{(\boldsymbol{e}_x)^T} \wedge \lceil P \wedge RR \rceil_{(\boldsymbol{e}_y)^T} \wedge \ell_x = 1 \wedge \ell_y = 1 \\
&\qquad\quad \langle \boldsymbol{e_y} \rangle \\
&\qquad\quad \lceil P \wedge \neg RR \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \wedge \ell_y = 10 \\
&\qquad \langle \boldsymbol{e_x} \rangle \\
&\qquad \lceil P \wedge \neg RR \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \wedge \ell_x = 10 \\
&\quad \langle \boldsymbol{e_y} \rangle \\
&\quad \lceil \neg(P \wedge RR) \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \\
&\langle \boldsymbol{e_x} \rangle \\
&\lceil \neg(P \wedge RR) \rceil_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T}) \\
&; \mathsf{true}
\end{aligned}
$$

4.2 Requirement

If we assume, that the robot is safe if at most 50 % of its area is outside the plateau, the safety-requirement is specified by

$$\Box_t \textstyle\int (RR \wedge \neg P)_{(\boldsymbol{e}_x, \boldsymbol{e}_y)^T} \leq 0.5.$$

Verification. Trying to verify the safety property, it turns out that the system is not safe even if we chose the velocity v to be low enough such that the robot can stop the motors in its response time. The erroneous situation is sketched in

figure 4. If we assume the upper bound for the response time δ_B to be 500 ms and the velocity v to be $v = 1m/s$ then less than half of the robot is not on the plateau, after having detected an edge. If this just happens in a corner of the plateau and the robot turns 90 degree and continuous to move in the wrong direction, after the response time of 500 ms $\frac{3}{4}$ of the robot will be off the plateau.

Such a behaviour could for example be avoided by driving backwards for δ_B before turning around.

5 Undecidability of Satisfiability

As Duration Calculus with dense time domain is undecidable in general [HZ97] and the shape calculus with a zero dimensional space is exactly the Duration Calculus, it is not decidable either.

However, a restricted subclass of Duration Calculus is decidable with discrete time domain [HZ97], but this does not hold for the shape calculus with more than 1 dimensions. We prove this by reduction from the emptiness problem for 2-dimensional tiling systems [GR97].

5.1 Tiling-Systems and Encoding of Turing Machine Computations

A tile is a 2×2 matrix whose elements are taken from a given alphabet $\Sigma \cup \{\#\}$ where $\#$ is a special symbol which is not included in the set Σ. For a set of tiles Θ the local language $L(\Theta)$ is the set of all $m \times n$ matrices M such that every 2×2 block of M is in Θ. Note, that the blocks are overlapping. Additionally all four borders of this matrix must only consist of the $\#$ symbol. So the matrix must be framed by $\#$. In [GR97] it is shown that the following problem is undecidable: "Given a set of tiles Θ, is $L(\Theta)$ empty?".

For the proof, a Turing Machine computation is simulated in a 2-dimensional array as sketched in figure 5. Each row represents a configuration of the Turing Machine and the row below its successor configuration.

5.2 Encoding Tilings in Shape Calculus

For a set of tiles $\Theta = \{p_1, \ldots, p_k\}$ we can give a formula F_Θ in shape calculus, such that $L(\Theta) \neq \emptyset$ iff F_Θ is satisfiable. To this end, every tile

$$p_i = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

is mapped to a formula

$$\begin{aligned} F_{p_i} = &((\lceil a \rceil \wedge \ell_{\boldsymbol{e_1}} = 1 \wedge \ell_{\boldsymbol{e_2}} = 1) \langle \boldsymbol{e_2} \rangle (\lceil b \rceil \wedge \ell_{\boldsymbol{e_1}} = 1 \wedge \ell_{\boldsymbol{e_2}} = 1)) \langle \boldsymbol{e_1} \rangle \\ &((\lceil c \rceil \wedge \ell_{\boldsymbol{e_1}} = 1 \wedge \ell_{\boldsymbol{e_2}} = 1) \langle \boldsymbol{e_2} \rangle (\lceil d \rceil \wedge \ell_{\boldsymbol{e_1}} = 1 \wedge \ell_{\boldsymbol{e_2}} = 1)). \end{aligned}$$

$\# \ldots \ \# \ \# \ \# \ \ \# \ \ \# \ \ \# \ \ldots \ \#$	
$\# \ \sqcup \ \ldots \ \sqcup \ q_0 \ \ w_1 \ \ w_2 \ \ w_3 \ \ldots \ \#$	(initial configuration)
$\# \ \sqcup \ \ldots \ \sqcup \ w_1^1 \ \ q^1 \ \ w_2^1 \ \ w_3^1 \ \ldots \ \#$	(2nd configuration)
$\# \ \sqcup \ \ldots \ \sqcup \ q^2 \ \ w_1^2 \ \ w_2^2 \ \ w_3^2 \ \ldots \ \#$	(3rd configuration)
$\# \ldots$	
$\# \ \sqcup \ \ldots \quad q_f \ \ w_1^m \ \ w_2^m \ \ w_3^m \ \ldots \ \#$	(final configuration)
$\# \ldots \ \# \ \# \ \# \ \ \# \ \ \# \ \ \# \ \ldots \ \#$	

Fig. 5. Encoding of Turing Machine computation in a 2-dimensional picture

With these sub-formulae we define F_Θ to be

$$
\begin{aligned}
F_\Theta = & \Box_{\boldsymbol{e_1}} \Box_{\boldsymbol{e_2}} ((\ell_{\boldsymbol{e_1}} = 2 \wedge \ell_{\boldsymbol{e_2}} = 2) \Rightarrow \bigvee_{i=1}^{k} F_{p_i}) \wedge \\
& \lceil \# \rceil \wedge \ell_{\boldsymbol{e_1}} = 1 \\
& \quad \langle \boldsymbol{e_1} \rangle \, (\lceil \# \rceil \wedge \ell_{\boldsymbol{e_2}} = 1 \, \langle \boldsymbol{e_2} \rangle \, \lceil \neg \# \rceil \, \langle \boldsymbol{e_2} \rangle \, \lceil \# \rceil \wedge \ell_{\boldsymbol{e_2}} = 1) \, \langle \boldsymbol{e_1} \rangle \\
& \lceil \# \rceil \wedge \ell_{\boldsymbol{e_1}} = 1
\end{aligned}
$$

The first part describes, that each 2×2 block must be in Θ. Note that we consider the discrete shape calculus and therefore we may only chop at discrete positions. The second part defines that the picture must be framed by #. As we consider a discrete time and space domain, the formula F_Θ is satisfiable if and only if the the local language $L(\Theta)$ is not empty. Therefore the satisfiability problem for shape calculus with discrete domain and more than one dimension is undecidable. □

6 Decidable Fragment

If we restrict the set of formulae to

$$F ::= \lceil P \rceil \mid \lceil P \rceil_{\boldsymbol{e}_x} \mid \lceil P \rceil_{\boldsymbol{e}_t} \mid F \wedge G \mid \neg F \mid F \, \langle \boldsymbol{e}_x \rangle \, G \mid F \, \langle \boldsymbol{e}_t \rangle \, G$$

and the set of models to one discrete infinite temporal dimension and consider all other spatial dimensions to be finite, the satisfiability problem is decidable.

We follow the lines of the decidability proof for discrete time Duration Calculus [HZ04] and extend it to one temporal and one finite spatial dimension with cardinality n, but it can be easily generalised to more than one finite spatial dimension. Assume we use boolean observables $X_1, \ldots, X_o$. Then for a point in space and time the vector $(x_1, \ldots, x_o) \in \{true, false\}^o$ describes which observables are true. We use $o \times n$ matrices to describe the space for a point of time and take the set $\Sigma^n = \{[w_1, \ldots, w_n] | w_i \in \{true, false\}^o\}$ of all $o \times n$ matrices as our alphabet.

We use $\Sigma^{\leq k} = \bigcup_{i=0}^{k} \Sigma^i$ as an abbreviation. For a state assertion P and $1 \leq i \leq n$ we define $\Sigma_{i,P}^n = \{[w_1 \ldots w_n] \in \Sigma^n | w_i \models P\}$ as all possible space configurations such that at point i in space the state assertion P is true. These

sets can be computed as we allow only propositional formulae for the state assertions.

Additionally we define $\Sigma^n_{\Box P} = \bigcap_{i=1}^n \Sigma^n_{i,P}$ and $\Sigma^n_{\Diamond P} = \bigcup_{i=1}^n \Sigma^n_{i,P}$. We define for every $1 \leq i, j \leq m \leq n$ the functions $h_{i,j,(m)}([w_1, \ldots, w_m]) = [w_i, \ldots, w_j]$ where we assume $h_{i,j,(m)}([w]) = \varepsilon$ if $j < i$. These functions yield the middle part from i up to j of the matrix $[w_1, \ldots, w_m]$.

For the construction of our languages, we use these functions as homomorphisms for languages $L \subseteq (\Sigma^m)^*$ and consider its inverse $h^{-1}_{i,j,(m)}$.

The construction of $\mathcal{L}_n(F)$ over the alphabet Σ^n proceeds inductively as follows:

$$\mathcal{L}_n(\lceil P \rceil) = (\Sigma^n_{\Box P})^+ \qquad \mathcal{L}_n(\lceil P \rceil_{\boldsymbol{e}_t}) = (\Sigma^n_{\Diamond P})^+$$

$$\mathcal{L}_n(\lceil P \rceil_{\boldsymbol{e}_x}) = \bigcap_{i=1}^{n}((\Sigma^n)^* \circ \Sigma^n_{i,P} \circ (\Sigma^n)^*) \qquad \mathcal{L}_n(F \wedge G) = \mathcal{L}_n(F) \cap \mathcal{L}_(G)$$

$$\mathcal{L}_n(F \langle \boldsymbol{e}_t \rangle G) = \mathcal{L}_n(F) \circ \mathcal{L}_n(G) \qquad \mathcal{L}_n(\neg F) = \overline{\mathcal{L}_n(F)}$$

For the spatial chop operation $F \langle \boldsymbol{e}_x \rangle G$, we consider every possible position i at which the chop can be applied. For every $0 \leq i \leq n$ we construct the languages $\mathcal{L}_i(F)$ and $\mathcal{L}_{n-1}(G)$. Every letter in $\mathcal{L}_i(F)$ is a $o \times i$ matrix which models a space with cardinality i. We have to consider every possible extention to the right of this matrix to an $o \times n$ matrix. This is done using the inverse homomorphism $h^{-1}_{1,i,(n)}$. The same is done for $\mathcal{L}_{n-1}(G)$. We end up with

$$\mathcal{L}_n(F \langle \boldsymbol{e}_x \rangle G) = \bigcup_{i=0}^{n} \left(h^{-1}_{1,i,(n)}(\mathcal{L}_i(F)) \cap h^{-1}_{i+1,n,(n)}(\mathcal{L}_{n-i}(G)) \right).$$

Using this construction for a formula F the following holds:

Theorem 1. *$\mathcal{L}_n(F)$ is empty if and only if F has no model.*

7 Conclusion

In this paper, we presented how a well known real-time formalism can be enriched to allow spatial verification. To this end, we generalised the Duration Calculus to more than one dimension and showed how spatial properties like continuous movement can be formalised. We gave an example to show that specifications in pure Duration Calculus can be used directly in the extension, spatial requirement just need to be added. We proved undecidability results and identified a decidable subset.

Related Work. There is a lot of related work around in the field of spatial and temporal logics. The *Region Connection Calculus* (RCC) proposed by Randell, Cui and Cohn [RCC92] is widely used in the AI community and fundamental for many extensions. It handles regions and their connectivity, for example whether

one region intersects another or whether it is a proper part. It does not handle time or quantitative measures and so it is not well suited for the description of mobile safety-critical real-time systems. It has been used in [Gal95] to develop a *qualitative* theory of movement. Additionally one should be able to encode the RCC in the shape calculus. This is still to be investigated.

A more recent approach by Merz et al [MWZ03] considers an extension of TLA to describe mobile systems, but this approach uses the π-Calculus [Mil99] notion of mobility, namely mobility by changes of links between processes. Also using this notion of mobility, Cardelli give a spatial logic tailored for the ambient calculus in [CG00] or the π-Calculus in [CC03] which allows to reason about freshness of names, name restriction and composition.

By contrast, the Real Space Process Algebra proposed by Baeten and Bergstra [BB92] uses the 3-dimensional space and in an relativistic extension the 4-dimensional space and describes actions occurring at certain points in space.

Closer to our approach are the recent multi-dimensional modal logics introduced for example in [BC02, AvB01, Mul98, ADN97] but they do not allow quantitative reasoning, neither spatial nor temporal. A logic specially tailored to reason about distances can be found in [WZ03] but this approach does not consider time.

Coming from Duration Calculus there are various extensions. The extension for hybrid systems by Zhou, Ravn and Hansen [ZRH93] introduces the diffential operator instead of the integral. But this formalism is still a temporal logic and does not take space into account.

Finally, a 2-dimensional extension of Duration Calculus by Pandya and Van Hung [PH98] uses 2-dimensional time, one dense time line (macro time) and orthogonal a discrete time line (stepped time) to model superdense computations.

Perspectives. For future work, we like to give a set of rules to make spatial reasoning more convenient and apply these rules on more case studies. The question whether there is a relative complete and sound calculus and finding richer decidable subsets are also things to tackle in the future.

Acknowledgements. The author thanks E.-R. Olderog and the members of the "Correct System Design" group for fruitful discussions and draft-reading earlier versions.

References

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADN97] S. N. Artemov, J.M. Davoren, and A. Nerode. Modal logics and topological semantics for hybrid systems. Technical Report 97-05, Cornell University Ithaca, 1997.

[AvB01] M. Aiello and H. van Benthem. A Modal Walk Through Space. Technical report, Institute for Logic, Language and Computation, University of Amsterdam, 2001.

[BB92] J.C.M. Baeten and J.A. Bergstra. Asynchronous Communication in Real Space Process Algebra. In Jan Vytopil, editor, *FTRTFT '92, Nijmegen, The Netherlands*, volume 571 of *LNCS*, pages 473–492. Springer, 1992.

[BC02] B. Bennett and A.G. Cohn. Multi-Dimensional Multi-Modal Logics as a Framworf for Spatio-Temporal Reasoning. *Applied Intelligence*, 17(3):239–251, 2002.

[CC03] L. Caires and L. Cardelli. A Spatial Logic for Concurrency. *Information and Computation*, 186(2):194–235, 2003.

[CG00] L. Cardelli and A. D. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *POPL 2000,*, pages 365–377. ACM Press, 2000.

[Gal95] A. Galton. Towards a qualitative theory of movement. In *Spatial Information Theory*, pages 377–396, 1995.

[GR97] D. Giammarresi and A. Restivo. *Handbook of Formal Languages – Beyond Words*, volume 3, chapter Two-Dimensional Languages, pages 215–267. Springer, 1997.

[HZ97] M. R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9:283–330, 1997.

[HZ04] M. R. Hansen and Zhou Chaochen. *Duration Calculus: A Formal Approach to Real-Time Systems*. EATCS: Monographs in Theoretical Computer Science. Springer, 2004.

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[Mul98] P. Muller. A Qualitative Theory of Motion Based on Spatio-Temporal Primitives. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *KR'98. Principles of Knowledge Representation and Reasoning, Trento, Italy*, pages 131–143. Morgan Kaufmann, 1998.

[MWZ03] S. Merz, M. Wirsing, and J. Zappe. A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems. In M. Pezzè, editor, *FASE 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 87–1014. Springer, 2003.

[PH98] P. K. Pandya and Dang Van Hung. Duration Calculus of Weakly Monotonic Time. In A. P. Ravn and H. Rischel, editors, *FTRTFT'98, Lyngby, Denmark*, number 1998 in LNCS, pages 55–64. Springer, 1998.

[Rav95] A. Ravn. Design of embedded real-time computing systems. Technical report, Dept. Comp. Science, Technical University of Denmark,Bld. 344, DK-2800 Lyngby, 1995.

[RCC92] D. A. Randell, Z. Cui, and A. Cohn. A Spatial Logic Based on Regions and Connection. In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning*, pages 165–176. Morgan Kaufmann, San Mateo, California, 1992.

[WZ03] F. Wolter and M. Zakharyaschev. Reasoning about distances. In G. Gottlob and T. Walsh, editors, *IJCAI-03, Acapulco, Mexico, August 9-15, 2003*, pages 1275–1282. Morgan Kaufmann, 2003.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZRH93] Zhou Chaochen, A.P. Ravn, and M.R. Hansen. An Extended Duration Calculus for Hybrid Real-Time Systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 36–59. Springer, 1993.

A Framework for Specification and Validation of Real-Time Systems Using *Circus* Actions

Adnan Sherif[1], He Jifeng[2,*], Ana Cavalcanti[3], and Augusto Sampaio[1]

[1] CIn/UFPE, P.O. Box 7851 Cidade Universitária, 50740-540 Recife - PE Brazil
ams@cin.ufpe.br,acas@cin.ufpe.br
[2] IIST/UNU, P.O. Box 3058, Macau
hjf@iist.unu.edu
[3] Computing Laboratory, University of Kent, Canterbury CT2 7NF, England
A.L.C.Cavalcanti@ukc.ac.uk

Abstract. In this work we propose a framework for specification and validation of real-time programs using *Circus* actions. *Circus* is a language that combines CSP, Z, and refinement calculus constructs. We have extended *Circus* and its model to capture time properties, and explored the relationship between the timed and the untimed model. Here we present a framework based on the integration of the timed and untimed versions of *Circus*. The integration aims at building a heterogeneous model that can express time properties using the untimed model. It is useful for the validation of real-time systems properties based on techniques and tools available for untimed languages. To illustrate the use of the framework, we apply it to an alarm system controller.

Keywords: Formal methods integration, formal verification, real-time systems.

1 Introduction

It has been noticed that a single software development method is not sufficient for solving all types of problems found in complex software systems. Integration has been a trend in the recent years: methods have been combined with the aim of obtaining more complete and expressive formalisms.

Several examples of combining formalisms can be found in the literature such as CSP-OZ, introduced by Fischer [3], in which CSP [11] and Z [17] are combined such that the CSP part defines the behaviour of the system, and Object-Z (OZ) [16] is used to specify the system data and operations. CSP-OZ is a typical case in which the integration aims at complementing a method

* On leave from East China Normal University, Shanghai. The work is partly supported by the 973 project of Ministry of Science and Technology of China (2002CB312001) *Formalizing the Grid software and design methods*, and a Key project of Ministry of Education of China *Component-based design methods*.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 478–493, 2005.
© Springer-Verlag Berlin Heidelberg 2005

with the aid of another method by increasing its expression capabilities. Other integrations are intended for reasoning in one formalism about other formalisms, such as the study presented by Yifeng and Zhuming concerning an integration of temporal logics [2]; their work explores the relation between different types of temporal logics.

The combination strategy can also be differentiated by the form in which the integration is carried out. Some approaches propose the use of informal links between the language constructs and limit the proposed integration to a relation that combines constructs of a notation with another; an example of this approach is Timed-CSP-Z [13]. Other approaches use a step by step framework that combines one method with another in a systematic manner; an example of such a method is that defined by Piage to integrate a subset of UML with a predicative programming design calculus [9]. Some more formal approaches propose the creation of a semantic model which can be used to relate and study different languages. An example is presented in [1] where a model based on category theory and viewpoints is used to relate and integrate different specification languages. Another example is the unifying theories of programming [5], which proposes an incremental semantic model based on simple predicate logic that can be expanded and used to define different paradigms and languages.

Circus is a combination of CSP and Z ; it also includes specification statements as in Morgan's refinement calculus [7] and Dijkstra's language of guarded commands. *Circus* has a formal semantics [21] based on the unifying theories of programming [5]. Case studies using the language are explored in [20]. A development method based on refinement is described in [12]. In [14, 15] a time model for *Circus* was proposed; some time operators have been added, giving *Circus* the capability of expressing time behaviour.

In this work, we present a framework for specification and validation of real-time programs using *Circus*. The main idea is to use the timed model for specification, and the untimed model to express desired properties. A normal form for a timed program is used to obtain an untimed program that can be verified to meet the time requirements. In this way, we reason about time properties without making explicit use of time. The untimed program, however, includes special (*timer*) events. The framework makes use of the meta-method for method integration based on heterogeneous notations [8]; in our case, the notations are a subset of the *Circus* actions and its timed extension.

In the next section we give a brief introduction to the meta-method for formal methods integration. In Section 3, we present our heterogeneous notation used for the specification of real-time systems using *Circus*. We present the framework and show its use in Section 4. In Section 5, we draw our conclusions.

2 Meta-method for Formal Methods Integration

A meta-method for formal methods integration is presented in [8]. It has been used to integrate a subset of UML with a predicative programming design calculus [9]. The method is summarized by the following steps.

Fix the Base Method. A base method provides the basic notation for the main processes, which will be combined with the other (invasive) methods. At this point, integrators need to have a clear understanding of the role of each method in the integrated approach.

Determine the Invasive Method(s). The invasive methods are chosen to complement the base method. They can complement the notation, or processes. In choosing the invasive method, it must also be decided how to reconcile overlaps.

Construct or Extend a Heterogeneous Basis. This is accomplished by constructing or adding notation from the base and invasive methods to a heterogeneous basis. A single formal notation for the heterogeneous basis can be fixed at this point.

Generalize and Relate the Method Steps. The method steps for the base and the invasive method are manipulated in order to define how they will work together in combination. Two forms of cooperation are particularly common:

- Generalization. The process of the base method is generalized to use heterogeneous notations constructed from those of the base and invasive methods.
- Interleaving. Relationships between the process of the base method and the processes of the invasive method are defined, using an informal entity relationship-like notation. Relationships can be established by defining a translation between notations, by replacing entire processes from the base method with processes from the invasive method, by supplementing a process from a method by a process from another method, or by defining relations that interleave the processes from the base and invasive methods.

Provide Guidance to the User. Examples and suggestions are given to the users to aid them in using the integrated method.

In the next section, we show the application of this method to integrate a subset of the *Circus* language with the time operators of its timed version.

3 The Heterogeneous Method

In our approach, we suggest the use of the *Circus* untimed original version as the base method, and the *Circus* timed model and the time operators as the invasive method. We define a heterogeneous notation that adds the time operators to the original language with the aid of timer events. Finally we give expansion laws that provide an axiomatic semantics for the timer events.

3.1 The Base Method

A modified subset of the action operators [21] of *Circus* is the base method. It is defined in Figure 1 using a BNF, where e stands for an expression, N for a valid name, N^+ for a list of names, and cs for a set of channel names.

$$
\begin{array}{l}
\mathsf{Action} ::= \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \\
\quad \mid \mathsf{Communication} \rightarrow \mathsf{Action} \\
\quad \mid \mathsf{b} \ \& \ \mathsf{Action} \\
\quad \mid \mathsf{Action} \sqcap \mathsf{Action} \\
\quad \mid \mathsf{Action} \ \Box \ \mathsf{Action} \\
\quad \mid \mathsf{Action}; \ \mathsf{Action} \\
\quad \mid \mathsf{Action} \ [\![\mathsf{cs}]\!] \ \mathsf{Action} \\
\quad \mid \mathsf{Action} \setminus \mathsf{cs} \\
\quad \mid \mathsf{Command} \\
\quad \mid \mu \, \mathsf{N} \bullet \mathsf{Action}
\end{array}
\qquad
\begin{array}{l}
\mathsf{Communication} ::= \mathsf{N} \ \mathsf{CParameter} \\
\mathsf{CParameter} ::= ? \, \mathsf{N} \mid ! \, \mathsf{e} \mid . \, \mathsf{e} \\
\mathsf{Command} ::= \mathsf{N}^{+} := \mathsf{e} \\
\qquad\qquad \mid \ \mathsf{Action} \ \ \mathsf{b} \ \ \mathsf{Action}
\end{array}
$$

Fig. 1. The subset of *Circus* used as the base method

The action *Skip* terminates immediately. *Stop* represents deadlock, which simply puts a program in an ever waiting state. *Chaos* is the worst action; nothing can be said about its behaviour.

An action can be prefixed with a communication (input or output) which takes place before the action starts. This action waits for the other actions that need to synchronize on the channel before the communication can take place. In $\mathsf{b} \ \& \ \mathsf{A}$, b is a guard: a boolean expression that has to be *true* for the action A to take place; otherwise A cannot proceed.

The internal choice $\mathsf{A} \sqcap \mathsf{B}$ selects A or B in a nondeterministic manner. The external choice $\mathsf{A} \ \Box \ \mathsf{B}$ waits for interaction with the environment; the first action that interacts with the environment (by either synchronizing on an event or terminating) is chosen. The sequential composition $\mathsf{A}; \ \mathsf{B}$ behaves as A followed immediately by B.

The parallel composition $\mathsf{A} \ [\![\mathsf{cs}]\!] \ \mathsf{B}$ involves a set cs containing the events on which A and B need to synchronize. A hiding operation $\mathsf{A} \setminus \mathsf{cs}$ takes a set cs of events that are to be excluded from the resulting observation of the action, hiding events that can no longer be seen by other actions.

Assignment is a command; it simply assigns a value to a variable in the current state. If the variable already exists, its value is overwritten, otherwise it is added to the current state and assigned the given value. The command $\mathsf{A} \lhd \mathsf{b} \rhd \mathsf{B}$ is a conditional: if b evaluates to *true*, then A is executed, otherwise B is chosen for execution.

For simplicity, we do not consider the Z and the refinement calculi specification constructs available in *Circus*. We also do not contemplate *Circus* processes, which effectively encapsulate state and action specifications. Our objective is to explore novel ideas related to the development of real-time systems; we leave the study of more elaborate *Circus* constructs as future work.

The semantics of *Circus* is based on the unifying theories of programming, and is explored in details in [21, 19]. The following is a description of the observation variables used by the untimed semantic model.

ok **and** *ok′* are boolean variables. When *ok* is *true*, the previous action did not diverge; *ok′* indicates that the current action is in a stable state.

wait **and** *wait′* are boolean variables. When *wait* is *true*, the program starts in an intermediate state of the execution of the previous program. When *wait′* is *true* the current action has not terminated; when it is false, it indicates termination.

state **and** *state′* are mappings from variable names to values. Each mapping associates every user variable in the action to a value. The dashed variable records the value of the variables at the final observation.

trace **and** *trace′* are sequences of observations on the action interaction with its environment: *trace* records the events that occurred before the action started, and *trace′* records a subsequent observation.

ref′ is a set of events the action can refuse.

A single observation is given by the combination of values of the above variables. The semantics of an action is given as a predicate over the observation variables. We give the semantics of the basic actions and communication as an example. For a complete specification see [21, 19].

The semantics of the action *Skip* establishes that the action can only terminate normally, and has no interaction with the environment.

$$[\![Skip]\!] \mathrel{\widehat{=}} ok' \land \neg wait' \land trace' = trace \land state' = state$$

The action *Stop* waits forever.

$$[\![Stop]\!] \mathrel{\widehat{=}} ok' \land wait' \land trace' = trace$$

An assignment assigns a value to a variable in the current state.

$$[\![x := e]\!] \mathrel{\widehat{=}} ok' \land \neg wait' \land trace' = trace \land state' = state \oplus \{x \mapsto e\}$$

If the variable does not exist it is added to the state. This is different from the *Circus* semantics, which requires that all variables are declared.

An action can engage in a communication if all the other actions involved in the same communication are ready to do so. We model this with the help of two predicates: $wait_com(c)$, which models the waiting state of an action to communicate on channel c, and $term_com(c.e)$ which represents the act of communicating a value e over a channel c.

$$wait_com(c) \mathrel{\widehat{=}} ok' \land wait' \land c \notin ref' \land trace' = trace$$
$$term_com(c.e) \mathrel{\widehat{=}} ok' \land \neg wait' \land trace' = trace \frown \langle c.e \rangle$$

The semantics of the output communication is given below.

$$[\![c!e]\!] \mathrel{\widehat{=}} wait_com(c) \lor (term_com(c.e) \land state' = state)$$

We can define the input operation in a similar manner.

$$[\![c?x]\!] \mathrel{\widehat{=}} wait_com(c) \lor \begin{pmatrix} term_com(c.e) \land \\ state' = state \oplus \{x \mapsto e\} \end{pmatrix}$$

In Section 4, we use this model as the base semantic model for the framework.

3.2 The Invasive Method

The invasive method selected is a timed version of *Circus*. It is presented in [14, 15]; two new time operators are added and the semantics is given in the context of a discrete time model.

The action Wait t halts the system for an amount of time determined by the positive integer expression t before terminating normally. The timeout construct ($\mathsf{A} \rhd^{\mathsf{t}} \mathsf{B}$) takes a positive integer value as the length of the timeout; it acts as a time guarded choice. If A performs an observable event or terminates before the specified time elapses, it is chosen. Otherwise, A is suspended and the only possible observations are those produced by B.

A new semantics for the language is given in [15] with the same observation variables ok, ok', $wait$, $wait'$, $state$ and $state'$. The variables $trace$, $trace'$ and ref, ref' are substituted by a new pair of variables tr and tr'.

The variable tr records the observations that occur before the program starts, and tr' records the subsequent observations. Each element of the sequence represents an observation at the end of one time unit. The sequence indexing starts from 0: the element at position 0 shows the observations before the action starts (at time zero). Each observation element is a tuple, where the first element is the sequence of events that occurred by the end of the time unit, and the second is the set of refusals at the end of the same time unit:

$$tr, tr' : \operatorname{seq}^{+}(\operatorname{seq} Event \times \mathbb{P}\, Event)$$

where seq^{+} stands for a non-empty sequence, and $Event$ represents all the possible events (communications) in which an action can engage. We show the use of these new variables in the definition of the Wait d action.

$$[\![\mathsf{Wait}\ \ \mathsf{d}]\!]_{time} \mathrel{\widehat{=}} \left(\begin{array}{c}(wait' \land (\#tr' - \#tr) < d) \lor \\ (\neg wait' \land (\#tr' - \#tr) = d \land state' = state)\end{array}\right) \land \\ (Flat(tr') - Flat(tr) = \langle\rangle \land ok')$$

where $Flat : \operatorname{seq}^{+}(\operatorname{seq} Event \times \mathbb{P}\, Event) \to \operatorname{seq} Event$ is defined as follows.

$$Flat(\langle (el, ref) \rangle) = el$$
$$Flat(\langle (el, ref) \rangle \frown S) = el \frown Flat(S)$$

The only possible behaviour for this action is to wait for the specified number of time units to pass before terminating immediately.

3.3 Linking Circus Models

In [14, 15] we explore a relation between the observations of the original untimed *Circus* model and the time model. We define a function L that, when applied to a set of timed observations related to a *Circus* action A, yields a related observation in the original model without time information. The mapping is useful to validate the safety properties that are time independent. The definition is as follows:

$$L([\![A]\!]_{time}) \mathrel{\widehat{=}} \exists\, tr, tr' \bullet [\![A]\!]_{time} \land trace = Flat(tr) \land trace' = Flat(tr') \land \\ ref' = second(last(tr'))$$

The *Flat* function is applied to the timed traces to obtain the original trace model. A projection on the second element of the last entry in tr' results in the refusal set of the original model.

The mapping L is conservative as it preserves all the behaviours of the original timed action in the untimed model, but it might add an undesired deadlock state. This is due to the fact that applying the mapping to Wait d results in $Stop \lor Skip$ (see [14, 15] for details).

A new mapping can be defined using L. It ensures that the only possible waiting state of a program is a waiting state that can wait forever. The definition of this function is as follows:

$$\hat{L}([\![A]\!]_{time}) \mathrel{\widehat{=}} L([\![A]\!]_{time}) \land (wait \Rightarrow EndlessObs)$$

where *EndlessObs* are observations that wait for an arbitrary n time units

$$EndlessObs \mathrel{\widehat{=}} \forall\, n \bullet \exists\, tr_o \bullet tr_o = tr \frown \langle(\langle\rangle, ref)\rangle^n \land L([\![A]\!]_{time})[tr_o/tr']$$

The notation $\langle e\rangle^n$ stands for a sequence with n occurrences of e. The mapping $\hat{L}$ is not conservative as it does not distribute over parallel composition. The mapping functions and their properties are explored in more detail in [14, 15].

3.4 The Heterogeneous Notation

The syntax of the heterogeneous notation is the same as the base notation, with the addition of timer events. We define a normal form to express the timed actions; it adds special timer events to the program which are used by the framework. The timer events have differentiated behaviour in a parallelism. Therefore, we define a new parallel composition operator for the heterogeneous notation, with an axiomatic semantics.

Usually timed programs are implemented with timers: the system clock or a dedicated timer. Following this idea, we give a normal form for the time operators. They are implemented as a parallel composition of a timer and an untimed program which synchronizes on dedicated events. The following is the specification of the timer.

$$Timer(i,d) \mathrel{\widehat{=}} \mu\, X \bullet \left(\begin{array}{l} setup.i?d \rightarrow \left(\begin{array}{l} (halt.i \rightarrow X) \\ \Box\ \mathsf{Wait}\ \ \mathsf{d};\ \left(\begin{array}{l}(out.i \rightarrow X)\\ \Box \\ (terminate.i \rightarrow Skip)\end{array}\right) \\ \Box\ (terminate.i \rightarrow Skip)\end{array}\right) \\ \Box\ (terminate.i \rightarrow Skip)\end{array}\right)$$

The timer is initiated with the event $setup.i?d$, which takes the timer instance identifier i and the delay d as input. The behaviour of the timer is then to offer the event $halt.i$ while it waits for d time units; at the end, the event $out.i$ is offered. When either $halt.i$ or $out.i$ takes place, the timer is reset. The timer always offers the event $terminate.i$ that terminates the time suspending its execution. The events $setup.i?d$ and $out.i$ are special, as discussed later.

The function Φ takes a timed action and returns an action in a normal form that uses the timer events, instead of timed constraints. It is an identity for the basic actions (except for *Wait*), and Φ distributes through the binary operators except for timeout, external choice and parallel composition. Further,

$$\begin{aligned}
\Phi(b \;\&\; A) &= b \;\&\; \Phi(A) \\
\Phi(A \setminus cs) &= \Phi(A) \setminus cs \\
\Phi(c \rightarrow A) &= c \rightarrow \Phi(A) \\
\Phi(\mu\, X \bullet A(X))) &= (\mu\, X \bullet \Phi(A(X)))
\end{aligned}$$

The relevant cases are *Wait*, timeout, external choice, and parallel composition.

$$\begin{aligned}
\Phi(\mathsf{Wait}\ \ \mathsf{d}) &\mathrel{\widehat{=}} (setup.i!d \rightarrow out.i \rightarrow Skip) \\
\Phi((\mathop{\Box}_{j=1}^{n} c_j \rightarrow A_j) \overset{d}{\triangleright} B) &\mathrel{\widehat{=}} setup.i!d \rightarrow \left(\begin{array}{c} (\mathop{\Box}_{j=1}^{n} c_j \rightarrow halt.i \rightarrow \Phi(A_j)) \\ \Box \\ (out.i \rightarrow \Phi(B)) \end{array}\right) \\
\Phi(A \,\Box\, B) &= \Phi(A) \boxtimes \Phi(B) \\
\Phi(A \,[\![\, cs \,]\!]\, B) &\mathrel{\widehat{=}} \Phi(A) \,[\![\, cs \,]\!]^{nf}\, \Phi(B)
\end{aligned}$$

For $\mathsf{Wait}\ \ \mathsf{d}$, Φ replaces the wait action by an action that initializes a timer with $setup.i!d$ and then waits for the occurrence of $out.i$. For the timeout operator, we consider a particular case: $(\mathop{\Box}_{j=1}^{n} c_j \rightarrow A_j) \overset{d}{\triangleright} B$. The reason is that the timeout has to stop the timer (using the event $halt.i$) after the occurrence of first event. This does not lead to loss of generality because, using the algebraic laws of *Circus* [12], we can transform any action to the required form. For external choice Φ introduces a new external choice operator $\boxtimes$. The new operator is used to give the semantics of the time events in a choice. The new external choice semantics is given by the following algebraic laws.

In the laws below we take c_i and d_i to be ordinary events (they can not be timer events) and A and B to be actions defined in the normal form. The first law shows the *idempotent* of the $\boxtimes$ operator

Law 1. $A \boxtimes A = A$

The following law states that the $\boxtimes$ operator is commutative

Law 2. $A \boxtimes B = B \boxtimes A$

The following is the associativity law for the $\boxtimes$ operator

Law 3. $A \boxtimes (B \boxtimes C) = (A \boxtimes B) \boxtimes C$

The next law states that the *setup* event is not determinant in the choice and it occurs without effecting the choice.

Law 4. $(setup.i.d \rightarrow A) \boxtimes B = setup.i.d \rightarrow (A \boxtimes B)$

The behaviour of the choice operator $\boxtimes$ is given for the case in which one of the actions involved in the choice is the time event *out*. As in the case of the *setup* event the *out* event is not determinant in the choice but different form *setup*, the *out* event offers the none time events c_i as a choice using the base method external choice operator.

Law 5. $(out.j \rightarrow A) \boxtimes (\overset{n}{\underset{i=1}{\Box}} c_i \rightarrow B_i) = \begin{array}{l}(out.j \rightarrow (A \boxtimes (\overset{n}{\underset{i=1}{\Box}} c_i \rightarrow B_i))) \\ \Box\ (\overset{n}{\underset{i=1}{\Box}} c_i \rightarrow halt.j \rightarrow B_i)\end{array}$

In the case that the choice operator $\boxtimes$ is involving two *out* events then the events are offered in the order of the smallest delay first. The cases are based on the delay of the timers where the function $delay(i)$ return the delay of a timer given by the index i. In the case the two *out* events in the choice correspond to timers with the same delay then the choice is nondeterministic.

Law 6

$$\begin{array}{c}(out.i \rightarrow A) \\ \boxtimes \\ (out.j \rightarrow B)\end{array} = \begin{cases} out.i \rightarrow (A \boxtimes (out.j \rightarrow B)), & \text{if } delay(i) < delay(j); \\ out.j \rightarrow ((out.i \rightarrow A) \boxtimes B), & \text{if } delay(i) > delay(j); \\ (out.i \rightarrow out.j \rightarrow (A \boxtimes B)) \\ \sqcap\ (out.j \rightarrow out.i \rightarrow (A \boxtimes B)), & \text{if } delay(i) = delay(j). \end{cases}$$

The next law states that the choice operator $\boxtimes$ is solved between timer events.

Law 7.

$$\begin{array}{c}((out.i \rightarrow A) \Box C) \\ \boxtimes \\ ((out.j \rightarrow B) \Box D)\end{array} = \left(\begin{array}{c}(out.i \rightarrow A) \\ \boxtimes \\ (out.j \rightarrow B)\end{array}\right) \Box (C \Box D)$$

The final law for the operator $\boxtimes$ states that for all the other in which A or B do no start with the timer events then the choice is the same as the external choice operator of the base method.

Law 8. $A \boxtimes B = A \Box B$ *Provided that setup and out are not in the initial events of* A *and* B.

Theorem 1. *An action of the form* $A \boxtimes B$ *can always be reduced to* $A' \Box B'$ *such that* A' *and* B' *are actions that do not contain the operator* $\boxtimes$.

For parallel composition Φ introduces a new parallel operator $[\![cs]\!]^{nf}$, whose semantics is given below.

We define some expansion laws that can be used to convert a parallel action into a sequential action. We use these expansion laws to give an axiomatic semantics for the timer events and the parallel operator introduced by Φ. The first law gives the basic expansion of the parallel composition of two actions that are ready to perform a communication independently

Law 9

$$(\mathop{\Box}_{i=1}^{n} c_i \to A_i) \,[\![\, cs \,]\!]^{nf}\, (\mathop{\Box}_{j=1}^{m} d_j \to B_j) = \left(\mathop{\Box}_{i=1}^{n} c_i \to \left(A_i \,[\![\, cs \,]\!]^{nf}\, (\mathop{\Box}_{j=1}^{m} d_j \to B_j)\right)\right) \Box \left(\mathop{\Box}_{j=1}^{m} d_j \to \left((\mathop{\Box}_{i=1}^{n} c_i \to A_i) \,[\![\, cs \,]\!]^{nf}\, B_j\right)\right)$$

if, for all i and j, $(c_i \notin cs) \wedge (d_j \notin cs)$.

The next law shows the expansion of a parallel composition where one of the actions needs to synchronize with the other; in this case the free action can proceed while the other waits for the synchronization.

Law 10. $(\mathop{\Box}_{i=1}^{n} c_i \to A_i) \,[\![\, cs \,]\!]^{nf}\, (\mathop{\Box}_{j=1}^{m} d_j \to B_j) = \mathop{\Box}_{j=1}^{m} d_j \to \begin{pmatrix} (\mathop{\Box}_{i=1}^{n} c_i \to A_i) \\ [\![cs]\!]^{nf} \\ B_j \end{pmatrix}$

if $(c_i \in cs) \wedge (d_j \notin cs)$.

Law 3 states that the parallel composition will stop if both programs need to synchronize, but disagree on the events to synchronize.

Law 11. $(\mathop{\Box}_{i=1}^{n} c_i \to A_i) \,[\![\, cs \,]\!]^{nf}\, (\mathop{\Box}_{j=1}^{m} d_j \to B_j) = Stop$
if $(c_i \in cs) \wedge (d_j \in cs) \wedge (c_i \neq d_i)$.

Law 4 states that the programs synchronize if they are ready to engage in the same event.

Law 12. $(\mathop{\Box}_{i=1}^{n} c_i \to A_i) \,[\![\, cs \,]\!]^{nf}\, (\mathop{\Box}_{i=1}^{n} c_i \to B_i) = A_i \mathop{}_{i=1}^{n} \,[\![\, cs \,]\!]^{nf} \mathop{}_{i=1}^{n} B_i$
if $(c_i \in cs)$.

The next law states that the events *setup*, when in parallel, have to occur before any other events in the action can occur.

Law 13. $(setup.i!d \to A) \,[\![\, cs \,]\!]^{nf}\, (setup.j!e \to B) = \begin{pmatrix} (setup.i!d \to setup.j!e) \\ \sqcap \\ (setup.j!e \to setup.i!d) \end{pmatrix} ;\ (A \,[\![\, cs \,]\!]^{nf}\, B)$

The last expansion laws state the order in which the events *out* synchronize, based on the delay of the timers for each event. The function *delay* takes a timer index and yields the delay setting of the timer associated to the index.

Law 14. $(out.i \to A) \,[\![\, cs \,]\!]^{nf}\, (out.j \to B) = out.i \to (A \,[\![\, cs \,]\!]^{nf}\, (out.j \to B))$
if $delay(i) < delay(j)$

Law 15. $(out.i \to A) \,[\![\, cs \,]\!]^{nf}\, (out.j \to B) = (out.i \to out.j) \sqcap (out.j \to out.i);\ (A [\![cs]\!]^{nf} B)$
if $delay(i) = delay(j)$

From the definition of the function Φ, and the semantics of the timed model, we derive a theorem that establishes a relative notion of semantic preservation: the generated (untimed) program, when operating with the timers, behaves as the original timed program.

Theorem 2. *For any timed action A,* $A \equiv \Phi(A)$ ***par*** $Timers_n$ *where* ***par*** *is defined as* A ***par*** $B \mathrel{\widehat{=}} (A;\ Terminate(n) \,[\![\, TSet \,]\!]\, B) \setminus TSet$. $Timers_n$ *is the interleaving of the n timers needed by the action A (one for each time operator).* $Terminate(n)$ *is given as* $\overset{n}{\underset{i=1}{|||}} (terminate.i \rightarrow Skip)$. $TSet$ *is given as the set of all timer events* $TSet = \{setup, halt, out, terminate\}$.

The operator **par** stands for the parallel composition of two actions that synchronize on the timer events. The timer events are hidden and, therefore, treated as internal events. We also have the following result.

Theorem 3. *For any timed action A, the following holds* $\hat{L}([\![A]\!]_{time}) \equiv [\![\Phi(A) \setminus \{setup, halt, out\}]\!]$.

This states that Φ does not change untimed behaviour.

3.5 The Heterogeneous Framework

In this section, we present the framework based on the heterogeneous notation. Figure 2 illustrates the steps for using the framework, which can be summarized as follows.

1. We start with a specification in the time model, using the timed version of the language. The designer gives a complete description of the system.

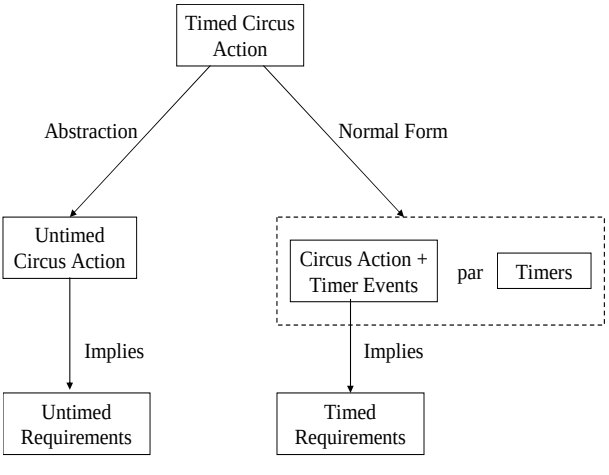


Fig. 2. A heterogeneous framework for analysis of timed programs in *Circus*

2. With the help of the abstraction function $\hat{L}$ (Section 3.3), we obtain an untimed version of the original specification.
3. The untimed action can be checked to validate the behaviour and safety requirements that are not time dependent. It satisfies any untimed behaviour requirement satisfied by the original action.
4. With the help of Φ, we obtain a program that has the same semantics as the original program (Theorem 1), but contains internal timer events.
5. Time requirements should be expressed using the timer events. This is why we can show that the untimed program satisfies the time requirements.

The next section explores the approach by applying it to a case study.

4 An Alarm System

The system used here as case study is also considered in [6]; it is a common burglar alarm controller connected to sensors which detect movements or changes in the environment. When disabled, the controller ignores any disturbance detected; when enabled, the controller will sound an alarm when a sensor signals a disturbance.

There are two timing requirements on the alarm controller. The first states that after enabling the alarm controller, there is a period T_1 before a disturbance causes the alarm to ring. This period permits a person to enable the alarm and get out. The second requirement states that, when a disturbance is detected, the controller will wait for a period T_2 before activating the alarm. This will allow a person to enter the building and deactivate the alarm.

4.1 The Timed Specification

The event *enable* indicates that the alarm system is enabled. To disable the alarm, the event *disable* is used. When the alarm system is disabled it responds only to the event *enable*. The event *disturbed* indicates that a sensor has detected a disturbance. Finally, *alarm* signals the firing of the alarm.

We can model the alarm controller with the help of several actions that are composed to produce the final system *Alarm*.

$$
\begin{aligned}
\mathit{Disable} &\mathrel{\widehat{=}} (\mathit{disable} \rightarrow \mathit{Skip}) \\
\mathit{Running} &\mathrel{\widehat{=}} \mathit{Disable} \,\Box\, (\mathit{disturbed} \rightarrow \mathit{Active}) \\
\mathit{Active} &\mathrel{\widehat{=}} \mathit{Disable} \overset{T_2}{\triangleright} (\mathit{alarm} \rightarrow \mathit{Disable}) \\
\mathit{Alarm} &\mathrel{\widehat{=}} \mu\, X \bullet (\mathit{enable} \rightarrow (\mathit{Disable} \overset{T_1}{\triangleright} \mathit{Running})) ;\; X
\end{aligned}
$$

The action *Disable* simply offers to engage on event *disable*. The action *Running* represents the armed behaviour of the alarm controller: the controller can either be disabled or it can be *disturbed* by a burglar. When the alarm is disturbed, it behaves as *Active*, which models the active state of the alarm. In this state,

the controller can to be disabled for the first T_2 time units, after what an *alarm* is fired. When fired, the controller will only terminate once disabled. The main action *Alarm* is recursive. The controller starts by assuming that the alarm is disabled and offers the *enable* event to start the controller. After the event *enable*, the action can be disabled for the first T_1 time units before it is armed.

4.2 Heterogeneous Alarm Controller

By applying the normal form function Φ to the timed specification of the alarm controller, we obtain the following result.

$$\Phi(Disable) = Disable$$
$$\Phi(Running) = Disable \,\Box\, (disturbed \rightarrow \Phi(Active))$$
$$\Phi(Active) = setup.2!\,T_2 \rightarrow \left(\begin{array}{c}(Disable;\ halt.2 \rightarrow Skip)\\ \Box\ (out.2 \rightarrow alarm \rightarrow Disable))\end{array}\right)$$
$$\Phi(Alarm) = \mu\, X \bullet (enable \rightarrow setup.1!\,T_1 \rightarrow \left(\begin{array}{c}(Disable;\ halt.1 \rightarrow Skip)\\ \Box\ (out.1 \rightarrow \Phi(Running))\end{array}\right);\ X$$

The generated program contains no time information: just timer events. We define one timer for each timeout operator used in the original specification.

$$Timer_1 \mathrel{\widehat{=}} Timer(1, T_1)$$
$$Timer_2 \mathrel{\widehat{=}} Timer(2, T_2)$$

From Theorem 2 we know that $(\Phi(Alarm)\ \mathsf{par}\ (Timer_1 \| Timer_2)) \equiv Alarm$. This equivalence is the basis for assuring that the validation step to come is correct.

4.3 Requirements

We express the alarm controller requirements as a time action. A safety requirement of our alarm controller is that it should guarantee the elapse of at least T_1 time units before the alarm can be *disturbed*. It should assure also that after detecting a disturbance it waits for at least T_2 time units before the alarm is trigged. These requirements can be expressed in the following action *Req*

$$Req \mathrel{\widehat{=}} \mu\, X \bullet enable \rightarrow \mathsf{Wait}\ \ \mathsf{T}_1;\ disturbed \rightarrow \mathsf{Wait}\ \ \mathsf{T}_2;\ alarm \rightarrow disable \rightarrow X$$

We can obtain an untimed version of our requirements by applying the Φ function as follows

$$\Phi(Req) = \mu\, X \bullet enable \rightarrow setup.1!\,T_1 \rightarrow out.1 \rightarrow disturbed \rightarrow setup.1!\,T_1 \rightarrow$$
$$out.1\, alarm \rightarrow disable \rightarrow X$$

The timers needed for the above program are as follows

$$Timer_1 \mathrel{\widehat{=}} Timer(1, T_1)$$
$$Timer_2 \mathrel{\widehat{=}} Timer(2, T_2)$$

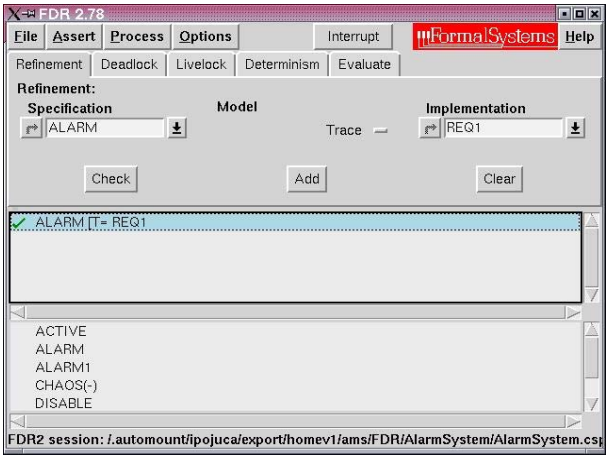


Fig. 3. Screen dump of the FDR tool used in the validation of the alarm controller example

Notice that the alarm controller and the requirement in the normal form use the same timers; therefore we need to prove that the untimed alarm controller $\Phi(Alarm)$ satisfies its untimed requirements $\Phi(Req)$. Because both actions are in the normal form, do not contain any more time operators and, the semantics of the time events has been given by the normal form expansion laws, we can use CSP model checking tool FDR [4] in the verification process. Figure 3 shows the use of FDR in the validation of the above problem. The proof can be conducted in an algebraic form. We can use the semantic model also in the proof, in A

5 Conclusions

In this paper we presented a framework for the specification and validation of real-time systems using a subset of *Circus*. We started by presenting a heterogeneous notation that combines the untimed version of the language and the time operators of timed *Circus*. The heterogeneous notation was created following the meta-method approach for formal integration presented by Paige in [8, 9].

A framework that makes use of the heterogeneous notation has also been introduced. The framework presents an approach to validate the time requirements of a system using the untimed heterogeneous notation.

Time requirements are specified using a particular model of time, either a discrete or a continuous model. We observe that our approach is independent of the time model used in the system specification. This is because the heterogeneous notation makes use of the untimed model and the time requirements are translated to timer events requirements. We have used *Circus* discrete time model, but a continuous model can also be used.

In [10], Qin, Dong and Chin propose a semantic model for Timed Communicating Object Z (TCOZ) based on the Unifying Theories of Programming. The semantic model gives a unified semantics for both channels and sensors/actuators based communication. The authors also extend TCOZ by adding two new timing constraints (*deadline* and *waituntil*). The aim is to create a complete and unified semantic model for TCOZ that can be later used to validate transformation rules from TCOZ to other semantic models such as Timed Automata. However, the authors do not show how this can be carried out.

The semantic model for TCOZ proposed by Qin,Dong and Chin in [10] and the model used for Timed RSL as proposed by Ri and Jifeng in [18] are both based on the semantic model proposed by Sherif and Jifeng in [15]; this model is also the basis of our framework. This permits the use of the same framework to validate time properties of these languages using the same principles but changing some of the time mapping functions. As a future exercise, our framework could be adopted to validate time requirements of Timed RSL and TCOZ.

As future work we also intend to apply this framework to an industrial real-time system and study its usage along with the *Circus* refinement calculus. We also intend to develop a tool, possibly an extension of FDR [4], to automate the validation process.

References

1. Marius C. Bujoriam. Integration of specification languages using veiwpoints. In Boiten, Derrick, and Smith, editors, *IFM 2004*, volume 2999 of *LNCS*, pages 422–440. Springer-Verlag, 2004.
2. Yifeng Chan and Zhuming Liu. Integrating Termporal Logics. In Boiten, Derrick, and Smith, editors, *IFM 2004*, volume 2999 of *LNCS*, pages 402–420. Springer-Verlag, 2004.
3. C. Fischer. *Combination and implementation of processes and data: from csp-oz to java.* PhD thesis, University of Oldenburg, 2000.
4. Formal Systems (Europe) Ltd. *FDR: User Manual and Tutorial, version 2.01*, August 1996.
5. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice-Hall Series in Computer Science, 1998.
6. Li Li and He Jifeng. A Denotational Semantics of Timed RSL using Duration Calculus. R 168, IIST/UNU, P.O. Box 3058 Macau, July 1999.
7. C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, 2nd edition, 1994.
8. R. Paige. A meta-method for formal method integration. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME 97*, volume 1313 of *LNCS*, pages 473–494. Springer-Verlag, 1997.
9. R. Paige. Integrating a program design calculus and a subset of UML. *The Computer Journal*, 42(2):82–99, 1999.
10. Shengchao Qin, Jin Song Dong, and Wei-Ngan Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In Araki, Gnesi, and Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*, pages 321–340. Springer-Verlag, 2003.
11. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, 1998.

12. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002*, volume 2391 of *LNCS*, pages 451–470. Springer, 2002.
13. A. Sherif. Formal Specification and Validation of Real-Time Systems. Master's thesis, Centro de Informática, UFPE, 2000.
14. Adnan Sherif and He Jifeng. Towards a Time Model for Circus. Technical Report 257, IIST/UNU, P.O. Box 3058, Macau, July 2002.
15. Adnan Sherif and He Jifeng. Towards a Time Model for Circus. In Chris George and Muaikou Miao, editors, *ICFEM 2002*, volume 2495 of *LNCS*, pages 613–624. Springer-Verlag, 2002.
16. Graeme Smith. *The Object-Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000.
17. M. Spivey. *The Z Notation*. Prentice-Hall International, 2nd edition, 1992.
18. Ri Hyon Sul and He Jifeng. Complete Verification System for Timed RSL. Technical Report 275, IIST/UNU, P.O. Box 3058, Macau, March 2003.
19. J. C. P. Woodcock and A. L. C. Cavalcanti. Circus: a concurrent refinement language. Technical report, Oxford University Computing Laboratory, July 2001.
20. J. C. P. Woodcock and A. L. C. Cavalcanti. The steam boiler in a unified theory of Z and CSP. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, 2001.
21. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.

Switched Probabilistic I/O Automata

Ling Cheung[1,⋆], Nancy Lynch[2,⋆⋆], Roberto Segala[3,⋆⋆⋆], and Frits Vaandrager[1]

[1] Nijmegen Institute for Computing and Information Sciences,
University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{lcheung, fvaan}@cs.kun.nl
[2] MIT Computer Science and Artificial Intelligence Laboratory,
Cambridge, MA 02139, USA
lynch@theory.csail.mit.edu
[3] Dipartimento di Informatica, Università di Verona,
Strada Le Grazie 15, 37134 Verona, Italy
roberto.segala@univr.it

Abstract. A *switched probabilistic I/O automaton* is a special kind of probabilistic I/O automaton (PIOA), enriched with an explicit mechanism to exchange control with its environment. Every closed system of switched automata satisfies the key property that, in any reachable state, at most one component automaton is active. We define a trace-based semantics for switched PIOAs and prove it is compositional. We also propose *switch extensions* of an arbitrary PIOA and use these extensions to define a new trace-based semantics for PIOAs.

1 Introduction

Probabilistic automata [Seg95, Sto02] constitute a mathematical framework for modeling and analyzing probabilistic systems, specifically, systems consisting of asynchronously interacting components capable of nondeterministic and probabilistic choices. This framework has been successfully adopted in the studies of distributed algorithms [LSS94, PSL00, Agg94] and practical communication protocols [SV99].

An important part of such a framework is a notion of *visible behavior* of system components. This is used to derive implementation and equivalence relations among components. For example, one can define the visible behavior of a nondeterministic automaton to be its set of *traces* (i.e., sequences of visible actions that arise during executions of the automaton [LT89]). This induces

⋆ Supported by DFG/NWO bilateral cooperation project 600.050.011.01 Validation of Stochastic Systems (VOSS).
⋆⋆ Supported by DARPA/AFOSR MURI Award #F49620-02-1-0325, MURI AFOSR Award #F49620-00-1-0327, NSF Award #CCR-0326277, and USAF, AFRL Award #FA9550-04-1-0121.
⋆⋆⋆ Supported by MURST project Constraint-based Verification of reactive systems (CoVer).

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 494–510, 2005.
© Springer-Verlag Berlin Heidelberg 2005

an implementation (resp. equivalence) relation on nondeterministic automata, namely inclusion (resp. equality) of sets of traces.

Perhaps the most important property of an implementation relation is *compositionality*: if P implements Q, then for every context R, one should be able to infer that $P \| R$ implements $Q \| R$. This property facilitates correctness proofs of complex systems by reducing properties of a large system to properties of smaller subsystems. In the setting of security analysis, for instance, compositionality ensures that plugging secure components into a security preserving context results again in a secure component [Can01].

Generalizing the notion of traces, Segala [Seg95] defines the visible behavior of a probabilistic automaton as its set of *trace distributions*, where each trace distribution is induced by a probabilistic *scheduler* which resolves all nondeterministic choices. This gives rise to implementation and equivalence relations as inclusion and equality of sets of trace distributions, respectively. It turns out that this notion of implementation relation is not compositional. A simple counterexample is illustrated in Figure 1.

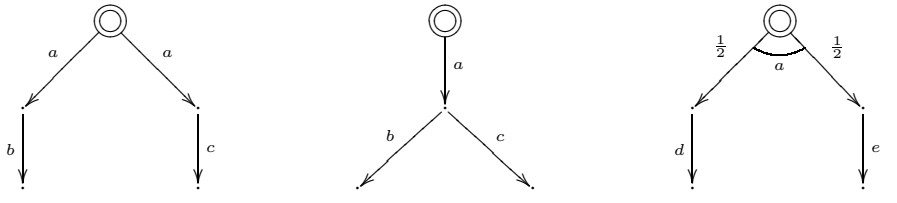


Fig. 1. Probabilistic automata Early, Late and Toss

As their names suggest, automaton Early forces its scheduler to choose between b and c as it chooses one of the two available a-transitions, whereas automaton Late allows its schedulers to make this decision after the a-transition. Clearly, these two automata have the same set of trace distributions, but they can be distinguished by the context Toss. The composed system Late $\|$ Toss has a trace distribution D_0 that assigns probability $\frac{1}{2}$ to each of these traces: adb and aec . Such total correlations between actions d and b, and between actions e and c, cannot be achieved by the composite Early $\|$ Toss.

Inspired by this example, we establish in [LSV03] that the coarsest precongruence refining trace distribution preorder coincides with the probabilistic simulation preorder. In other words, probabilistic contexts are capable of exposing internal branching structures of other components.

Aside from its inspirational merits, this example reveals an unsatisfactory aspect of the composition mechanism of probabilistic automata. Namely, nondeterministic choices are resolved after the two automata are composed, allowing the global scheduler to make decisions in one component using state information of the other. This phenomenon can be viewed as a form of "information leakage": the global scheduler channels private information from one component to the other, in particular, from Toss to Late.

In this paper, we present a composition mechanism where local scheduling decisions are based on strictly local information. That is, (i) local nondeterministic choices of each component are resolved by that component alone; (ii) global nondeterministic choices (i.e., inter-component choices) are resolved by some independent means. To address the first issue, we introduce an input/output distinction to our model and pair each automaton with an *input/output scheduler*. For the second, we introduce a control-passage[1] mechanism, which eliminates global scheduling conflicts.

Before describing our model in greater detail, we take a quick look at related proposals in the existing literature. (We refer to [SV04] for a comparative study of various probabilistic models.) For purely synchronous, variable-based models, global nondeterministic choices are resolved by "avoidance": in each transition of the global system, all components may take a step. This intrinsic feature of synchronous models allows De Alfaro, Henzinger and Jhala [dAHJ01] to successfully define a compositional, trace-based semantics for their model of probabilistic reactive modules. For asynchronous models such as probabilistic automata, global nondeterministic choices must be resolved explicitly in order to assign a probability mass to each possible interleaving of actions. Wu, Smolka and Stark [WSS94] propose a compositional model based on probabilistic input/output automata. In that model, global nondeterminism is resolved by a "race" among components: each component draws a delay from an exponential distribution (thus leaving the realm of discrete distributions). Assuming independence of these random draws, the probability of two components drawing the same delay is zero, therefore there is almost always a unique winner.

In this paper, we introduce the model of *switched probabilistic I/O automata* (or *switched automata* for short). This augments the probabilistic I/O automata model with some additional structures and axioms. In particular, we add a predicate active on the set of states, indicating whether the automaton is active or inactive. We require that locally controlled actions are enabled only if the automaton is active. In other words, an inactive automaton must be quiescent and can only accept inputs from the environment.

A switched automaton changes its activity status by performing special control input and control output actions. Control inputs switch the machine from inactive to active and vice versa for control outputs. All other actions must leave the activity status unchanged. It is important that all control communications are "handshakes": at most two components may participate in a synchronization labeled by a control action. Together with an appropriate initialization condition, this ensures that at most one component is active at any point of an execution. Intuitively, we model a network of processes passing a single token among them,

[1] Throughout this paper, the term *control* is used in the spirit of "control flow" in sequential programming: a component is said to possess the control of a system if it is scheduled to actively perform the next action. This should not be confused with the notion of controllers for plants, as in control theory.

with the property that a process enables a locally controlled transition if and only if it possesses the token.

The main technical result of this paper is compositionality of a trace-based semantics for switched probabilistic I/O automata (Section 6, Theorem 1). Sections 2 and 3 are devoted to the basic theory. There we introduce new technical notions such as I/O schedulers, scheduled automata and parallel composition of scheduled automata. In Section 4, we define pseudo probabilistic executions and pseudo trace distributions for automata with open inputs, and prove important projection and pasting results. Section 5 treats two standard operators: renaming and hiding. In Section 7, we propose the notion of *switch extensions* for PIOAs, which can be used to derive a new form of composition for the original PIOA model. Concluding discussions follow in Section 8. Due to space constraints, we have omitted many proofs. These can be found in a full version of this paper available at `http://www.cs.kun.nl/ita/publications/papers/fvaan/switched.html`.

2 Preliminaries

In this section, we define probabilistic I/O automata and some related notions. This is a straightforward combination of the Input/Output Automata model of Lynch and Tuttle [LT89] and the Simple Probabilistic Automata model of Segala [Seg95].

A *discrete probability (resp. sub-probability) measure* over a set X is a measure μ on $(X, 2^X)$ such that $\mu(X) = 1$ (resp. $\mu(X) \leq 1$). With slight abuse of notation, we write $\mu(x)$ for $\mu(\{x\})$. The set of all discrete probability measures over X is denoted $\mathsf{Disc}(X)$; similarly for $\mathsf{SubDisc}(X)$. Moreover, we use $\mathsf{Supp}(\mu)$ to denote the *support* of a discrete measure μ: the set of elements in X to which μ assigns nonzero measure. Given $x \in X$, the *Dirac distribution* on x is the unique measure assigning probability 1 to x, denoted $(x \mapsto 1)$.

A *probabilistic I/O automaton (PIOA)* P consists of:

- a set $\mathsf{States}(P)$ of states and a *start state* $s^0 \in \mathsf{States}(P)$;
- a set $Act(P)$ of action symbols, partitioned into: I (*input* actions), O (*output* actions) and H (*hidden* actions);
- a transition relation $\rightarrow \subseteq \mathsf{States}(P) \times Act(P) \times \mathsf{Disc}(\mathsf{States}(P))$.

An action is *visible* if it is not hidden. It is *locally controlled* if it is non-input (i.e., either output or hidden); we define $L := O \cup H$. We write $s \xrightarrow{a} \mu$ for $\langle s, a, \mu \rangle \in \rightarrow$, and $s \xrightarrow{a} s'$ if there exists μ with $s \xrightarrow{a} \mu$ and $s' \in \mathsf{Supp}(\mu)$. A state is *quiescent* if it enables only input actions. A PIOA is *closed* if its set of input actions is empty. As with I/O automata, we always assume *input enabling*: $\forall s \in \mathsf{States}(P)\ \forall a \in I\ \exists \mu : s \xrightarrow{a} \mu$.

An *execution* of P is a (possibly finite) sequence $p = s_0 a_1 \mu_1 s_1 a_2 \mu_2 s_2 \ldots$, such that:

- each s_i (resp., a_i, μ_i) denotes a state (resp., action, distribution over states);
- $s_0 = s^0$ and, if p is finite, then p ends with a state;
- for each non-final i, $s_i \xrightarrow{a_{i+1}} \mu_{i+1}$ and $s_{i+1} \in \mathsf{Supp}(\mu_{i+1})$.

In some literature, executions are defined to be sequences of states and actions in alternating fashion, thus omitting the target distributions. We adopt the current style for a more straightforward generalization to probabilistic executions.

Given a finite execution p, we use $\mathsf{last}(p)$ to denote the last state of p. A state s is *reachable* if there exists a finite execution p such that $\mathsf{last}(p) = s$. We write $\mathsf{Exec}(P)$ for the set of all executions of P and $\mathsf{Exec}^{<\omega}(P)$ for the set of finite executions. Given an execution p, the sequence of visible action symbols in p is called the *(visible) trace* of p, denoted $\mathsf{tr}(p)$.

A finite set of PIOAs $\{P_1, \ldots, P_n\}$ is said to be *compatible* if for all $i \neq j$, $O_i \cap O_j = Act(P_i) \cap H_j = \emptyset$. Such a set is *closed* if $\bigcup_{1 \leq i \leq n} I_i \subseteq \bigcup_{1 \leq i \leq n} O_i$. We define $P = \|_{1 \leq i \leq n} P_i$ as usual with synchronization of shared actions:

- $\mathsf{States}(P) := \prod_{1 \leq i \leq n} \mathsf{States}(P_i)$ and the start state of P is $\langle s_1^0, \ldots, s_n^0 \rangle$;
- $I := \bigcup_{1 \leq i \leq n} I_i \setminus \bigcup_{1 \leq i \leq n} O_i$, $O := \bigcup_{1 \leq i \leq n} O_i$, and $H := \bigcup_{1 \leq i \leq n} H_i$;
- given a state $\langle s_1, \ldots, s_n \rangle$, an action a and a target distribution μ, there is a transition $\langle s_1, \ldots, s_n \rangle \xrightarrow{a} \mu$ if and only if μ is of the form $\mu_1 \times \ldots \times \mu_n$ and for all $1 \leq i \leq n$,
  - either $a \in Act(P_i)$ and $s_i \xrightarrow{a} \mu_i$,
  - or $a \notin Act(P_i)$ and $\mu_i = (s_i \mapsto 1)$.

Notice $\|$ is commutative and associative for PIOAs (up to isomorphism).

The notion of (probabilistic) schedulers for a PIOA P is introduced as a means to resolve all nondeterministic choices in P. Each scheduler consists of an input component and an output component. Given a finite history of the automaton, the output scheduler chooses probabilistically the next locally controlled transition, whereas the input scheduler responds to inputs from the environment and chooses probabilistically a transition carrying the correct input symbol.

Definition 1. *An* input scheduler σ *for* P *is a function*

$$\sigma : \mathsf{Exec}^{<\omega}(P) \times I \longrightarrow \mathsf{Disc}(\rightarrow)$$

such that for all $\langle p, a \rangle \in \mathsf{Exec}^{<\omega}(P) \times I$ *and transitions* $(s \xrightarrow{b} \mu) \in \mathsf{Supp}(\sigma(p, a))$, *we have* $s = \mathsf{last}(p)$ *and* $b = a$. *An* output scheduler ρ *for* P *is a function*

$$\rho : \mathsf{Exec}^{<\omega}(P) \longrightarrow \mathsf{SubDisc}(\rightarrow)$$

such that for all $p \in \mathsf{Exec}^{<\omega}(P)$ *and transition* $(s \xrightarrow{a} \mu) \in \mathsf{Supp}(\rho(p))$, *we have* $s = \mathsf{last}(p)$ *and* $a \in L$. *An* I/O *scheduler for* P *is then a pair* $\langle \sigma, \rho \rangle$ *where* σ *is an input scheduler for* P *and* ρ *is an output scheduler for* P.

Notice input schedulers must return a discrete probability distribution, reflecting the requirement that each input issued by the environment is received

with probability 1. (This is always possible because of the input enabling assumption.) In contrast, output schedulers may choose to halt with an arbitrary probability θ by returning a proper sub-distribution whose total probability mass is $1-\theta$. Finally, we write $\sigma(p,a)(\mu)$ as a shorthand for $\sigma(p,a)(\mathsf{last}(p) \xrightarrow{a} \mu)$ and $\rho(p)(a,\mu)$ for $\rho(p)(\mathsf{last}(p) \xrightarrow{a} \mu)$.

Consider a closed PIOA P. Obviously, any I/O scheduler for P has a trivial input component (i.e., the empty function). Every output scheduler ρ thus induces a purely probabilistic behavior, which is captured by the following notion of probabilistic executions. The *probabilistic execution* induced by ρ is the function $Q_\rho : \mathsf{Exec}^{<\omega}(P) \longrightarrow [0,1]$ defined recursively by:

- $Q_\rho(s^0) := 1$, where s^0 is the initial state of P;
- $Q_\rho(p') := Q_\rho(p) \cdot \rho(p)(a,\mu) \cdot \mu(s')$, where p' is of the form $pa\mu s'$.

A probabilistic execution Q_ρ induces a probability space over the sample space $\Omega_P := \mathsf{Exec}(P)$ as follows. Let $\sqsubseteq$ denote the prefix ordering on sequences. Each $p \in \mathsf{Exec}^{<\omega}(P)$ generates a *cone* of executions: $\mathbf{C}_p := \{p' \in \mathsf{Exec}(P) \mid p \sqsubseteq p'\}$. Let $\mathcal{F}_P$ denote the smallest σ-field generated by the collection $\{\mathbf{C}_p \mid p \in \mathsf{Exec}^{<\omega}(P)\}$. There exists a unique measure $\mathbf{m}_\rho$ on $\mathcal{F}_P$ with $\mathbf{m}_\rho[\mathbf{C}_p] = Q_\rho(p)$ for all p in $\mathsf{Exec}^{<\omega}(P)$; therefore Q_ρ gives rise to a probability space $(\Omega_P, \mathcal{F}_P, \mathbf{m}_\rho)$.

Trace distributions are obtained from probabilistic executions by removing non-visible elements. In our case, these are states, hidden actions and distributions of states. To state this precisely, we need the notion of minimal executions: a finite execution p of P is said to be *minimal* if every proper prefix of p has a strictly shorter trace. Notice, the empty execution (i.e., the sequence containing just the initial state) is minimal. Moreover, if p is nonempty and finite, then p is minimal if and only if the last transition in p has a visible action label. For each $\alpha \in Act(P)^{<\omega}$, let $\mathsf{tr}^{-1}_{\min}(\alpha)$ denote the set of minimal executions of P with trace α.

Now we define a lifting of the trace operator $\mathsf{tr} : \mathsf{Exec}^{<\omega}(P) \longrightarrow Act(P)^{<\omega}$. Given a function $Q : \mathsf{Exec}^{<\omega}(P) \longrightarrow [0,1]$, define $\mathsf{tr}(Q) : Act(P)^{<\omega} \longrightarrow [0,1]$ by

$$\mathsf{tr}(Q)(\alpha) := \sum_{p \in \mathsf{tr}^{-1}_{\min}(\alpha)} Q(p).$$

Given an output scheduler ρ of a closed PIOA P, the *trace distribution* induced by ρ (denoted D_ρ) is simply the result of applying tr to the probabilistic execution Q_ρ. That is, $D_\rho := \mathsf{tr}(Q_\rho)$. We often use variables D, D', etc. for trace distributions, thus leaving the scheduler ρ implicit.

Similar to the case of probabilistic executions, each D_ρ induces a probability measure on the sample space $\Omega := Act(P)^{\leq\omega}$. There the σ-field $\mathcal{F}$ is generated by the collection $\{\mathbf{C}_\alpha \mid \alpha \in Act(P)^{<\omega}\}$, where $\mathbf{C}_\alpha := \{\alpha' \in \Omega \mid \alpha \sqsubseteq \alpha'\}$. The measure $\mathbf{m}^\rho$ on $\mathcal{F}$ is uniquely determined by the equations $\mathbf{m}^\rho[\mathbf{C}_\alpha] = D_\rho(\alpha)$ for all $\alpha \in Act(P)^{<\omega}$.

In the literature, most authors define probabilistic executions (resp. trace distributions) to be the probability spaces $\langle \Omega_P, \mathcal{F}_P, \mathbf{m}_\rho \rangle$ (resp. $\langle \Omega, \mathcal{F}, \mathbf{m}^\rho \rangle$). Here we find it more natural to reason with the functions Q_ρ and D_ρ, rather

than the induced measures. We refer to [Seg95] for these alternative definitions and proofs that they are equivalent to our versions.

3 Switched Probabilistic I/O Automata

As we argued in Section 1, one must distinguish between global and local nondeterministic choices and must resolve them separately. This section describes our solution, namely, an explicit mechanism of control exchange among parallel components. The presentation is organized as follows: (i) first we define *pre-switched automata*, where we describe control action signatures and the Boolean-valued state variable active; (ii) then we introduce the notion of *input well-behaved executions* of a pre-switched automaton and state four axioms defining *switched automata*; (iii) finally, we introduce the notion of a *scheduled automaton*, essentially a switched automaton paired with an I/O scheduler.

For technical simplicity, we assume a universal set Act of action symbols such that $Act(P) \subseteq Act$ for every PIOA P. Moreover, Act is partitioned into two sets: $BAct$ (*basic actions*) and $CAct$ (*control actions*). Both sets are assumed to be countably infinite, so we can rename hidden actions using fresh symbols whenever necessary (cf. Section 5).

Definition 2. *A pre-switched automaton P is a PIOA endowed with a function* $\mathsf{active} : \mathsf{States}(P) \longrightarrow \{0,1\}$ *and a set* $Sync \subseteq O \cap CAct$ *of* synchronized control actions.

We use variables P, Q, etc. to denote pre-switched automata. Given a pre-switched automaton P, we further classify its action symbols:

- $BI := I \cap BAct$ (*basic inputs*);
- $BO := O \cap BAct$ (*basic outputs*);
- $CI := I \cap CAct$ (*control inputs*);
- $CO := (O \cap CAct) \setminus Sync$ (*control outputs*).

Essentially, we have a partition $\{BI, BO, H, CI, CO, Sync\}$ of $Act(P)$. We say that P is *initially active* if $\mathsf{active}(s^0) = 1$. Otherwise, it is *initially inactive*.

As described in Section 1, the Boolean-valued function active on the states of P indicates whether P is active or inactive, while control actions allow P to exchange control with its environment. The designation of synchronized control actions helps to achieve the "handshake" condition on control synchronizations: whenever we compose two automata, we classify the shared control actions as "synchronized", so that they are no longer available for further synchronization with a third component. This is made precise in the definitions of compatibility and composition for pre-switched automata.

A finite set of pre-switched automata $\{P_1, \ldots, P_n\}$ is said to be *compatible* if (i) $\{P_1, \ldots, P_n\}$ is a compatible set of PIOAs; (ii) for all $i \neq j$, $Act(P_i) \cap Sync_j = CI_i \cap CI_j = \emptyset$; (iii) at most one P_i is initially active. Notice that such a set is

compatible if and only if for all $i \neq j$, P_i and P_j are compatible. The *parallel composition* of $\{P_1, \ldots, P_n\}$, denoted $\|_{1 \leq i \leq n} P_i$, is the result of composing $P_1, \ldots, P_n$ as PIOAs, together with:

- $Sync := \bigcup_{1 \leq i \leq n} Sync_i \cup \bigcup_{1 \leq i,j \leq n} (CI_i \cap CO_j)$;
- $\mathsf{active}(s_1, \ldots, s_n) = 1$ if and only if for some i, $\mathsf{active}_i(s_i) = 1$.

Clearly, the composite $\|_{1 \leq i \leq n} P_i$ is again a pre-switched automaton. In the binary case, we write $P_1 \| P_2$ as shorthand for $\|_{1 \leq i \leq 2} P_i$. Observe that $P_1 \| P_2 \cong P_2 \| P_1$; that is, composition of pre-switched automata is commutative up to isomorphism. Next we check that composition is also associative on the class of pre-switched automata.

Lemma 1. *Let P_1, P_2 and P_3 be pre-switched automata. Assume P_1 is compatible with P_2, and P_3 is compatible with $P_1 \| P_2$. Then P_2 is compatible with P_3, and P_1 is compatible with $P_2 \| P_3$. Moreover, $(P_1 \| P_2) \| P_3 \cong P_1 \| (P_2 \| P_3)$.*

Recall that switched automata are intended to be composed in such a way that at most one component is active at any point of an execution. In particular, any environment automaton must also follow the rules of control exchange; that is, after activating some system component, the environment must itself become inactive. This leads to the definition of *input well-behavedness*. Let P be a pre-switched automaton. An input transition $s \xrightarrow{a} \mu$ is *well-behaved* if $\mathsf{active}(s) = 0$. An execution p of P is *input well-behaved* if all input transitions occurring in p are well-behaved. Let $\mathsf{Exec}^{<\omega}_{\mathsf{iwb}}(P)$ denote the set of finite, input well-behaved executions of P. Moreover, we say that a state s is *input well-behaved reachable*, notation $\mathsf{iwbr}(s)$, if there exists an input well-behaved execution p such that $s = \mathsf{last}(p)$. Clearly, the empty execution is input well-behaved and thus the initial state is always input well-behaved reachable. If P is closed (i.e., $I = \emptyset$), then every execution of P is trivially input well-behaved and every reachable state is input well-behaved reachable. We are now prepared to define the notion of switched probabilistic I/O automata.

Definition 3. *A* switched (probabilistic I/O) automaton *is a pre-switched automaton P that satisfies the following axioms.*

$$s \xrightarrow{a} \mu \ \wedge \ \mathsf{active}(s) = 0 \quad \Rightarrow \quad a \in I \tag{1}$$

$$s \xrightarrow{a} s' \ \wedge \ a \in CI \quad \Rightarrow \quad \mathsf{active}(s') = 1 \tag{2}$$

$$s \xrightarrow{a} s' \ \wedge \ a \notin CI \cup CO \quad \Rightarrow \quad \mathsf{active}(s) = \mathsf{active}(s') \tag{3}$$

$$\mathsf{iwbr}(s) \ \wedge \ s \xrightarrow{a} s' \ \wedge \ a \in CO \quad \Rightarrow \quad \mathsf{active}(s') = 0 \tag{4}$$

These four axioms formalize our intuitions about control passage. Axiom (1) requires all inactive states to be quiescent. Axioms (2) and (4) say that control inputs lead to active states and control outputs to inactive states. Axiom (3) says that non-control transitions and synchronized control transitions do not change the activity status. Together, they describe an "activity cycle" for the automaton P: (i) while in inactive mode, P does not enable locally controlled

transitions, although it may still receive inputs from its environment; (ii) when P receives a control input it moves into active mode, where it may perform hidden or output transitions, possibly followed by a control output; (iii) via this control output P returns to inactive mode.

Notice that Axiom (4) is required for input well-behaved reachable states only. Without this relaxation, the composition of two switched automata may not satisfy Axiom (4).

We proceed to confirm that the class of switched automata is closed under the parallel composition operator for pre-switched automata. A set $\{P_1, \ldots, P_n\}$ of switched automata is *compatible* if the set of underlying pre-switched automata is compatible. Define the composite, $\|_{1 \leq i \leq n} P_i$, to be the result of composing the switched automata as pre-switched automata. The first three axioms can be verified by unfolding the definition of active in a composition and applying appropriate axioms for the components. Axiom (4) follows from Lemma 2 below. The proof is by induction on the length of executions and relies heavily on invariant-style reasoning based on the definition of input well-behaved executions and the axioms of switched automata.

Lemma 2. *Let $\{P_1, \ldots, P_n\}$ be a compatible set of switched automata. For each finite, input well-behaved execution p of $\|_{1 \leq i \leq n} P_i$, we have:*

(i) for all i, $\pi_i(p)$ is also input well-behaved;
(ii) there is at most one i such that $\mathsf{active}_i(\pi_i(\mathsf{last}(p))) = 1$.

To summarize, $\|_{1 \leq i \leq n}$ is a well-defined n-ary operator for switched automata and, in the binary case, associativity of $\|$ follows from Lemma 1.

Next we turn to scheduling decisions. The notion of I/O schedulers for switched automata is inherited from that of its underlying PIOA.

Definition 4. *A* scheduled *automaton is a triple $\langle P, \sigma, \rho \rangle$ such that P is a switched automaton and $\langle \sigma, \rho \rangle$ is an I/O scheduler for P.*

We use letters S, T, etc. to denote scheduled automata. For each $1 \leq i \leq n$, let S_i denote a scheduled automaton $\langle P_i, \sigma_i, \rho_i \rangle$. The set $\{S_i \mid 1 \leq i \leq n\}$ is said to be *compatible* if $\{P_i \mid 1 \leq i \leq n\}$ is compatible as a set of switched automata. Given such a compatible set of scheduled automata, we obtain its composite by combining the I/O schedulers $\{\langle \sigma_i, \rho_i \rangle \mid 1 \leq i \leq n\}$ into an I/O scheduler $\langle \sigma, \rho \rangle$ for the switched automaton $\|_{1 \leq i \leq n} P_i$.

Definition 5. *Suppose $\{S_i \mid 1 \leq i \leq n\}$ is a compatible set of scheduled automata, where $S_i = \langle P_i, \sigma_i, \rho_i \rangle$ for each i. We construct from this set a* composite *scheduled automaton $\|_{1 \leq i \leq n} S_i := \langle P, \sigma, \rho \rangle$ as follows.*

- $P := \|_{1 \leq i \leq n} P_i$.
- *For every finite execution p of P with $\mathsf{last}(p) = s$ and for every $a \in I$,*
  - $\sigma(p, a)(t \xrightarrow{b} \mu) := 0$ *if $t \neq s$ or $b \neq a$;*
  - *otherwise, $\sigma(p, a)(s \xrightarrow{a} \mu_0 \times \ldots \times \mu_n) := \Pi_i c_i$, where c_i equals*

  - $*$ $\sigma_i(\pi_i(p), a)(\mu_i)$, *if* $a \in I_i$;
  - $*$ 1, *otherwise.*
- *For every finite execution* p *of* P *with* $\mathsf{last}(p) = s$,
  - $\rho(p)(t \xrightarrow{a} \mu) := 0$ *if* p *is not input well-behaved,* $t \neq s$, *or* $a \notin L$;
  - *otherwise,* $\rho(p)(s \xrightarrow{a} \mu_0 \times \ldots \times \mu_n) := \Pi_i c_i$, *where* c_i *equals*
    - $*$ $\rho_i(\pi_i(p))(a, \mu_i)$, *if* $a \in L_i$;
    - $*$ $\sigma_i(\pi_i(p), a)(\mu_i)$ *if* $a \in I_i$;
    - $*$ 1, *otherwise.*

It is routine to check that $\sigma(p, a)$ is a discrete distribution for all $p \in \mathsf{Exec}^{<\omega}(P)$ and $a \in I$. Lemma 2 guarantees that, at the end of every input well-behaved finite execution p, there is at most one i such that component i enables a locally controlled transition. This allows us to conclude that $\rho(p)$ is a discrete sub-distribution for all $p \in \mathsf{Exec}^{<\omega}(P)$.

As usual, we write $S_1 \| S_2$ for $\|_{1 \leq i \leq 2} S_i$, provided S_1 and S_2 are compatible. Associativity of $\|$ for scheduled automata follows from that for switched automata and a routine check on the I/O schedulers. Finally, the notions of probabilistic executions and trace distributions for closed scheduled automata are inherited from those of PIOAs. In particular, we write Q_S (respectively, D_S) for the probabilistic execution (respectively, trace distribution) induced by the output scheduler of a closed scheduled automaton S.

4 Projection and Pasting

In this section, we study projection and pasting of probabilistic behaviors. Such results are essential elements in constructing a compositional modeling framework. We begin by introducing the notion of regular executions, which will be used to define pseudo trace distributions for automata with open inputs. In Lemma 5, we prove that the pseudo distribution of a composite is uniquely determined by those of its components. Finally, we prove the main pasting lemma for closed automata (Lemma 7), which plays a crucial role in the proof of our main compositionality theorem (Theorem 1).

Given an execution p of a switched automaton P, we say that p is *regular* if it is both minimal and input well-behaved. Given a finite sequence α of visible actions in P, let $\mathsf{tr}^{-1}_{\mathsf{reg}}(\alpha)$ denote the set of regular executions of P with trace α. Notice that regularity coincides with minimality in case P is closed.

Lemma 3 states that, given a fixed trace, there is a bijective correspondence between the set of regular executions of the composite and the Cartesian product of the sets of regular executions of the two components.

Lemma 3. *Let* X *denote* $\mathsf{tr}^{-1}_{\mathsf{reg}}(\alpha)$ *in* $P_1 \| P_2$. *Let* Y *and* Z *denote* $\mathsf{tr}^{-1}_{\mathsf{reg}}(\pi_1(\alpha))$ *in* P_1 *and* $\mathsf{tr}^{-1}_{\mathsf{reg}}(\pi_2(\alpha))$ *in* P_2, *respectively. There exists an isomorphism* $\mathsf{zip} : Y \times Z \longrightarrow X$ *whose inverse is* $\langle \pi_1, \pi_2 \rangle$.

Next we introduce a notion of *pseudo* probabilistic execution for automata with open inputs. The definition itself is completely analogous to probabilistic

executions for closed automata; however, a pseudo probabilistic execution does not always induce a probability measure, because it does not take into account the probabilities with which inputs are provided by the environment.

Definition 6. *Let $S = \langle P, \sigma, \rho \rangle$ be a scheduled automaton. Define the* pseudo probabilistic execution *Q of S as follows: for all finite executions p' of S,*

- *if p' is of the form s^0, where s^0 is the initial state of S, then $Q(p') := 1$;*
- *if p' is of the form $pa\mu s'$ with $a \in I$, then $Q(p') := Q(p) \cdot \sigma(p, a)(\mu) \cdot \mu(s')$;*
- *if p' is of the form $pa\mu s'$ with $a \in L$, then $Q(p') := Q(p) \cdot \rho(p)(a, \mu) \cdot \mu(s')$.*

Similarly, we define *pseudo* trace distributions.

Definition 7. *Let $S = \langle P, \sigma, \rho \rangle$ be a scheduled automaton. The* pseudo trace distribution *D of S is the function from $(Act(S) \setminus H_S)^{<\omega}$ to $[0, 1]$ given by $D(\alpha) := \sum_{p \in \mathsf{tr}_{\mathsf{reg}}^{-1}(\alpha)} Q(p)$, where Q is the pseudo probabilistic execution of S.*

Notice that, if S is closed, then the pseudo probabilistic execution of S coincides with the probabilistic execution of S. Moreover, an execution of a closed automaton S is regular if and only if it is minimal, thus the pseudo trace distribution of S coincides with the trace distribution of S.

For the rest of this section, let S and T be a pair of compatible scheduled automata. Let $Q_{S\|T}$, Q_S and Q_T denote the pseudo probabilistic executions of $S\|T$, S and T, respectively. Similarly for pseudo trace distributions $D_{S\|T}$, D_S and D_T. Lemma 4 below says we can project a pseudo probabilistic execution of the composite to yield pseudo probabilistic executions of the components. The proof is routine, by induction on the length of executions. Lemma 5 then combines Lemma 3 and Lemma 4 to show the analogous projection result for pseudo trace distributions.

Lemma 4. *For all finite executions p of $S\|T$, we have $Q_{S\|T}(p) = Q_S(\pi_1(p)) \cdot Q_T(\pi_2(p))$.*

Lemma 5. *Let α be a finite sequence of visible action symbols of $S\|T$. Then $D_{S\|T}(\alpha) = D_S(\pi_1(\alpha)) \cdot D_T(\pi_2(\alpha))$.*

To prove the main pasting lemma, we need yet another technical result; namely, inputs must be received with probability 1. This can be viewed as "input enabling" in the probabilistic sense and it follows from basic properties of target distributions and input schedulers.

Lemma 6. *Let α be a finite sequence of visible action symbols of $S\|T$ and let $a \in Act(S\|T)$ be given. If a is not locally controlled by T, then $D_T(\pi_2(\alpha)) = D_T(\pi_2(\alpha a))$.*

Two switched/scheduled automata are said to be *comparable* if they have the same visible signature and their start states have the same status. We are now ready for the main pasting lemma.

Lemma 7 (Pasting). *Let S_1, S_2, T_1 and T_2 be scheduled automata satisfying (i) S_1 and S_2 are comparable; (ii) $\{S_1, T_1\}$, $\{S_2, T_2\}$ and $\{S_1, T_2\}$ are compatible sets; (iii) the pseudo trace distributions $D_{S_1\|T_1}$ and $D_{S_2\|T_2}$ coincide (denoted D). Then D also coincides with the pseudo trace distribution $D_{S_1\|T_2}$.*

5 Renaming and Hiding

In this section, we consider the standard renaming and hiding operators. We start with an equivalence relation on switched automata: $P_1 \equiv_R P_2$ just in case there exists a bijection $f : H_1 \longrightarrow H_2$ such that P_2 can be obtained from P_1 by replacing every hidden action symbol $a \in H_1$ by $f(a) \in H_2$ (notation: $P_2 = f(P_1)$).

It is routine to check this is in fact an equivalence relation. If $P_1 \equiv_R P_2$, we say that P_2 can be obtained from P_1 by *renaming of hidden actions*. This operation also induces an equivalence relation on scheduled automata: $\langle P_1, \sigma_1, \rho_1\rangle \equiv_R \langle P_2, \sigma_2, \rho_2\rangle$ just in case there exists a renaming function f such that $P_1 \equiv_R P_2$ via f and $\langle \sigma_2, \rho_2\rangle$ is obtained from $\langle \sigma_1, \rho_1\rangle$ via f and f^{-1} (notation: $S_2 = f(S_1)$).

The following lemma says, as long as the renaming operation does not introduce incompatibility of signatures, it does not affect the behavior of an automaton in a closing context.

Lemma 8. *Let S and C be compatible scheduled automata with $S\|C$ closed. Suppose $S \equiv_R S'$ via the renaming function $f : H \longrightarrow H'$ with H' disjoint from $Act(C)$. Then $\{S', C\}$ is closed and compatible and $D_{S\|C} = D_{S'\|C}$.*

Next we consider the issue of hiding output actions. Let Hide denote the standard hiding operator for PIOA. This is also an operator for switched automata, provided we hide only basic outputs and synchronized control actions.

Lemma 9. *Let P be a switched automaton and let $\Omega \subseteq BO \cup Sync$ be given. Then $\mathsf{Hide}_\Omega(P)$ is again a switched automaton.*

Notice that every I/O scheduler for P is an I/O scheduler for $\mathsf{Hide}_\Omega(P)$. Therefore Hide can be extended to scheduled automata:

$$\mathsf{Hide}_\Omega\langle P, \sigma, \rho\rangle := \langle \mathsf{Hide}_\Omega(P), \sigma, \rho\rangle.$$

In the rest of this section we investigate the effect of Hide_Ω on (pseudo) trace distributions. Let $S = \langle P, \sigma, \rho\rangle$ be a scheduled automaton with signature $\langle I, O, H\rangle$. For convenience, write P' for $\mathsf{Hide}_\Omega(P)$, O' for $O \setminus \Omega$, and tr' for the trace operator for $\mathsf{Hide}_\Omega(P)$. (If we view Hide_Ω as an operator on traces, then tr' is precisely $\mathsf{Hide}_\Omega \circ \mathsf{tr}$.)

Moreover, for all $\beta' \in (I \cup O')^{<\omega}$, let $\mathcal{M}_{\beta'}$ denote the set of all minimal (w.r.t. $\sqsubseteq$) traces in $\mathsf{Hide}_\Omega{}^{-1}(\beta')$. That is, if β' is empty, then $\mathcal{M}_{\beta'}$ is the singleton set containing the empty trace ϵ; otherwise,

$$\mathcal{M}_{\beta'} := \{\beta \in (I \cup O)^{<\omega} \mid \mathsf{Hide}_\Omega(\beta) = \beta' \text{ and the last symbol on } \beta \text{ is not in } \Omega.\}$$

We make a simple observation about minimal executions of P and those of P'.

Lemma 10. *For all $\beta' \in (I \cup O')^{<\omega}$, the following two sets are equal:*

- $X := \bigcup_{\beta \in \mathcal{M}_{\beta'}} \{p \in \mathsf{Exec}^{<\omega}(P) \mid \mathsf{tr}(p) = \beta$ *and* p *minimal w.r.t.* $\mathsf{tr}\}$;
- $Y := \{p \in \mathsf{Exec}^{<\omega}(P') \mid \mathsf{tr}'(p) = \beta'$ *and* p *minimal w.r.t.* $\mathsf{tr}'\}$.

Now consider the pseudo trace distribution D_S. Define the effect of Hide_Ω on D_S to be the following function from $O'^{<\omega}$ to $[0, 1]$:

$$\mathsf{Hide}_\Omega(D_S)(\beta') := \sum_{\beta \in \mathcal{M}_{\beta'}} D_S(\beta).$$

We have the following corollary of Lemma 10.

Corollary 1. *The pseudo trace distribution of* $\mathsf{Hide}_\Omega(S)$ *is precisely* $\mathsf{Hide}_\Omega(D_S)$. *That is,* $D_{\mathsf{Hide}_\Omega(S)} = \mathsf{Hide}_\Omega(D_S)$.

Finally, we consider the effect of hiding in a parallel composition. We claim that the act of hiding in one component does not affect the behavior of the other, as long as the actions being hidden in the first component are not observable by the second component. This idea is captured in the following lemma, which follows from Corollary 1 and Lemma 5.

Lemma 11. *Let S_1, S_2, T be scheduled automata satisfying: (i) S_1 and S_2 are comparable and (ii) T is compatible with S_1 and S_2. Let $\Omega \subseteq O_T$ be given and let T' denote* $\mathsf{Hide}_\Omega(T)$. *If T' is compatible with S_1 (and thus with S_2), then*

$$D_{S_1 \| T} = D_{S_2 \| T} \Leftrightarrow D_{S_1 \| T'} = D_{S_2 \| T'}.$$

6 Probabilistic Systems

In this section, we give a formal definition of our implementation preorder and prove compositionality. The basic approach is to model a system as a switched PIOA together with a set of I/O schedulers. Observable behavior is then defined in terms of trace distributions induced by the prescribed schedulers.

Formally, a *probabilistic system* $\mathcal{P}$ is a set of scheduled automata that share a common underlying switched automaton. (Equivalently, a probabilistic system is a pair $\langle P, \mathcal{S} \rangle$ where P is a switched automaton and $\mathcal{S}$ is a set of I/O schedulers for P.) Such a system is *full* if $\mathcal{S}$ is the set of all possible I/O schedulers for P.

Two probabilistic systems $\mathcal{P}_1 = \langle P_1, \mathcal{S}_1 \rangle$ and $\mathcal{P}_2 = \langle P_2, \mathcal{S}_2 \rangle$ are *compatible* just in case P_1 is compatible with P_2. The *parallel composite* of $\mathcal{P}_1$ and $\mathcal{P}_2$, denoted $\mathcal{P}_1 \| \mathcal{P}_2$, is the probabilistic system: $\{S_1 \| S_2 \mid S_1 \in \mathcal{P}_1$ and $S_2 \in \mathcal{P}_2\}$. Notice the underlying automaton of the composite is $P_1 \| P_2$.

Let S be a scheduled automaton. A *context* for S is a scheduled automaton C such that (i) C is compatible with S; (ii) S and C have complementary signatures (i.e., $I_C = O_S$ and $I_S = O_C$). Given probabilistic system $\mathcal{P} = \langle P, \mathcal{S} \rangle$, we say that D is a *trace distribution of* $\mathcal{P}$ just in case there exist scheduled automata

$S \in \mathcal{P}$ and context C for S such that $D = D_{S\|C}$. We write $\mathsf{td}(\mathcal{P})$ for the set of trace distributions of $\mathcal{P}$.

Two probabilistic systems are *comparable* whenever the underlying switched automata are comparable. Given comparable systems $\mathcal{P}_1$ and $\mathcal{P}_2$, we define the *trace distribution preorder* by: $\mathcal{P}_1 \leq_{\mathsf{td}} \mathcal{P}_2$ whenever $\mathsf{td}(\mathcal{P}_1) \subseteq \mathsf{td}(\mathcal{P}_2)$. We are now ready to present our main theorem, namely, that the trace distribution preorder for probabilistic systems is compositional.

Theorem 1. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be comparable probabilistic systems with $\mathcal{P}_1 \leq_{\mathsf{td}} \mathcal{P}_2$. Suppose $\mathcal{P}_3$ is compatible with both $\mathcal{P}_1$ and $\mathcal{P}_2$. Then $\mathcal{P}_1\|\mathcal{P}_3 \leq_{\mathsf{td}} \mathcal{P}_2\|\mathcal{P}_3$.*

7 PIOA Revisited

Before concluding, we give an example in which switched automata are used to obtain a new trace-based semantics for general PIOAs. The idea is to convert a general PIOA to a switched PIOA by adding control actions and activity classification. We then hide all control actions in trace distributions generated by the resulting switched PIOA. In many cases, this yields a set of trace distributions strictly smaller than that considered by Segala [Seg95].

Let P be a PIOA and assume $Act(P) \subseteq BAct$. Let $\mathsf{go}, \mathsf{done} \in CAct$ be fresh symbols and let b_0 be a Boolean value. The *switch extension* of P with go, done and initialization b_0 (notation: $\mathcal{E}(P, \mathsf{go}, \mathsf{done}, b_0)$), is the switched automaton P' constructed as follows:

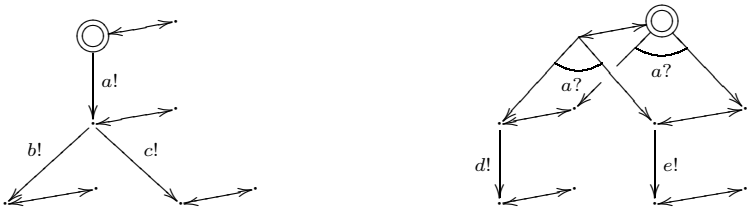
- $\mathsf{States}(P') = \mathsf{States}(P) \times \{0, 1\}$ and the start state of P' is $\langle s^0, b_0 \rangle$;
- $I' = I \cup \{\mathsf{go}\}$, $O' = O \cup \{\mathsf{done}\}$, and $Sync' = \emptyset$;
- $\mathsf{active}'(\langle s, b \rangle) = b$ for $b \in \{0, 1\}$;
- the transition relation is the union of the following:
  - $\{\langle \langle s, 1 \rangle, a, \mu^1 \rangle \mid s \xrightarrow{a} \mu \text{ in } P\}$,
  - $\{\langle \langle s, 0 \rangle, a, \mu^0 \rangle \mid s \xrightarrow{a} \mu \text{ in } P \text{ and } a \in I\}$,
  - $\{\langle \langle s, b \rangle, \mathsf{go}, (\langle s, 1 \rangle \mapsto 1) \rangle \mid s \in \mathsf{States}(P) \text{ and } b \in \{0, 1\}\}$,
  - $\{\langle \langle s, 1 \rangle, \mathsf{done}, (\langle s, 0 \rangle \mapsto 1) \rangle \mid s \in \mathsf{States}(P)\}$,

  where μ^b denotes the distribution that assigns probability $\mu(t)$ to $\langle t, b \rangle$ and 0 to $\langle t, 1 - b \rangle$.

Roughly speaking, P' is obtained from P by (i) adding a Boolean flag active' to each state; (ii) enabling locally controlled transitions only if $\mathsf{active}' = 1$; and (iii) adding go and done transitions which update active' appropriately. It is not hard to check that P' satisfies all axioms of switched automata. Moreover, the pair $\langle \mathsf{go}, \mathsf{done} \rangle$ can be easily generalized to a pair of disjoint sets of control actions.

Given any two compatible PIOAs, we can always extend them with complementary control actions and initialization statuses, resulting in a pair of compatible switched automata. As an example, we consider the automata Late and Toss in Figure 1. Actions a, b and c are considered outputs of Late, whereas

action a is an input of Toss and actions e and f are outputs of Toss. The following diagrams illustrate $\mathcal{E}(\mathsf{Late}, \mathsf{go}, \mathsf{done}, 1)$ and $\mathcal{E}(\mathsf{Toss}, \mathsf{done}, \mathsf{go}, 0)$. For a clearer picture, we have omitted the probabilities on the input a-transition in Toss, as well as all nonessential input loops. The active region, which is identical to the original PIOA, is drawn in the foreground. The inactive region, in which all locally controlled transitions are removed, is in the background. Each two-headed arrow indicates a control output from active to inactive and a control input from inactive to active.



Now consider the problematic trace distribution D_0 of $\mathsf{Late} \,\|\, \mathsf{Toss}$, as described in Section 1. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ denote the full probabilistic systems on $\mathcal{E}(\mathsf{Late}, \mathsf{go}, \mathsf{done}, 1)$ and $\mathcal{E}(\mathsf{Toss}, \mathsf{done}, \mathsf{go}, 0)$, respectively. As we compose these two systems, D_0 is no longer a trace distribution of $\mathcal{P}_1 \| \mathcal{P}_2$ (even after hiding go and done), because I/O schedulers in $\mathcal{P}_1$ have no way of knowing whether action d or action e was performed by $\mathcal{P}_2$, thus they cannot establish the correlations between actions d and b, and between actions e and c.

This leads to our proposal of a new notion of visible behaviors for PIOA. Let P be a PIOA and let $\mathcal{P}$ be the full probabilistic system on $\mathcal{E}(P, \mathsf{go}, \mathsf{done}, 0)$. A PIOA E is a *context* for P if $I_E = O_P$, $O_E = I_P$, and E is compatible with P. For each such E, write $\mathcal{P}_E$ for the full probabilistic system on $\mathcal{E}(E, \mathsf{done}, \mathsf{go}, 1)$. We say that D is a *trace distribution* of P if there exists a context E for P such that $D \in \mathsf{td}(\mathsf{Hide}_{\{\mathsf{go},\mathsf{done}\}}(\mathcal{P} \| \mathcal{P}_E))$, where Hide is lifted from scheduled automata to probabilistic systems.

8 Conclusions and Further Work

Our ultimate goal, of course, is to obtain a compositional semantics for PIOAs. The notion of switch extensions opens up an array of new options for that end. A promising approach is to model each system as a finite set of PIOAs, rather than a single PIOA. Composition is taken to be set union, representing the act of placing two sets of processes in the same computing environment. Behavior is then defined in terms of switch extensions, which instantiate the system with a particular network topology for control passage. In that case, a behavior of a finite set $\mathcal{F}$ is determined by (i) a context E for $\mathcal{F}$; (ii) a combination of switch extensions of $\mathcal{F} \cup \{E\}$; (iii) a combination of I/O schedulers for these switch extensions. By ranging over all contexts and all extension-

scheduler combinations, we capture all possible behaviors of the system represented by $\mathcal{F}$.

In other future work, we plan to apply our theory of composition for PIOAs to the task of verifying security protocols. For example, we will try to model typical Oblivious Transfer protocols within the PIOA framework and verify correctness in the style of Canetti's Universal Composability [Can01]. We will also explore the use of PIOAs as a semantic model for the probabilistic polynomial time process calculus of Lincoln, Mitchell, Mitchell and Scedrov [LMMS98].

References

[Agg94] S. Aggarwal. Time optimal self-stabilizing spanning tree algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as Technical Report MIT/LCS/TR-632.

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computing*, pages 136–145, 2001.

[dAHJ01] L. de Alfaro, T.A. Henzinger, and R. Jhala. Compositional methods for probabilistic systems. In K.G. Larsen and M. Nielsen, editors, *Proceedings CONCUR 01,* Aalborg, Denmark, August 20-25, 2001, volume 2154 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2001.

[LMMS98] P. Lincoln, J.C. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic polytime framework for protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 112–121, 1998.

[LSS94] N.A. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings of the 13th Annual ACM Symposium on the Principles of Distributed Computing*, pages 314–323, Los Angeles, CA, August 1994.

[LSV03] N.A. Lynch, R. Segala, and F.W. Vaandrager. Compositionality for probabilistic automata. In R. Amadio and D. Lugiez, editors, *Proceedings 14th International Conference on Concurrency Theory (CONCUR 2003),* Marseille, France, volume 2761 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, September 2003.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[PSL00] A. Pogosyants, R. Segala, and N.A. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. *Distributed Computing*, 13(3):155–186, 2000.

[Seg95] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995. Available as Technical Report MIT/LCS/TR-676.

[Sto02] M.I.A. Stoelinga. An introduction to probabilistic automata. *Bulletin of the European Association for Theoretical Computer Science*, 78:176–198, October 2002.

[SV99] M.I.A. Stoelinga and F.W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proceedings 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems,* Bamberg, Germany, volume 1601 of *Lecture Notes in Computer Science*, pages 53–74. Springer-Verlag, 1999.

[SV04] A. Sokolova and E.P. de Vink. Probabilistic automata: system types, parallel composition and comparison. In C. Baier et al., editor, *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 2004.

[WSS94] S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic i/o automata. In B. Jonsson and J. Parrow, editors, *Proceedings CONCUR 94,* Uppsala, Sweden, volume 836 of *Lecture Notes in Computer Science*, pages 513–528. Springer-Verlag, 1994.

Decomposing Controllers into Non-conflicting Distributed Controllers*

Padmanabhan Krishnan

Centre for Software Assurance, Faculty of Information Technology,
Bond University, Gold Coast, Queensland 4229, Australia
pkrishna@staff.bond.edu.au

Abstract. In this article we present an application of decompositions of automata to obtain distributed controllers. The decomposition technique is derived from the classical method of partitions. This is then applied to the domain of discrete event systems. We show that it is possible to decompose a monolithic controller into smaller controllers which are non-conflicting. This is derived from the notion of decompositions via partitions. Some global state information is necessary to ensure that the joint behaviour of the component automata is identical to the original controller. The global state information required is identical to the global information present in Zielonka asynchronous automata. The joint behaviour of the component automata is shown to be non-conflicting.

Keywords: Decompositions, Asynchronous Automata, Controllers.

1 Introduction

It is well known that one can synthesise controllers for discrete event systems (DES) using von-Neumann discrete game playing techniques [RW89]. In the context of automata one is given a system description (also called a plant or environment) and a specification of desired behaviour. The synthesis process generates an automaton called the controller or supervisor. The synthesis process ensures that the joint behaviour of the controller and the plant is within the behaviours stated in the specification. The principal advantage of this synthesis process is that it is completely automatic. Furthermore, the synthesised controller is the most general controller (i.e., the controller that permits the largest possible behaviour). In this article we present an approach to synthesising distributed/parallel controllers.

The modular approach to developing (including specification, refinement, verification) systems is the most promising technique to overcoming complexity [Jon94b, dRLP98]. Compositional techniques allow one to combine smaller systems to obtain larger ones. But it imposes certain conditions to ensure that the large system satisfies the requirements. While there is knowledge concerning general compositional ideas [dRLP98], the situation for controller synthesis is not that clear. [WR88] identify a sufficient condition called *non-conflicting* under which the joint operation of two controllers

* Part of this work was supported by Siemens Research, Munich, Germany.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 511–526, 2005.
© Springer-Verlag Berlin Heidelberg 2005

is valid. The literature [Won04] indicates that modular synthesis of controllers is difficult and not always successful. [SW04] show how the state space of a single controller can be reduced but do not address the distributed case. The general synthesis problem especially related to implementation is either undecidable or NP-complete [SEM03]. [PTV01] also show how decentralised control over partial observations does not always admit finite state controllers (although the overall system may be finite state). This is because the projection on the individual observations may yield non-regular languages. Tripakis in [Tri04] shows that determining the existence of a function that determines the validity of a distributed observation is undecidable. This result is related to the undecidability of a decentralized supervisory control problem. Therefore, the next best thing is to perform a "monolithic" synthesis and then automatically decompose the controller to obtain distributed controllers. By obtaining a suitable distributed automata over different alphabets, this process can also be viewed as yielding a decentralised controller over partial observations. It is this problem that we solve in this paper.

Our solution to this problem is presented in two stages. In the first stage we treat the controller obtained from the monolithic synthesis as an independent entity (i.e., an open system). We adapt the classical decomposition technique [HS66] to split the large controller into two components along with synchronisation restrictions. The key result is that this process yields non-conflicting controllers. In the second stage we simplify the two controllers (i.e., relax the synchronisation requirements) by taking the behaviour of the plant into account. While this affects the overall behaviour of the controller, it does not alter the correctness of the controller working in conjunction with the plant. This technique illustrates how a collection of simple controllers each controlling only certain aspects of behaviour but achieving the overall control can be constructed. The strategy described in this paper can be summarised as follows.

1. Generate modular controllers from the distributed system.
2. If the generated controllers are non-conflicting then no further action is necessary.
3. Otherwise, generate a monolithic controller.
4. Decompose the monolithic controller to obtain two non-conflicting controllers. This is described in section 3 and an example is presented in section 4.
5. Simplify the two controllers using the behaviour of the plant. This is described in section 5.

In the next section we review the relevant results pertaining to decomposition of automata using partitions, relevant definitions from DES and the definition of distributed automata. We then introduce a different presentation of distributed automata. This is followed by the main results of this paper. First we show the decomposition of a single automaton into distributed automata such that the languages accepted by them are the same. Then we show that the decomposition process using partitions ensures non-conflicting behaviour. Two examples, one of which is devoted to the decomposition of controllers, are then presented. Finally, we present the extension to the basic technique which takes into account the behaviour of the plant that is being controlled.

2 Preliminaries

In this section we briefly recall the concepts and notation required to explain decompositions and controllers. The details concerning decompositions can be found in [HS66, Hol82] while the details concerning discrete event systems and controllers can be found in [RW89].

2.1 Decompositions

A finite state automaton $\mathcal{A}$ consists of a 5 tuple $(Q, \Sigma, \longrightarrow, q^0, F)$ where Q is a finite set of states, Σ a finite input alphabet, $\longrightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, $q^0 \in Q$ the initial state and $F \subseteq Q$ the set of final states. As a notational convenience $(q, a, q') \in \longrightarrow$ is often written as $q \xrightarrow{a} q'$.

We extend the transition relation to sets. That is, for $(X \subseteq Q)$ and $(a \in \Sigma)$ we let $X \xrightarrow{a} Y$ where $Y = \{q' \mid (q \in X), (q \xrightarrow{a} q')\}$. Given two automata $\mathcal{A}_1$=$(Q_1, \Sigma, \longrightarrow_1, q_1^0, F_1)$ and $\mathcal{A}_2$=$(Q_2, \Sigma, \longrightarrow_2, q_2^0, F_2)$ we write $(\mathcal{A}_1 \leq \mathcal{A}_2)$ if there is a *surjective partial* map η from Q_2 to Q_1 satisfying the following three properties: 1) $\forall\, q, q' \in Q_2$, $a \in \Sigma$: $\eta(q) \xrightarrow{a}_1 \eta(q')$ implies $q \xrightarrow{a}_2 q'$, 2) $\eta(q_2^0) = q_1^0$ and 3) $F_2 = \{q \in Q_2 \mid \eta(q) \in F_1\}$

That is, there is a *covering homomorphism* from $\mathcal{A}_2$ to $\mathcal{A}_1$. It is possible for $\mathcal{A}_2$ to have more states and transitions than $\mathcal{A}_1$. However, if $\mathcal{A}_2$ has no a move in a state q, there cannot be an a move from $\eta(q)$ or $\eta(q)$ is undefined.

Given two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ over a common input alphabet Σ, define $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ to represent the *synchronous product* (also called the meet operator in [Won04]). That is, the automaton $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is defined as $(Q_1 \times Q_2, \Sigma, \longrightarrow, q_1^0 \times q_2^0, F_1 \times F_2)$ where $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ if and only if $q_1 \xrightarrow{a}_1 q_1'$ and $q_2 \xrightarrow{a}_2 q_2'$. We say $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is a *decomposition* of $\mathcal{A}$ if and only if $\mathcal{A} \leq (\mathcal{A}_1 \parallel \mathcal{A}_2)$. A decomposition of $\mathcal{A}$ into $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ is *non-trivial* if $\mathcal{A}$ is not identical to $\mathcal{A}_1$ or $\mathcal{A}_2$.

Given $\mathcal{A}$, a partition π over Q is *admissible* if and only if for every X belonging to π and for every a in the input alphabet, there is a Y belonging to π such that $X \xrightarrow{a} Z$ and $Z \subseteq Y$.

The product of two partitions π_1 and π_2 written as $\pi_1 \cdot \pi_2$ is defined as follows. $\pi_1 \cdot \pi_2$ = $\{\, X \cap Y \mid X \in \pi_1, Y \in \pi_2, X \cap Y \neq \emptyset\}$. The finest partition $\perp_Q$ is defined to be set of all singletons of Q. The coarsest partition $\top_Q$ is the set $\{\, Q \,\}$. A partition is non-trivial if it is neither the coarsest nor the finest partition. Two partitions are *orthogonal* if their product yields the finest partition.

Proposition 1 ([Hol82]). If a given automaton $\mathcal{A}$ has two non-trivial orthogonal partitions each of which is admissible, then it has a non-trivial decomposition.

The proof of the proposition constructs $\mathcal{A}_1$ and $\mathcal{A}_2$ by considering orthogonal admissible partitions. That is, each state in the component automata is actually a subset of states of the original automaton. The construction also ensures that $\mathcal{A} \leq (\mathcal{A}_1 \parallel \mathcal{A}_2)$. In such a case, $\mathcal{A}$ is said to be *decomposed via partitions*.

2.2 Non-conflicting Controllers

Given a description of a plant (say P) (or the environment) and a desired specification (say S), the purpose of a controller (say C) is to ensure that the operation of the plant is

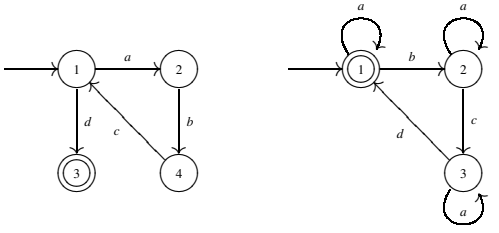


Fig. 1. Conflicting Automata

within the desired specification. That is, the behaviour of (P ‖ C) should be contained in S. There are certain restrictions on C for it to be called a controller for P. Every symbol in the alphabet of the automaton P can be classified as either controllable or uncontrollable. A controllable action can either be disabled or enabled by the controller. An uncontrollable action cannot be disabled to the controller. Hence in any global state of (P ‖ C), if P can exhibit an uncontrollable action, C must also be able to exhibit it.

This can be formally stated as follows. Given an alphabet Σ, let Σ_u denote the uncontrollable subset of Σ and Σ_c denote the controllable subset of Σ. A prefix closed language L is a *controllable language* with respect to a prefix closed language G if and only if for every $\alpha \in \Sigma^*$ belonging to L and a belonging to Σ_u, if αa belongs to G, αa belongs to L. Here G represents the trace behaviour of the plant P and L the trace behaviour of the controller C. In any given state of the computation (denoted by α), if the plant can perform an uncontrollable action (a) the controller must allow it to occur. The above definition can be extended to non-prefix closed languages. A language L is a controllable language with respect to a language G iff the prefix closure of L controllable with respect to the prefix closure of G.

The controller itself may consist of a number of parallel components, and the key idea is that they should not be conflicting one another. Given a regular language L, let $\widetilde{L}$ represent the prefix closure of L. Two languages L and K are *non-conflicting* iff $\widetilde{L} \cap \widetilde{K}$ is identical to $\widetilde{L \cap K}$. It is easy to see that $\widetilde{L \cap K} \subseteq \widetilde{L} \cap \widetilde{K}$. So, to verify non-conflicting behaviour, we only have to show inclusion in one direction. The usefulness of non-conflicting behaviour is illustrated by the following proposition.

Proposition 2 ([WR88]). *Let L_1 and L_2 be supremal (largest with respect to inclusion) controllable sub-languages of E_1 and E_2 respectively. If L_1 and L_2 are non-conflicting controllable languages , $L_1 \cap L_2$ is the supremal controllable sublanguage of $E_1 \cap E_2$.*

We now give an example to illustrate the problem of conflicting controllers or supervisors.

Consider the two automata shown in figure 1. Let L be the language accepted by the automaton shown on the left and K be the language accepted by the automaton shown on the right. Note that $L \cap K = \{abcd\}$. Consider the string *abca* which belongs to $(\widetilde{L} \cap \widetilde{K})$. This is because the string *abcabcd* belongs to L and the string *abcad* belongs to K. Hence the string *abca* belongs to the intersection of the prefix closures of the two languages. However, it cannot belong to the prefix closure of $(L \cap K)$. Such behaviour is conflicting because if the environment asks for permission to exhibit the symbol a after

exhibiting *abc*, both controllers give their permission. But the system becomes stuck as no extension of this string can lead to a final state. Hence the system enters a deadlock state. This arises as the automaton is viewed as interacting with an environment rather than as a pure acceptor. In other words, non-conflicting controllers ensure the plant does not deadlock under their joint supervision.

2.3 Asynchronous Automata

We now describe asynchronous automata [Zie87]. We assume a finite set $Loc = \{1, \ldots, n\}$ of agents. Associated at each location is an automaton with its corresponding alphabet. These are combined as follows. Let $(\Sigma_1, \Sigma_2, \ldots, \Sigma_n)$ be a distributed alphabet and $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$ be the associated automata. We will assume that the state spaces of the component automata are pairwise disjoint but could have overlapping alphabets.

Let $\Sigma = \bigcup_{i \in Loc} \Sigma_i$. For $a \in \Sigma$ let $loc(a) = \{j \mid a \in \Sigma_j\}$. We define $Q = \bigcup_{i \in Loc} Q_i$ to be the set of local states. Given $L \subseteq Loc$, we define the set of possible L-states, $Q_L = \{ q: L \longrightarrow Q \mid \forall l \in L, q(l) \in Q_l \}$. Q_L defines the global state of the automaton as viewed from locations in L. For every action $a \in \Sigma$ we write Q_a to represent $Q_{loc(a)}$. Similarly, for $q \in Q_{Loc}$, we let q_a represent the restriction of q to $loc(a)$ and q_{-a} represent the restriction of q to $Loc - loc(a)$.

A set of a-transitions (indicated by $\Rightarrow_a \subseteq Q_a \times Q_a$) is defined for every q and q' belonging to Q to be $(q, q') \in \Rightarrow_a$ implies that for every l belonging to $loc(a), q(l) \xrightarrow{a}_l q'(l)$.

These are the a moves or the moves made by the automata to exhibit a. Strictly speaking the individual transitions are not necessary, but they help in the presentation of the automata. If one is given only $\Rightarrow_a$, the local transitions can be derived. The global transition relation on Q_{Loc}, written as $q \xrightarrow{a} q'$ is defined to be $(q_a, q'_a) \in \Rightarrow_a$ and $q_{-a} = q'_{-a}$.

Assume a global initial q^0 and a set F of final states. That is, for every l, $q^0(l)$ is identical to to q^0_l. Similarly, if f belongs to F, for every l, $f(l)$ belongs to F^l. The Zielonka asynchronous automaton is given by the 5-tuple $(Q_{Loc}, \Sigma, \longrightarrow, q^0, F)$. At this stage it is relevant to note that the asynchronous automata are not the same as the taking the sychronous product of the individual automata. In asynchronous automata not all states in the product space where the an action is possible individually is necessarily enabled. As shown in [Zie87] the language $((aa \mid bb)c + (a \mid b)c)^*$ where $\mid$ indicates the shuffle cannot be accepted by a product of two automata over the alphabet $\{a, c\}$ and $\{b, c\}$. An asynchronous automaton can be defined to accept this by defining the global states where a c can be exhibited to be either after one a and one b or after two as and two bs.

We use a new presentation of asynchronous automata called *restricted product automata*. The new representation is useful in the synthesis process. The notion of local and global states is as before. Instead of having the family of transition relations $\Rightarrow_a$, we have a family of synchronisation constraints. That is, the states in which the action a is possible is explicitly defined. For each input symbol a, we assume a set $synch_a$ which is contained in Q_a such that if the cardinality of $loc(a)$ is 1, $synch_a$ is identical to Q_a. The interpretation is that an action a can be exhibited only from a permitted a-state.

Such a set is defined for each action in the alphabet. This family of sets represents the global synchronisation information. The requirement on $synch_a$ states that purely local actions do not depend on any global information and hence cannot be disabled. The global transition relation, $\longrightarrow$, can now be defined as follows.

We let $(q \xrightarrow{a} q')$ if and only if the three conditions $q_l \xrightarrow{a}_l q'_l$ for all $l \in loc(a)$, $q_{-a} = q'_{-a}$ and $q_a \in synch_a$ hold.

The first two conditions are similar to that of Zielonka automata. The last condition states that the action a can be exhibited only from a permitted state; or in other words an action is allowed in state q_a. In synchronising automata described in [Ram95], the component automata can have common states. If a common action is exhibited, the resulting states have to be identical. As an aside, the difference with synchronising automata in [Ram95] is that we require the states to "agree" before the action rather than after the action.

The following proposition makes it clear that restricted product automata are only a new presentation of asynchronous automata. That is, we are using syntactic sugar to simplify the synthesis process.

Lemma 1. *Deterministic Zielonka asynchronous automata correspond exactly to deterministic restricted product automata.*

Proof: By converting every a-transition to transitions for the component automata and by including the a-state into the set of permissible states we can translate one automaton into another. That is, $(q, q') \in \Rightarrow_a$ iff q belongs to $synch_a$. Furthermore, for every l belonging to $loc(a)$, we demand $q(l) \xrightarrow{a}_l q'(l)$ •

While the theory for asynchronous automata is presented in terms of some finite (n) components, we now restrict our attention to two component system. This is primarily because the decomposition theory yields two components. The approach presented here can be iterated to obtain any number of component systems.

3 Exact and Non-conflicting Decompositions

Let $\mathcal{A}$ be a controller synthesised from some plant description along with the specified behaviour. The task is to decompose this automata into two non-conflicting controllers. We show that decomposition via partitions followed by the generation of synchronisation conditions and the removal of unnecessary transitions yields the desired result.

Let $\mathcal{A}$ be decomposed via partitions into $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\mathcal{A} \leq (\mathcal{A}_1 \parallel \mathcal{A}_2)$. Also assume that the covering map from the states of $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ to $\mathcal{A}$ is given by η. The initial state and the set of final states in $(\mathcal{A}_1 \parallel \mathcal{A}_2)$ are precisely those which are mapped under η to the initial state or final states in the original automaton.

We now describe the technique of extracting the distributed alphabet, the component automata and the synchronising conditions. The first step is to synthesise the synchronisation restrictions and the second step is to obtain the distributed alphabet. To compute the synchronisation restrictions, for every action a, define $synch_a$ to be $\{q \in (Q_1 \times Q_2) \mid \exists q' \in Q, \eta(q) \xrightarrow{a} q'\}$. Let *synch* represent the collection the synchronisation sets. The following proposition characterises the importance of *synch*.

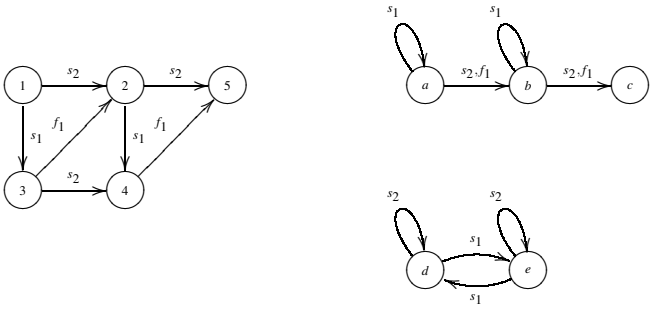


Fig. 2. Example

Lemma 2. *Let $\mathcal{A}$ be decomposed via partitions into $\mathcal{A}_1$, $\mathcal{A}_2$. Also assume that synch as specified above has been obtained.*

If the state (q_1, q_2) of the automaton ($\mathcal{A}_1 \parallel \mathcal{A}_2$) is reachable under synch, $\eta((q_1, q_2))$ is defined and is identical to the intersection of q_1 and q_2.

Proof: Initially this is true as the initial state is (q_1, q_2) where $q_0 \in (q_1 \cap q_2)$. If $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$, it is essential that $q_1 \xrightarrow{a}_1 q_1'$ and $q_2 \xrightarrow{a}_2 q_2'$. The definition of *synch* requires $q_1 \cap q_2 \xrightarrow{a} s$. But then s has to belong to q_1' and q_2'. As the partitions are orthogonal, $q_1' \cap q_2'$ is identical to $\{s\}$. Therefore, $\eta((q_1', q_2'))$ will be s. •

3.1 Distributed Alphabet

Thus far we have only generated the two automata along with the synchronisation restrictions. However, the alphabets of the two automata are identical. We now describe the procedure to drop certain symbols (along with their transitions) from the component automata thereby obtaining truly distributed automata.

The distributed alphabet is obtained by deleting all "redundant" symbols and hence the associated transitions. The idea behind removing redundant symbols is that any symbol that cannot change the local state of machine (say $\mathcal{A}_2$) or block the other component (say $\mathcal{A}_1$), can be ignored in $\mathcal{A}_2$. Furthermore, all transitions on the symbol must also be permitted by *synch*. In other words, every a a-move of the the first component is independent of the second component and is permitted by the synchronisation requirements.

More precisely, a symbol of $\mathcal{A}_2$ is *redundant* if the following two conditions hold.

1. $\mathcal{A}_2$ has only self-loops on the symbol a. That is, for every q_2 belonging to Q_2, $q_2 \xrightarrow{a}_2 q_2$.
2. $\forall q_1 \in Q_1$, ($q_1 \xrightarrow{a}_1$ implies $\forall q_2 \in Q_2$, $(q_1, q_2) \in synch_a$).

A symmetric definition for the redundant symbols of $\mathcal{A}_1$ will be assumed.

Consider the automaton and its decomposition shown in figure 2 which will illustrate the identification of a redundant symbol. Let a denote the state $\{1,3\}$, b denote the state $\{2,4\}$, c denote the state $\{5\}$, d denote the state $\{1,2,5\}$ and e denote the state $\{3,4\}$. The covering map η is given as follows:

$\eta(a,d) = 1$, $\eta(a,e) = 3$, $\eta(b,d) = 2$, $\eta(b,e) = 4$, $\eta(c,d) = 5$ and $\eta(c,e)$ is not defined.

The set $synch_{s_2}$ is $\{(a,d),(a,e),(b,d)\}$. The state (b,e) does not belong to this set as there is no s_2 transition from state 4. Hence the self-loops on s_2 in the second component cannot be removed. However, if *synch* permitted all moves, the self-loops can be removed. A more detailed case is presented in section 4.

Let $\mathcal{A}$ over Σ be decomposed via partitions into $\mathcal{A}_1$ and $\mathcal{A}_2$. Also assume that the synchronisation sets *synch* has been synthesised. Let Σ_i' be the subset of Σ obtained by eliminating redundant symbols with $\mathcal{A}_i{}'$ to be the restriction of $\mathcal{A}_i$ to Σ_i'. Given $\mathcal{A}_1{}'$ over Σ_1' and $\mathcal{A}_2{}'$ over Σ_2' the synchronisation constraints are now be restricted to only those actions that occur in both automata.

Lemma 3. *Under the above conditions, if $\Sigma_1' \cup \Sigma_2' = \Sigma$, the language accepted by $\mathcal{A}$ (say L) is identical to the language accepted by restricted product automaton derived from $\mathcal{A}_1{}'$, $\mathcal{A}_2{}'$ and synch (say M).*

Proof: We first show that $L \subseteq M$. The complete proof can be found in [Hol82]. Here we present the general idea. Let $a_0 a_1 \cdots a_n$ belong to L. Hence there is a run (a sequence of states) of $\mathcal{A}$ $q_0, q_1, \ldots q_{n+1}$ such that q_0 is the initial state and q_{n+1} belongs to F. Also $q_i \xrightarrow{a_i} q_{i+1}$. As η is a covering map, there has to exist a global state (q_1^i, q_2^i) mapped onto to q_i. Similarly a global state (q_1^{i+1}, q_2^{1+1}) has to exist with the proviso that (q_1^i, q_2^i) has an a transition to (q_1^{i+1}, q_2^{i+1}). The initiality and finality conditions follow directly from the definition.

We now show $M \subseteq L$. Let $a_0 a_1 \cdots a_n$ belong to M. Hence there is a global run (q_1^0, q_2^0), (q_1^1, q_2^1), $\ldots$,(q_1^{n+1}, q_2^{n+1}). We translate this run into a run of the original automaton. For each (q_1^i, q_2^i), there will be a state q_i such that $\eta(q_1^i, q_2^i) = q_i$. This is guaranteed by Lemma 2.

Consider the transition $(q_1^i, q_2^i) \xrightarrow{a_i} (q_1^{i+1}, q_2^{i+1})$ made by the restricted product automaton. In this case *synch* will contain (q_1^i, q_2^i). If a_i belongs to both Σ_1' and Σ_2', the transition will be permitted in the original automaton. Otherwise, without loss of generality assume that a_i does not belong to Σ_1'. In that case a_i is redundant in $\mathcal{A}_1$ which means that in the decomposed automaton $q_1^i \xrightarrow{a_i} q_1^i$. Once again this implies that the transition will be permitted in the original automaton.

In otherwords, if the global automaton had an a_i transition in state (q_1^i, q_2^i), $synch_a$ would have to contain (q_1^i, q_2^i). This is possible only if q_i had an a_i transition. •

The above proposition is valid for any $\mathcal{A}$, $\mathcal{A}_1{}'$ and $\mathcal{A}_2{}'$ with the necessary covering map and synchronisation sets. A few simplifications can be made for $\mathcal{A}_1{}'$ and $\mathcal{A}_2{}'$ that are derived by using partitions for the decomposition process, i.e., with subsets of states as the state space. This is related to proposition 2. From the construction of the decompositions, each state of $\mathcal{A}_1$ and $\mathcal{A}_2$ is a subset of the states of $\mathcal{A}$. As the partitions are orthogonal, η corresponds precisely to the intersection of the appropriate sets. Furthermore, if the intersection of two sets is empty, η on those states will be undefined. By proposition 2 it follows that such a global state will not be reachable. We reiterate that this observation would not hold without *synch*.

Furthermore, the transition relation on a given input symbol for $\mathcal{A}_1$ (similarly for $\mathcal{A}_2$) is defined to the union of the transitions on the same symbol for the original automaton.

Hence for action a which belongs to only to one automaton, the set $synch_a$ can be ignored. The details of this construction follows from the proof in [Hol82–page 80].

Sometimes it is helpful to add 'irrelevant' self loops so that a symbol can be abstracted away to obtain automata that are more loosely coupled. Towards, this the following observation will help.

Let $q_2 \in \mathcal{A}_2$ such that q_2 has no a move. Furthermore, for every $q_1 \in \mathcal{A}_1$ such that q_1 has an a move, let $q_1 \cap q_2$ be the emptyset. Then adding a self loop on a to q_2 does not change the language accepted by the automata in question. This is because the global state (q_1, q_2) is not reachable and hence the question of synchronisation does not arise. This concludes the discussion of generating synchronising automata.

We now show that the two component automata are non-conflicting. Ensuring this property is key to our approach. Towards that we assume the following (a property called *trim*) about the original automaton. First, we assume that it has no unreachable states. Second, we also assume that from every state, a final state is reachable. The conversion of an automaton to an equivalent trim automaton is standard [HU79].

Lemma 4. Let $\mathcal{A}$ be a trim deterministic finite automaton decomposed (by partitions) into $\mathcal{A}_1$ and $\mathcal{A}_2$ with *synch*. Let L_1 be the language accepted by the automaton $\mathcal{A}_1$ and L_2 be the language accepted by the automaton $\mathcal{A}_2$. The joint behaviour of $\mathcal{A}_1$ and $\mathcal{A}_2$ under *synch* is non-conflicting.

Proof: Let $\alpha \in (\widetilde{L_1} \cap \widetilde{L_2})$. As the state space of the automata accepting $\widetilde{L_i}$ is identical to Q_i, we assume that the joint behaviour of the two components on α leads to the global state (q_1, q_2). Now we have to show that $\alpha\beta$ belongs to both L_1 and L_2 for some $\beta \in \Sigma^*$.

The original automaton on reading α would have reached a state q such that $\eta(q_1, q_2)$ equals q. If (q_1, q_2) does not have any move it implies that q does not have any move. But the original automaton was trim. Hence q has to be the final state in which case (q_1, q_2) is a final state. Hence β being the empty string suffices.

Otherwise, q will accept some β towards a final state. As the languages accepted by the two automata are the same, the joint behaviour should be able to exhibit β to a final state. This shows that the two automata are non-conflicting. •

To summarise, we have shown that a decomposition based on partitions yields non-conflicting Zielonka asynchronous automata.

A prototype program to help the user construct orthogonal partitions has been written. The program written in Hugs (a variant of Haskell) [Jon94a] works on an explicit state transition representation. A program to handle symbolic representation is under construction. Given a transition system, the user can specify an initial seed partition. There are two ways of specifying this. The first is a direct enumeration of the seed partition. The second technique is by using a list of actions. This list represents part of the desired distributed alphabet for the resulting controller. This list of actions can be automatically generated using the original distributed alphabet. But the user can also provide any set of actions. A user defined set of actions is useful in the context of PLCs where inputs/outputs are explicitly specified. In this case, a seed partition is constructed such that for any action in the given list all transitions are self-loops. For example, if a belongs to the list of actions and if $q \xrightarrow{a} q'$, the states q and q' are identified. The program then computes the finest admissible partition consistent with the seed partition.

Further refinement of the partition is possible by invoking special functions. Given a list of seed partitions the program determines if the resulting admissible partitions are orthogonal. If they are orthogonal, it computes the *synch* sets and removes the redundant symbols.

4 Examples

We present two examples to illustrate the approach described earlier. The first is an asynchronous automaton that cannot be represented by the product of two automata over a given alphabet. The second is a larger example from the theory of discrete event systems.

Consider the global automaton accepting the language $((ab+ba+aabb+abab+abba+baba+bbaa+baab)d)^*$. From [Zie87] it is known that no product decomposition over the distributed alphabet ($\{b,d\}$,$\{a,d\}$) can exist. Therefore, in order to obtain a correct decomposition global information in the form of synchronisation restrictions is required. Applying the decomposition yields the automata shown in figure 3. That is the first partition is { {1,2,4}, {3,5,7}, {6,8,9}} while the second partition is { {1,3,6}, {2,5,8}, {4,7,9} }. The self loops on a (in the first component) and b (in the second component) do not have any synchronisation restrictions. So they satisfy the requirements for being redundant and hence can be removed. This results in { $\{b,d\}$, $\{a,d\}$ } as the distributed alphabet.

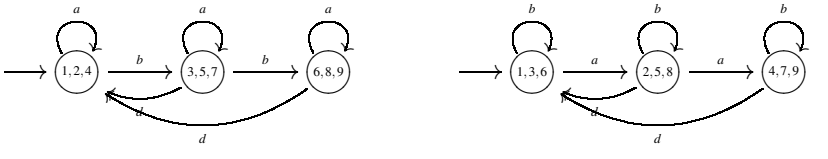
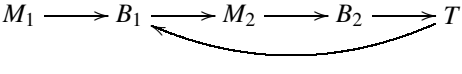


Fig. 3. The Component Automata

The covering map η can be described by the usual intersection of the partitions. From η one can compute the set $synch_d$ to contain precisely two elements, viz., ($\{3,5,7\}$, $\{2,5,8\}$) and ($\{6,8,9\}$,$\{4,7,9\}$). This is because a d transition is possible only from states 5 and 9. Another way to look at this is that the global state ($\{3,5,7\}$,$\{4,7,9\}$) cannot exhibit a d. Furthermore, this state could be reachable as their intersection is non-empty (i.e., 7). But there is no d transition in state 7 in the original automaton and hence this transition needs to be disabled. By a similar argument, the d transition in the global state ($\{6,8,9\}$,$\{2,5,8\}$) needs to be disabled as state 8 has no d transition.

Our next example is based on the controller for a transfer line containing two machines and a tester linked by buffers as shown below.



This example has been discussed in [Won04]. This example illustrates both the strength and weakness of the decomposition approach.

The machine M_1 can start (s_1) or finish (f_1). When the machine finishes it places an item in the buffer B_1. If B_1 contains any element the machine M_2 can start (s_2) and when it finishes (f_2) it places an item in B_2. The tester can pick up an item from B_2 (s_3) and either accept it (a) or reject it (r) in which case it places it in buffer B_1. The starting of the machines is controllable and the key is to avoid buffer overflow or underflow. We will assume that buffer B_1 has capacity three and buffer B_2 has capacity one.

While we present the behaviour of the controller and the decomposition using explicit state transition diagrams, symbolic synthesis techniques [BHG+93, NW94, MW98] are actually used in practice. However, it is hard to discuss the behaviour of such controller using symbolic representations such as BDDs [Bry86]. While the state transition diagrams may look complex, they illustrate the available symmetry which is actually exploited in the decomposition process. Issues related to the realisation of synthesised controllers are discussed in [Zha96].

The monolithic controller for this system can be synthesised and is partially shown in figure 4. The transitions on a and r have been omitted in the diagram. All other transitions are shown. There are 22 more transitions (for example from state 9 to state 10 on an a, from state 9 to state 15 on an r). The purpose of this figure is only to illustrate the regular structure present in the controller. Also, the controller synthesised using symbolic representations cannot be easily understood. In general the action a signifies that that an item has left the system. Hence it allows M_1 to start and hence introduce a new item in the system while an r action is as if an old item is reintroduced into the system. To begin with the controller can allow three items to be introduced into the system ($s_1, f_1, s_1, f_1, s_1, f_1$ leading to state 15). After that an s_1 can only occur after an a occurs.

By applying the theory of partitions twice, we obtain the three component automata shown in figures 5, and 6. This was achieved by the user guiding the prototype program. The first partition was achieved by requiring self-loops on the actions s_1, f_1, a and r. This is because these actions have no effect on B_2. The second partition is obtained by

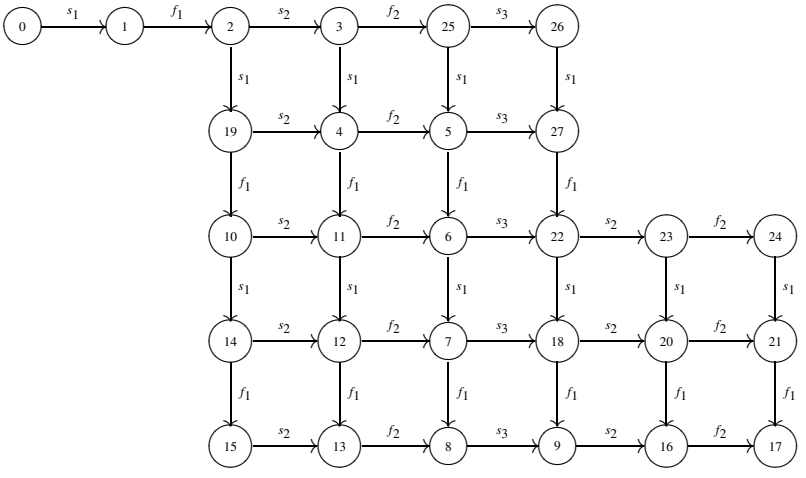


Fig. 4. Controller

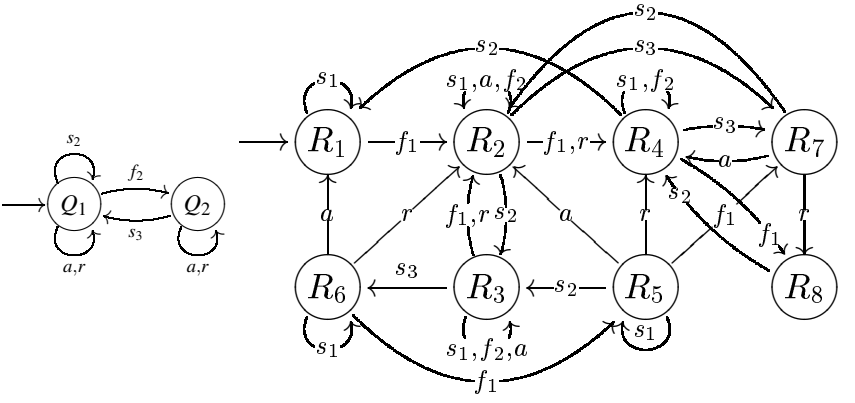


Fig. 5. *Buffer*$_2$ and *Buffer*$_1$ Underflow Control

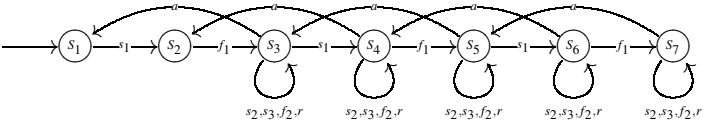


Fig. 6. *Buffer*$_1$:Overflow Control

requiring self-loops on s_1 and f_2 (trying to obtain the controller that prevents underflow of B_1) while the third partition was obtained using self-loops on the actions s_2, f_2, s_3 (trying to prevent the overflow of B_1).

The *synch* set was initially calculated. It contained elements for the actions a and r. For example, the decomposed global state (Q_1, R_2, S_3) (corresponding to state 2) will not be present in $synch_a$. This is because the decomposed global state can exhibit an a but an a action is not possible from state 2. For the sake of simplicity we do not present the complete *synch* set. By applying the simplifications discussed in the next section it can be shown that no extra global information is required. As an aside, the *synch* set can be represented as a constraint on the variables shared by the two component automata.

In the diagrams the state Q_1 refers to the set $\{0 \ldots 4,9 \ldots 16,18 \ldots 20,22,23,26,27\}$, Q_2 to the set $\{5 \ldots 8,17,21,24,25\}$, R_1 to the set $\{0,1\}$ R_2 to the set $\{2,6,7,11,12,16,17, 19\}$ R_3 to the set $\{3 \ldots 5,20,21,23,\ldots,25\}$, R_4 to the set $\{8,10,13,14\}$, R_5 to the set $\{18,22\}$, R_6 to the set $\{26,27\}$, R_7 to the set $\{9\}$, R_8 to the set $\{15\}$, S_1 to the set $\{0\}$, S_2 to the set $\{1\}$, S_3 to the set $\{2,3,25,26\}$, S_4 to the set $\{4,5,19,27\}$, S_5 to the set $\{6,10,11,22,\ldots,24\}$, S_6 to the set $\{7,12,14,18,20,21\}$ and S_7 to the set $\{8,9,13,15,\ldots,17\}$.

The automata in figure 5 ensure that buffer B_2 never overflows or underflows and that B_1 avoids underflow with respect to machine M_2. For example, an action f_1 or an action r inserts an element into B_1 after which M_2 can be started. The action s_2 removes an element from the buffer and when the count reaches 0, the machine M_2 cannot be started. This ensures that the buffer B_1 cannot underflow. However, the various transitions on the actions s_3, a and r are necessary to maintain orthogonality of the projec-

tions. We return to this automata later. The automaton in figure 6 specifies the control on buffer B_1 with respect to M_1 and more or less avoids overflow. Machine M_1 has to know the number of parts in the system before it can decide to produce more parts. This is because the tester could reject all the parts in the system and a safe behaviour requires that B_1 has at least that many free places. For the sake of readability the state associations (derived from the original automaton) are not shown in the decomposed automata.

The procedure to obtain non-conflicting controllers can be summarised as follows:

1. Synthesise the single controller.
2. Obtain two orthogonal partitions along with the synchronisation restrictions.
3. Remove redundant symbols and thereby obtain a distributed alphabet.

The current theory focuses only on partitioning into two components. The direct decomposition into multiple components needs further research.

The limitation of the partition approach is clearly demonstrated in the second automaton in figure 5 and figure 6 In the two automata shown, certain states deserve special attention. We focus our discussion on the state marked R_1. Similar arguments hold for the state marked R_6, R_7, R_8, S_1 and S_2. The decomposition process does not yield any self loop on the actions a and f_2 for the state R_1. Hence the actions s_1, a and f are not redundant. This forces a tighter coupling than desired; but as we show in the next section this tighter coupling can be loosened.

Also, for this particular problem the states R_1, R_3 and R_6 can be merged. This would then introduce self loops on various actions to the automaton. This could then be used to obtain a more loosely coupled automaton. But the merging of R_1, R_3 and R_6 does not yield orthogonal partitions. Similarly, states R_5 and R_2 can be merged and R_7 and R_4 can be merged. This can be done for this problem as state R_1, R_3 and R_6 effectively remember the buffer B_1 being empty, R_2 and R_5 indicate the buffer containing one item, R_4 and R_7 the buffer containing two items.

However, such analyses are problem specific and beyond what the theory of partitions yields. That is, the partition technique has to be augmented with problem specific analysis to further simplify the controllers. This issue is considered in the following section.

5 Including the Plant Model

So far we have not used the fact that the original automaton was a controller for a given plant. We have also not used the division of the input alphabet into controllable and uncontrollable actions. Recall that only controllable actions can be disabled. This implies that if a controller cannot exhibit an uncontrollable action in a given state, the environment cannot exhibit the action. Therefore, augmenting the controller with a transition on an uncontrollable action does not change the overall behaviour. Similarly, adding a transition on a controllable action where the plant cannot use it does not enable any new behaviours.

The general observation (for product automata) is as follows. Let q_1 be a state in the first component and q_2 be a state in the second component. If both the states have no

a transition from them adding an a transition to *only one* of them will not affect their joint behaviour. As we are synthesising the controllers, we can add extra transitions to them without enabling unspecified behaviours.

By using transitions that the plant cannot perform, the component controllers automata obtained by the partition technique can be simplified by reducing the required global information. This simplification will alter the language accepted by the controller but will not affect the joint behaviour of the plant and the controller.

We now present this more precisely. Let $\mathcal{A}_C$ represent the controller and $\mathcal{A}_P$ represent the plant. The effect of the controller on the plant can be characterised as follows. Let $cp : Q^C \longrightarrow 2^{Q^P}$ such that $q_p \in cp(q_c)$ iff the state (q_p, q_c) is reachable in $(\mathcal{A}_P \parallel \mathcal{A}_C)$. In other words, $cp(q_c)$ identifies all the states the plant could be in while the controller is in state q_c.

If for every q_p belonging to $cp(q_c)$, q_p has no a-move (a can be any action) and q_c has no a transition, consider $\mathcal{A}_C$ augmented with the transition $q_c \xrightarrow{a} q_c$. It is easy to see that the joint behaviour of the plant and the augmented automaton is identical to the joint behaviour of the plant and the original automaton.

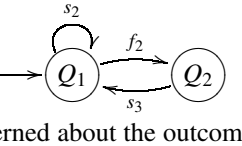
We now focus on the component controller automata. Let $\mathcal{A}_P$, $\mathcal{A}_C$ and cp be as before. Let $\mathcal{A}_{C_1}$ and $\mathcal{A}_{C_2}$ be the decomposition of $\mathcal{A}_C$ with *synch* the synchronisation set.

Consider an action a such that all a transitions in $\mathcal{A}_{C_1}$ are only self-loops. Let $N_a = \{\, q \in \mathcal{A}_{C_1} \mid q \not\xrightarrow{a}_1 \,\}$. The set N_a identifies the states that have no a transitions. Let q_1 be a state belonging to N_a. If for every state (q_2) of the automaton $\mathcal{A}_{C_2}, q_2 \xrightarrow{a}_2$ implies $cp(\eta(q_1, q_2)) \not\xrightarrow{a}_P$, augment $\mathcal{A}_{C_1}$ with the transition $q_1 \xrightarrow{a}_1 q_1$ and let *synch*′ be *synch* augmented with $\{\, (q_1, q_2) \mid q_2 \xrightarrow{a}_2 \,\}$. Call this new automaton $\mathcal{A}'_{C_1}$. By a symmetric process obtain $\mathcal{A}'_{C_2}$ from $\mathcal{A}_{C_2}$.

Under the above assumptions the following property is valid. The proof of the above proposition is obvious and follows from the definition of the product of automata.

Lemma 5. *The joint behaviour of $\mathcal{A}'_{C_1}$ and $\mathcal{A}'_{C_2}$ under synch′ with $\mathcal{A}_P$ is identical to the joint behaviour of $\mathcal{A}_{C_1}$ and $\mathcal{A}_{C_2}$ under synch with $\mathcal{A}_P$.*

We conclude this section with a discussion of an example from the previous section. The first automaton shown in figure 5 (the controller for the second buffer) has self-loops on the actions a and r which are uncontrollable actions. By considering the other decomposed automata $synch_a$ and $synch_r$ can be augmented with the extra states such that actions a and r become redundant. Hence the controller for that buffer has only s_2, f_2 and s_3 in its alphabet. The new controller is



It shows clearly that the controller is unconcerned about the outcome of the testing process as this affects only the first buffer.

Consider the automaton shown in figure 6. When the automaton is in state S_1 or S_2 (the state of the other component automata being immaterial) the plant cannot exhibit

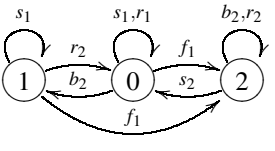
an f_2 or an r. This is because both buffers are empty. Hence one can add self loops on f_2 and r without changing the joint behaviour. Now by applying the definition of redundancy, the symbols f_2 and r can be removed from the automaton; thus obtaining a simpler automaton. This is because the controller has to be prepared for an item to be rejected (an uncontrollable action) and hence reserves a slot for it. However, if the item is accepted the slot can be released.

Note that self-loops for the symbols s_2 and s_3 cannot be added as the plant can indeed perform these actions and the controller explicitly disables them. Although one could add a-loops to the state S_1 or S_2, these are not useful as the other states do not have self-loops on a.

The simplifications can be summarised as follows. In the first case we only augment the *synch* sets while in the second case we add extra transitions as well as augment the *synch* sets. In both cases the desired outcome is identical, viz., to make a particular action redundant. As these changes are purely on the structure of the automata they can be fully automated.

6 Conclusion and Future Work

We have presented a framework in which a single controller can be decomposed into non-conflicting controllers. Although we have taken a distributed system, constructed a monolithic system and then synthesised a controller, we have suggested in the introduction that this be used only when the controllers generated from the distributed system are conflicting.

We now present an example to show the limitations for the partition based technique. Consider the automaton 

(with 0 as the start state). For the first automaton the partition $\{\{0,1\},\{2\}\}$ suffices. However, equating states 0 and 2 or 1 and 2 results in the need to equate all three states. Hence two orthogonal partitions are not possible.

A method based on set systems [HS66] (instead of partitions arbitrary subsets are used to obtain the components) could be developed. For example, the set system $\{\{0,2\}, \{0,1\}\}$ along with the partition $\{\{0,1\},\{2\}\}$ yields a decomposition. But the drawback of this approach is that one may need to consider non-deterministic systems which then have to be determinised. At this point it is not clear if the method based on set systems is as clean as the one based on partitions for deterministic systems.

Acknowledgments. The author thanks the synthesis and modular verification group at Siemens research for their support. Special thanks to Burkhard Stubert, Klaus Winkelmann and Jorge Cuellar for their detailed comments. One of the anonymous referee deserves special thanks for providing insightful comments on the original submission.

References

[BHG+93] S. Balemi, G. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. Franklin. Supervisory Control of a Rapid Thermal Multiprocessor. *IEEE Trans. on Automatic Control*, 38:1040–1059, 1993.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C35(8):677–691, 1986.

[dRLP98] W-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Different: COMPOS 1997*, volume LNCS 1536, Bad Malente, Germany, 1998.

[Hol82] W. M. L. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1982.

[HS66] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, 1966.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[Jon94a] M. Jones. *An introduction to HUGS, Version 1.01*. University of Nottingham, 1994.

[Jon94b] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[MW98] H. Melcher and K. Winkelmann. Controller synthesis for the production cell case study. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 24–33, New York, March 4–5 1998. ACM Press.

[NW94] K. Noekel and K. Winkelmann. CSL. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems: Case Study Production Cell*, volume LNCS 891, pages 55–74. Springer Verlag, 1994.

[PTV01] A. Puri, S. Tripakis, and P. Varaiya. Problems and examples of decentralized observation and control for discrete event systems. In *Symposium on the Supervisory Control of Discrete Event Systems*, 2001.

[Ram95] R. Ramanujam. A local presentation of synchronizing systems. In *Structures in Concurrency Theory*, Springer Workshops in Computing, pages 91–118. Springer Verlag, 1995.

[RW89] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[SEM03] A. Stefanescu, J. Esparza, and A. Muscholl. Synthesis of distributed algorithms using asynchronous automata. In *Proc. of CONCUR'03*, LNCS 2761, pages 27–41, 2003.

[SW04] R. Su and W. M. Wonham. Supervisory Reduction for Discrete-Event Systems. *Discrete Event Dynamic Systems*, 14(1):31–53, January 2004.

[Tri04] S. Tripakis. Undecidable problems of decentralized observation and control on regular languages. *Information Processing Letters*, 90(1):21–28, 2004.

[Won04] W. M. Wonham. Supervisory control of discrete-event systems (updated notes). Technical report, Systems Control Group, University of Toronto, Canada, 2004.

[WR88] W. M. Wonham and P. J. G. Ramadge. Modular Supervisory Control of Discrete Event Systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988.

[Zha96] Y. Zhang. Software for State-Event Observation Theory and its Application to Supervisory Control. Master's thesis, Department of Control Engineering, University of Toronto, Canada, 1996.

[Zie87] W. Zielonka. Notes on Finite Asynchronous Automata. *RAIRO: Theoretical Informatics and Applications*, 21(2):101–135, 1987.

Reasoning About Co–Büchi Tree Automata*

Salvatore La Torre[1] and Aniello Murano[1,2]

[1] Università degli Studi di Salerno
[2] Hebrew University
{slatorre, murano}@unisa.it

Abstract. We consider co–Büchi tree automata along with both alternating and generalized paradigms, as a characterization of the class of languages whose complement is accepted by generalized Büchi tree automata. We first prove that for alternating generalized co–Büchi tree automata the simulation theorem does not hold and the generalized acceptance does not add to the expressive power of the model. Then, we show that the emptiness problem for this class is EXPTIME-complete. For the class of languages whose complement is accepted by deterministic generalized Büchi tree automata, we get better complexity bounds: we give a characterization of this class in terms of generalized co–Büchi tree automata that yields an algorithm for checking the emptiness that takes time linear in the product of the number of states and the number of sets in the acceptance condition. Finally, we compare the classes of languages whose complement is respectively accepted by deterministic and nondeterministic Büchi tree automata with the main classes studied in the literature.

1 Introduction

Since its early days the theory of automata had an astonishing impact in computer science. Several models of automata have been extensively studied and applied to many fields. In the sixties, with their pioneering work, Büchi [Büc62], McNaughton [McN66], and Rabin [Rab69, Rab70] enriched this theory by introducing finite automata on infinite objects. Such automata turn out to be very useful for those areas of computer science where nonterminating computations are studied. They give a unifying paradigm to specify, verify, and synthesize reactive systems [Kur94, VW86, VW94]. A system specification can be translated into an automaton, and thus, questions about systems and their specifications are reduced to decision problems in the automata theory. More precisely, given a system S and its specification φ, we can design an automaton A_S representing the system and an automaton $A_{\neg\varphi}$ accepting all computations that violate the specification. Thus, we can check the correctness of S with respect to φ by checking the emptiness of $A_S \times A_{\neg\varphi}$.

* This research was supported by the MIUR grants 60% 2003-2004.

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 527–542, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In system modelling, computations can be seen as finite or infinite sequences of system states. To model nondeterminism, it is useful and natural to arrange computations in trees. It is worth noticing that some concurrent programs, such as operating systems, communication protocols, and air-traffic control systems, are intrinsically nondeterministic and nonterminating. Moreover, nondeterminism is successfully used to obtain models of concurrent programs in general (nondeterministic interleaving of atomic processes).

Automata on infinite objects recognize objects according to an acceptance criteria. In the literature, several acceptance criteria have been fruitfully investigated on words and trees: recall Büchi, co–Büchi, Muller, Rabin, Streett, and parity acceptance conditions (for more on these models see [GTW02]). For example, in the Büchi condition a subset of the automaton states is defined as accepting and a word/tree is accepted if and only if there exists a run such that "at least an accepting state repeats infinitely often". The co–Büchi condition expresses the dual condition, that is, it requires that "all states that are not accepting repeats finitely often" or equivalently "all the states that repeat infinitely often are accepting". Büchi and co–Büchi conditions can be generalized in the sense that more than one subset of states can be defined as accepting. Thus, an infinite word/tree is accepted by a *generalized* Büchi automaton if and only if *for each* accepting set there is at least a state that repeats infinitely often. Consistently, an infinite word/tree is accepted by a *generalized* co–Büchi automaton if and only if *there is* an accepting set that contains all the states that repeat infinitely often.

Generalized Büchi and co–Büchi acceptance conditions lead to automata with fewer states and simpler underlying structure than the corresponding standard conditions. For example, the traditional translation of an LTL formula φ to a Büchi word automaton results in an automaton with $2^{\mathcal{O}(|\varphi|\times|\varphi|)}$ states [VW94], while using generalized Büchi automata we only need $2^{\mathcal{O}(|\varphi|)}$ states [GPVW95]. Generalized conditions have become popular in system verification and now are fruitfully used in several applications [Kur94]. The generalized co-Büchi condition was first introduced and studied on infinite words in [Lan69]. Its extension to infinite trees has been investigated in [LMN02].

The kind of acceptance condition influences both the closure properties and the complexity of the decision algorithms. For generalized Büchi and generalized co-Büchi tree automata non-emptiness is decidable in polynomial time [VW86, LMN02], for Rabin tree automata it is known to be NP-complete [Rab69]. On the other hand, generalized Büchi and generalized co-Büchi tree automata are not closed under language complementation, while Rabin and Muller tree automata are [Tho90].

Alternating tree automata along with the co–Büchi paradigm characterize the complement of languages nondeterministically accepted by Büchi tree automata [MS87]. Here, we consider these automata along with the generalized paradigm, namely, we consider alternating generalized co–Büchi tree automata (AGCTA), as a direct characterization of the class of languages whose complement is accepted by generalized Büchi tree automata (co–GBTA). In [MS87], it

is shown that while alternation does not give more expressive power to Muller, parity, Rabin, Street and Büchi paradigms (*simulation theorem*), it allows us to get more succinct automata. For example, translating an alternating Büchi tree automaton to a Büchi tree automaton might involve an exponential blow-up [MS95]. Once the simulation theorem holds, emptiness for alternating automata can be checked in an easy, and often efficient, way via translation to the corresponding nondeterministic model. For example, for an alternating Büchi tree automaton, we can construct a language equivalent Büchi tree automaton (which involves an exponential blow-up) and thus we can check for emptiness the starting automaton in exponential time which matches the known lower bound for the computational complexity of this problem.

Here, we prove that unfortunately the simulation theorem does not hold for AGCTA. In fact, we observe that generalized co–Büchi tree automata are not sufficiently powerful to characterize co–GBTA. We also prove that AGCTA and alternating co–Büchi tree automata (ACTA) are polynomially equivalent, that is, there exists a polynomial translation from an AGCTA to a language equivalent ACTA and viceversa. We observe that, when the generalized and the corresponding non-generalized paradigms are language equivalent, the generalized one is still of interest since it can lead to more succinct automata with evident benefits in designing efficient algorithms. As an example, we recall that nondeterministic generalized Büchi word automata and nondeterministic Büchi word automata are polynomially equivalent [Cho74]. However, computing the complement of a nondeterministic generalized Büchi automaton without constructing first the language equivalent nondeterministic Büchi automaton, may result in an automaton that is more succinct by an exponential factor [KV04].

Using the equivalence between AGCTA and ACTA, it follows that an AGCTA A can be translated to a parity tree automaton with two parity sets whose size is polynomial in the size of A. Thus, the emptiness problem for alternating generalized co–Büchi tree automata can be decided in exponential time. This result is also complete, since we can reduce the emptiness problem for weak alternating Büchi tree automata that is known to be EXPTIME–hard [KVW00].

To characterize the class of languages whose complement is accepted by generalized deterministic Büchi tree automata (co–DGBTA) we use the generalized co–Büchi paradigm along with the request that at least one path of an accepting run must be successful ($\exists$*–acceptance*). This kind of "existential" acceptance differs from the usual request for tree automata that all paths need to be successful in order to accept. With respect to the emptiness problem, this "existential" acceptance condition is equivalent to consider the tree automaton as a word automaton: each transition is split into several transitions (one for each state successor). Thus, given a DGBTA A with n states and k accepting sets, we can construct a Büchi word automaton B with $\mathcal{O}(nk)$ states such that the language accepted by B is empty if and only if the complement of the language accepted by A is empty. Using the fact that for Büchi word automata the emptiness problem is decidable in linear time [EL85], checking the emptiness for co–DGBTA can be decided in quadratic time. We recall that an elegant characterization of

co-DGBTA can be obtained via weak alternating Büchi tree automata. Unfortunately, this characterization gives an exponential-time algorithm to solving the emptiness problem for co–DGBTA, since the emptiness problem for weak alternating Büchi tree automata is EXPTIME–complete [KVW00].

The rest of the paper is organized as follows. In Section 2, we give some basic definitions and recall some results of the theory of finite automata on infinite trees. In Section 3, we recall the concept of alternation and the main properties of alternating tree automata. In Section 4, we consider alternation along with the generalized co–Büchi paradigm and compare the corresponding class of accepted languages with the main classes of languages considered in the literature. In Section 5, we deal with the class of languages whose complement is accepted by deterministic Büchi tree automata. Finally, we conclude with few remarks in Section 6.

2 Preliminaries

Let Σ be an alphabet, k be a positive integer, and $\text{DOM} = \{0, 1, \ldots, k-1\}^*$. We define an infinite k-ary Σ-tree t as a map $t : \text{DOM} \to \Sigma$. The elements in DOM are the nodes of the tree, the empty word ε corresponds to the root and for each $w \in Dom$, wi is its i-child for $i \in \{0, 1, \ldots, k-1\}$. In the following, unless differently stated, an infinite k-ary Σ-tree will be referred to simply as a tree. Let u, $v \in \text{DOM}$, we say that u *precedes* v, denoted as $u < v$, if there exists an x such that $v = ux$. Let $\pi \subseteq \text{DOM}$, π is a *path* of a tree t if it is a maximal subset of DOM linearly ordered by $<$. If π is a path of a tree t, then t/π denotes the restriction of the function t to the set π. We say that a symbol $a \in \Sigma$ occurs infinitely often in t/π if there exists an infinite number of nodes $u \in \pi$ such that $t(u) = a$. The set of symbols that occur infinitely often in t/π is denoted by $Inf(t/\pi)$.

Given a tree t and a node $u \in \text{DOM}$, we define the *subtree* of t rooted at u as the tree t_u such that $t_u(v) = t(uv)$, for $uv \in \text{DOM}$. Let Σ be a finite alphabet, we denote by T_Σ^ω the set of Σ-valued trees. A language is a subset of T_Σ^ω. Given a language $L \subseteq T_\Sigma^\omega$ we denote with $\overline{L}$ the complement of L, that is, $\overline{L} = T_\Sigma^\omega \backslash L$. In the following, we deal exclusively with binary trees ($\text{DOM} = \{0, 1\}^*$). All the results we obtain also hold for k-ary trees, where $k \geq 2$. According to the introduced notation, we use t_0 and t_1 to denote, respectively, the subtrees of t rooted respectively at 0 and 1 (the two children of the root). Moreover, with π_0 we denote the path $\{0\}^*$.

A *finite automaton on infinite trees* (TA) is a tuple $A = \langle \Sigma, Q, Q_0, \delta, F \rangle$ where Σ is the input alphabet, $Q \neq \emptyset$ is a finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $\delta \subseteq Q \times \Sigma \times Q \times Q$ is the transition relation, and F specifies the acceptance condition (a condition that defines a subset of Q^ω, we define several types of acceptance conditions below). Intuitively, each transition suggests a nondeterministic choice for the next configuration of the automaton. If $|Q_0| = 1$ and δ is a total function $\delta : Q \times \Sigma \to Q \times Q$, then A is a *deterministic* automaton (DTA). Given an input tree t, a *run* r of A on t is a Q–tree such that $r(\varepsilon) \in Q_0$,

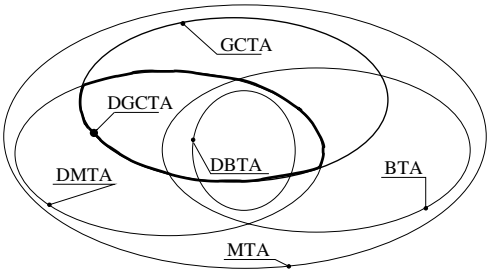


Fig. 1. Relationships between (D)BTA, (D)MTA and (D)GCTA

and $(r(u), t(u), r(u0), r(u1)) \in \delta$, for all $u \in \text{DOM}$. With $Run_A(t)$ we denote the set of runs of a TA A on a tree t. Clearly, if A is deterministic then $|Run_A(t)| = 1$. Different classes of languages are obtained defining different notions of *successful* run. A tree t is accepted by a TA A if there exists a successful run r of A on t, that is, r satisfies the acceptance condition on all the paths of t. We consider here the following conditions:

- a run r satisfies a *generalized Büchi* condition $F = \{F_1, \ldots, F_k\} \subseteq 2^Q$ iff for each path π of r and for each set $F_i \in F$, $Inf(r/\pi) \cap F_i \neq \emptyset$;
- a run r satisfies a *generalized co–Büchi* condition $F = \{F_1, \ldots, F_k\} \subseteq 2^Q$ iff for each path π of r there is a set $F_i \in F$ such that $Inf(r/\pi) \subseteq F_i$;
- a run r satisfies a *Muller* condition $F = \{F_1, \ldots, F_k\} \subseteq 2^Q$ iff for each path π of r, $Inf(r/\pi) \in F$;
- a run r satisfies a *Rabin* condition $F = \{(B_1, G_1), \ldots, (B_k, G_k)\} \subseteq 2^{Q \times Q}$ iff for each path π of r, there is a pair $(B_i, G_i) \in F$ such that $Inf(r/\pi) \cap B_i = \emptyset$ and $Inf(r/\pi) \cap G_i \neq \emptyset$;
- given a partition $F = \{F_1, F_2, \ldots, F_{2k}\}$, $k \geq 1$, of the set of states, a run r satisfies the *parity* condition F iff for each path π of r the minimal index i for which $Inf(r/\pi) \cap F_i \neq \emptyset$ is even.

In the following, we refer to the number k appearing in the acceptance condition as the *index* of the corresponding automaton. Recall that Büchi and co–Büchi conditions are defined as the corresponding generalized conditions defined above with index 1. With $L(A)$ we denote the language accepted by a TA A, that is, the set of accepted trees.

To denote the different types of tree automata, we will use acronyms of the form DXTA and XTA, where X is one of B, GB, C, GC, M, R, P. The letter D stands for deterministic and the X is used to denote the kind of acceptance condition: Büchi (B), generalized Büchi (GB), co-Büchi (C), generalized Co–Büchi (GC), Muller (M), Rabin (R) and parity (P). For example, deterministic co–Büchi tree automata are denoted by DCTA. We also use the same acronyms to denote the corresponding class of accepted languages.

Figure 1 recalls the known relationships between all the considered classes of tree languages (automata) [Rab70, LMN02, Tho90, GTW02]. Since Rabin and parity conditions are equivalent to the Muller condition, the classes of languages

accepted by the corresponding tree automata coincide in both the deterministic and nondeterministic paradigms. Thus, when we compare the class of languages we only refer to MTA and DMTA, and the obtained results clearly apply to the other classes. Analogously, since the class of languages (D)GBTA and (D)BTA coincide, in the language comparisons we only refer to (D)BTA.

The following theorem summarizes the closure properties of the above classes of automata and languages [Tho90, LMN02].

Theorem 1

- *DMTA, DGBTA, and DGCTA are closed under intersection, but they are not closed under union and complementation.*
- *GBTA and GCTA are closed under intersection and union, but they are not closed under complementation.*
- *MTA is closed under intersection, union, and complementation.*

In the following theorem, we recall some known results on the decision problems for Büchi, generalized co–Büchi, Rabin, and parity tree automata.

Theorem 2

- *The emptiness problem for BTA is decidable [Rab70], and is* LogSpace*-complete for* Ptime*[VW86].*
- *The emptiness problem for GCTA is decidable and is in* Ptime *[LMN02].*
- *The emptiness problem for PTA is UP ∩ co–UP [Jur98]*[1]*.*
- *The non-emptiness problem for RTA is* Np*-complete [Rab69, EJ88].*

3 Alternating Tree Automata

Alternating automata generalize the notion of nondeterminism by allowing several successor states along the same branch of the tree [MS87]. Muller and Schupp were the first to apply to tree automata the concept of alternation, introduced by Chandra, Kozen, and Stockmeyer [CKS81]. Here we briefly recall the basic definitions and refer to [MS95] for more details.

An *alternating tree automaton* is a TA with the transition relation defined as a function $\delta : Q \times \Sigma \to \mathcal{B}^+(K \times Q)$, where K is the set of *directions* in the tree ($K = \{0, 1\}$, for binary trees) and $\mathcal{B}^+(K \times Q)$ is the set of all positive boolean combinations of pairs (d, q), where d is a direction and q is a state.

As an example, $\delta(q, a) = ((0, q_0) \vee (1, q_1)) \wedge (1, q_0)$ means that the automaton has two nondeterministic choices: one copy of the automaton proceeds to the 0-child of the current node entering state q_0 and another copy proceeds to the 1-child also entering state q_0; or two copies proceed to the 1-child entering respectively states q_1 and q_0. Hence, $\vee$ and $\wedge$ in $\delta(q, a)$ represent, respectively, choice and concurrency.

[1] The class UP is a subset of NP, where each word accepted by the Turing machine has a unique accepting run.

A run of an alternating automaton on a binary tree t is a $(\{0,1\}^* \times Q)$–labeled (possibly non binary) tree such that the root is labelled (ε, q_0) and labels of each node and its successors must satisfy the transition relation δ. For example, if $t(\varepsilon) = a$ and $\delta(q_0, a) = ((0, q_1) \vee (0, q_2)) \wedge ((0, q_3) \vee (1, q_2))$, then, a run r on t at level 1 must include a node labeled $(0, q_1)$ or a node labeled $(0, q_2)$, and must include a node labeled $(0, q_3)$ or a node labelled $(1, q_2)$.

As for standard tree automata, we can couple different acceptance conditions to an alternating tree automaton, defining different classes of languages and automata. To denote alternating automata, we use a prefix "A" to the acronyms used so far. For example, we use ABTA to denote alternating Büchi tree automata, as well as the class of languages accepted by these automata.

In [Cho74], it is shown that GBTA and BTA are polynomially equivalent. In the next lemma, we extend this result to the alternating paradigm. That is, given an AGBTA with m states and index k, we can build a language equivalent ABTA with $\mathcal{O}(m(k+1))$ states.

Lemma 1. *Given an AGBTA A, there exists an ABTA A' accepting $L(A)$ and whose size is polynomial in the size of A.*

Proof. Let $A = \langle Q, \Sigma, \delta, Q_0, \{F_1, \ldots F_k\}\rangle$ be an AGBTA. Consider $A' = \langle Q \times \{0, \ldots k\}, \Sigma, \delta', Q_0 \times \{0\}, Q \times \{k\}\rangle$ as an ABTA, such that, for each formula $\delta(q, \sigma)$ in A, the automaton A' contains a formula $\delta'(q, i, \sigma)$ obtained from $\delta(q, \sigma)$ by coupling each pair (q', d) in $\delta(q, \sigma)$ with a value j as follows: *(i)* j=0 if $i = k$, *(ii)* $j = i + 1$ if $q \in F_j$, or *(iii)* $j = i$ otherwise. Thus, A' enters an accepting state if at least one state for all accepting sets from A has been visited infinitely often. Thus, $L(A') = L(A)$ and the size of A' is polynomial in the size of A. □

In [MS87], Muller and Schupp introduced *weak alternating Büchi tree automata* (WABTA) as a special case for ABTA. In a WABTA, we have a Büchi acceptance condition $F \subseteq Q$ and there exists a partition of Q into disjoint sets, $Q_1, \ldots, Q_m$, such that for each set Q_i, either $Q_i \subseteq F$, in which case Q_i is an *accepting* set, or $Q_i \cap F = \emptyset$, in which case Q_i is a *rejecting* set. In addition, there exists a partial order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ for which $\delta(q, \sigma, q', q'')$ or $\delta(q, \sigma, q'', q')$ occurs, we have $Q_j \leq Q_i$. Thus, transitions from a state in Q_i lead to states in either the same set Q_i or in a lower one. It follows that every infinite path of a run of a WABTA ultimately gets "trapped" within some Q_i . The path then satisfies the acceptance condition if and only if Q_i is an accepting set.

The main properties about weak alternating Büchi, alternating Büchi, and alternating parity tree automata are summarized in the following theorem. We recall that an alternating automaton is deterministic if and only if the transition relation δ does not use $\vee$ [MS95].

Theorem 3

- *Given an A(D)BTA (resp., an A(D)PTA) A, there exists a (D)BTA (resp., a (D)PTA) accepting $L(A)$, whose size is exponential in the size of A [MS87].*

– *The emptiness problem for(W)ABTA is* EXPTIME*–complete[KVW00, MS87].*
– *The emptiness problem for APTA is in* EXPTIME *[EJ91, Wil01].*

Directly from Lemma 1 and Theorem 3 we also get the following result.

Corollary 1. *The emptiness problem for AGBTA is decidable in exponential time.*

As discussed in [MS87], an advantage of using alternation is that one can complement an alternating automaton by dualizing its transition function and acceptance condition. Formally, given a transition function δ, let $\widetilde{\delta}$ denote the dual function of δ. That is, for every $\varphi \in \delta$, we have $\widetilde{\varphi}$ in $\widetilde{\delta}$, where $\widetilde{\varphi}$ is obtained by φ switching $\vee$ and $\wedge$ and by switching *true* and *false*. The dual of an acceptance condition F, denoted as $\widetilde{F}$, is a condition that holds exactly on all the runs on which F does not hold. In particular, by denoting with $\widetilde{A} = \langle Q, \Sigma, \widetilde{\delta}, Q_0, \widetilde{F}\rangle$ the dual automaton of an automaton $A = \langle Q, \Sigma, \delta, Q_0, F\rangle$, the following holds.

Theorem 4. [MS87] For an ABTA A, the ACTA $\widetilde{A}$ accepts $\overline{L(A)}$, and viceversa.

4 Alternating Generalized Co–Büchi Tree Automata

In this section, we deal with alternating tree automata along with the generalized co–Büchi paradigm (AGCTA, for short). The definition of duality given in the previous section, along with the result shown in Theorem 4 makes this class a suitable choice for a direct characterization of the class of languages whose complement is in AGBTA, as pointed out in the following.

Corollary 2. *Given an AGBTA A, its dual $\widetilde{A}$ is an AGCTA accepting $\overline{L(A)}$, and viceversa, given an AGCTA A, its dual $\widetilde{A}$ is an AGBTA accepting $\overline{L(A)}$.*

Since A(G)CTA accepts the class of languages whose complement is accepted by (G)BTA, in the following, we also denote the class of languages accepted by A(G)CTA as co–(G)BTA. Lemma 2 shows that the class of languages accepted by AGCTA is polynomially equivalent to that accepted by ACTA.

Lemma 2. *Given an AGCTA A, there exists an ACTA A' accepting $L(A)$ and whose size is polynomial in the size of A.*

Proof. Let A be an AGCTA, from Corollary 2, it follows that there exists an AGBTA B such that $\overline{L(B)} = L(A)$. From Lemma 1, it is possible to build an ABTA B' whose size is polynomial in the size of B such that $\overline{L(B)} = L(B')$. From Theorem 4, there exists an ACTA A' dual to B' such that $\overline{L(B')} = L(A')$. Thus, $L(A') = \overline{L(B')} = \overline{L(B)} = L(A)$ and the size of A' is polynomial in the size of A, from the definition of duality. □

The above lemma is useful to show the following result.

Theorem 5. *The emptiness problem for AGCTA is* EXPTIME*–complete.*

Languages	Ranking
$L_1 = \{t \in T_\Sigma^\omega \mid \exists\ \pi$, either $a \notin Inf(t/\pi)$ or $b \notin Inf(t/\pi)\}$	(GCTA $\cap$ BTA) $\setminus$ DMTA
$\overline{L_1} = \{t \in T_\Sigma^\omega \mid \forall\ \pi,\ a \in Inf(t/\pi)$ and $b \in Inf(t/\pi)\}$	DBTA $\setminus$ GCTA
$L_2 = \{t \in T_\Sigma^\omega \mid \forall\ \pi$, either $a \notin Inf(t/\pi)$ or $b \notin Inf(t/\pi)\}$	DGCTA $\setminus$ BTA
$\overline{L_2} = \{t \in T_\Sigma^\omega \mid \exists\ \pi,\ a \in Inf(t/\pi)$ and $b \in Inf(t/\pi)\}$	BTA $\setminus$ GCTA
$L_3 = \{t \in T_\Sigma^\omega \mid a \notin Inf(t/\pi_0)\}$	(BTA $\cap$ GCTA $\cap$ DMTA) $\setminus$ DBTA
$\overline{L_3} = \{t \in T_\Sigma^\omega \mid a \in Inf(t/\pi_0)\}$	DBTA

Fig. 2. Some tree languages and their classification [LMN02]

Proof. We first show that given an AGCTA A there exists an APTA B accepting $L(A)$ and whose size is polynomial in the size of A. From Lemma 2, we first translate A into an ACTA A', whose size is polynomial in the size of A. Let $A' = \langle Q, \Sigma, \delta, Q_0, \{F\}\rangle$ be the obtained ACTA. An APTA accepting $L(A')$ is the automaton $B = \langle Q, \Sigma, \delta, Q_0, \{Q \setminus F, F\}\rangle$. Thus, for the emptiness problem for AGCTA membership to EXPTIME follows from the fact that the size of B is linear in the size of A', the size of A' is polynomial in the size of A, and the emptiness problem for APTA is in EXPTIME (see Theorem 3).

For the lower bound, we observe that each weak alternating Büchi tree automaton A can be translated into a language equivalent alternating co–Büchi tree automaton by simply interpreting its acceptance set as a co–Büchi condition. In fact, by the structure of the transition relation of a WABTA and the property that each set of the partition of its states is either contained into or disjoint from the acceptance set, we get that, along the paths of an accepting run of A, the states that repeat infinitely often are only states within the acceptance set. Since the emptiness problem for weak alternating Büchi tree automata is EXPTIME–hard [KVW00], we get that the emptiness problem for ACTA, and thus AGCTA, is EXPTIME-hard. □

Directly from the above result, we also obtain the following.

Corollary 3. *The universality problem for GBTA is* EXPTIME*-complete.*

Proof. The upper bound follows from Corollary 2 and from the fact that the emptiness problem for AGCTA is decidable in exponential time (Theorem 5). For the lower bound, we observe that the universality problem for automata on finite trees is EXPTIME–hard [Sei90]. □

We now study the relationships between co–BTA and the classes of languages we have introduced in the previous sections. For this purpose, in Figure 2 we list some languages along with their ranking relatively to the classification illustrated in Figure 1. For more details see [LMN02]. For all these languages we assume that $\Sigma = \{a, b\}$. Since the classes of languages co–GBTA, co–BTA, AGCTA, and ACTA coincide, in the following we only use co–BTA to refer to this class.

Lemma 3. *co–BTA* $\not\subseteq$ *GCTA.*

Proof. This result can be shown using the languages L_1 and $\overline{L_1}$. From the table in Figure 2, we know that $L_1 \in$ BTA, thus $\overline{L_1} \in$ co–BTA, while $\overline{L_1} \notin$ GCTA. □

Directly from the non–equivalence between generalized co–Büchi and alternating generalized co–Büchi paradigms shown in Lemma 3, we get the following important result for the latter.

Theorem 6. *The simulation theorem does not hold for alternating generalized co–Büchi tree automata.*

The following lemma states the results of all the remaining comparisons involving the class co–BTA. The complete picture of the relationships among all discussed classes is given in Figure 3.

Lemma 4

1. *GCTA* $\subset$ *co–BTA.*
2. *DMTA* $\subset$ *co–BTA.*
3. *BTA and co–BTA are not comparable.*
4. *co–BTA* $\not\subseteq$ *(GCTA* $\cup$ *BTA* $\cup$ *DMTA).*
5. *(BTA* $\setminus$ *(GCTA* $\cup$ *DMTA))* $\cap$ *co–BTA* $\neq \emptyset$.
6. *BTA* $\cup$ *co–BTA* $\subset$ *MTA.*

Proof. To prove that GCTA $\subseteq$ co–BTA, we recall that any GCTA A is also an AGCTA, and thus, from Theorem 4, $L(A)$ is the complement of a language accepted by the AGBTA B dual of A. Thus, the result follows from the fact that the generalized Büchi paradigm is equivalent to Büchi. Strict containment is a consequence of Lemma 3. Thus part 1 holds.

To prove that DMTA $\subseteq$ co–BTA, we first observe that on a tree t, a deterministic tree automaton can only check that the acceptance condition holds on a fixed path of t, or on all paths of t. Thus, given a DMTA M, $\overline{L(M)}$ consists of all trees such that the acceptance condition of M does not hold on a fixed path or on a nondeterministicly selected path of t. Since a nondeterministic selection can be easily done in BTA, and since on a single path the deterministic Muller paradigm is equivalent to the nondeterministic Büchi one (see [Tho90]), we conclude that $\overline{L(M)}$ is in BTA, thus, $L(M)$ is in co–BTA. Moreover, since GCTA $\subseteq$ co–BTA and DMTA $\subseteq$ co–BTA but GCTA and MTA are not comparable (see Figure 1), we get that part 2 holds.

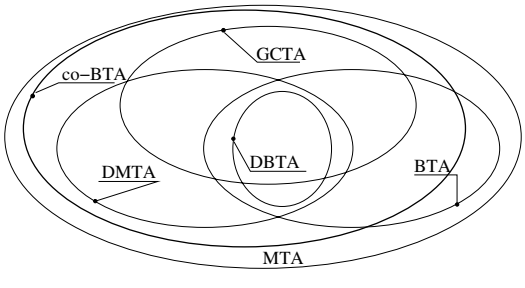


Fig. 3. Summary of the comparisons involving co–BTA

Part 3 follows directly from the non–closure under complementation of BTA (Theorem 1). Finally, to prove parts 4, 5, and 6 we can respectively use the languages $\{t \in T_\Sigma^\omega \mid t_0 \in L_1 \text{ and } t_{10} \in L_2 \text{ and } t_{11} \in \overline{L_1}\}$, $\{t \in T_\Sigma^\omega \mid t_0 \in L_1 \text{ and } t_1 \in \overline{L_1}\}$ and $\{t \in T_\Sigma^\omega \mid t_0 \in \overline{L_2} \text{ and } t_1 \in L_2\}$, where L_1, L_2, L_3, and their complements are given in Figure 2. □

5 Co–DGBTA

In this section, we deal with the class of languages whose complement is deterministically accepted by generalized Büchi tree automata (co–DGBTA, for short). We study the relationships of co–DGBTA with the other classes introduced so far and the complexity of the emptiness problem. Clearly, from the results obtained in the previous sections, the fact that DGBTA is polynomially equivalent to DBTA, and the fact that DBTA is a special case of BTA, it follows that the emptiness problem for co–DGBTA can be solved in exponential time. Here, we prove that this problem is indeed decidable in polynomial time.

The first example of class within BTA closed under complementation has been the remarkable characterization by Rabin [Rab72] of languages defined by a formula of weak monadic logic (where only quantifiers over finite sets are allowed): A language L is *weakly definable* if and only if both L and its complement $\overline{L}$ are accepted by Büchi tree automata. In [MS87], it is shown that a language is weakly definable if and only if it is accepted by a weak alternating Büchi tree automaton. The class of languages accepted by these automata includes co–DBTA. In general, we have the following strict containment.

Lemma 5. *DBTA ∪ co–DBTA ⊂ WABTA.*

Proof. Let $L = \{t \in T_\Sigma^\omega \mid t_0 \in L_3 \text{ and } t_1 \in \overline{L_3}\}$, where L_3 and $\overline{L_3}$ are given in Figure 2. Since L_3 is not in DBTA, it follows that L is not in DBTA ∪ co–DBTA. On the other hand, since both L_3 and $\overline{L_3}$ are in BTA, it follows that L is in WABTA. Hence, $L \in$ WABTA $\setminus$(DBTA ∪ co–DBTA). □

Recall that an alternating automaton is deterministic if and only if the transition relation δ does not use $\vee$ [MS95]. Directly form this definition and from the fact that complementing a WABTA by dualization gives an automaton with the same paradigm [MS87], we get the following characterization for co–DBTA.

Corollary 4. *Each language in co–DBTA is accepted by a WABTA whose transition relation contains only disjunctions.*

From Theorem 3, it follows that the emptiness problem for co–DBTA with the characterization given by Corollary 4 can be solved in exponential time. As we show in the following, this complexity can be reduced to a polynomial if we use a direct approach. First observe that a tree is not accepted by a DGBTA A if and only if the unique run r of A on t contains a path π that does not satisfy the acceptance condition. Thus, it is possible to characterize the complement

of a language accepted by a DGBTA A with an automaton that, for each tree, nondeterministically selects a path and then deterministically checks that π does not satisfy the acceptance condition of A. This last corresponds to check that π satisfies the generalized co–Büchi condition obtained dualizing the acceptance condition of A. Thus, we modify the definition of accepting run. Given a tree t and a GCTA B with an accepting condition $F = \{F_1, \ldots, F_k\}$, we say that a run $r \in Run_B(t)$ is $\exists$*–successful* if there exists a path π of r, such that $Inf(r/\pi) \subseteq F_i$ for some $F_i \in F$. A tree t is $\exists$*–accepted* by B if there exists an $\exists$–successful run of B on t. The language $\exists$–accepted by B is denoted by $L_\exists(B)$. In the next lemma, we show that the $\exists$–acceptance along with the generalized co–Büchi paradigm suffices to accept co–DGBTA.

Lemma 6. *Given a DGBTA B, there exists a DGCTA A such that $L_\exists(A) = \overline{L(B)}$. Moreover, if B is DBTA then A is DCTA.*

Proof. Let L be a language whose complement is accepted by a DGBTA $B = \langle Q, \Sigma, \delta, Q_0, \{F_1, \ldots, F_k\}\rangle$. Let $A = \langle Q, \Sigma, \delta, Q_0, \{Q \backslash F_i \mid i = 1, \ldots, k\}\rangle$ be a DGCTA. A tree $t \notin L(B)$ if and only if the only run r in $Run_B(t)$ (B is deterministic) is not successful. That is, r contains at least a path π such that $Inf(r/\pi) \cap F_i = \emptyset$, for some i. Thus, by the definition of $\exists$-acceptance, $t \notin L(B)$ if and only if $t \in L_\exists(A)$. □

With respect to the emptiness problem, notice that the characterization of co–DGBTA via "existential" tree automata is equivalent to consider tree automata as word automata[2]. In more details, given a DGBTA B, consider a generalized co-Büchi word automaton C that is obtained from B by dualizing the acceptance condition and splitting each transition into several transitions, one for each state successor. That is, for each transition (s, σ, s', s'') of B, we get two transitions (s, σ, s') and (s, σ, s'') of C. It is easy to verify that there is a tree that is not accepted by B if and only if there is a word accepted by C. Using this observation, we get an efficient algorithm for solving the emptiness problem for co–DGBTA. First, we notice that a generalized co-Büchi word automaton can be easily translated into a language equivalent generalized Büchi word automaton whose size is linear in the size of the starting automaton. Thus, given a DGBTA B with n states and k accepting sets, we can construct a Büchi word automaton A with $\mathcal{O}(nk)$ states such that the language accepted by A is empty if and only if the complement of the language accepted by B is empty. Since the emptiness problem for Büchi word automata is decidable in linear time [EL85], we get that checking for the emptiness of $\overline{L(B)}$ can be done in $\mathcal{O}(nk)$ time. Thus, the following theorem holds.

Theorem 7. *Given a DGBTA B with n states and index k, checking if $\overline{L(B)}$ is empty can be done in $\mathcal{O}(nk)$ time.*

[2] Tree automata generalize word automata, in the sense that a word is a tree of arity 1. Thus, we omit a formal definition of word automata here.

The rest of the section is devoted to compare co-DGBTA with the other classes considered in this paper.

Lemma 7. *Given a DGBTA B, there exists a GCTA A such that $L(A) = \overline{L(B)}$. Moreover, if B is DBTA then A is CTA.*

Proof. Let L be a language whose complement is accepted by a DGBTA $B = \langle Q, \Sigma, \delta, Q_0, \{F_1, \ldots, F_k\}\rangle$. We build a GCTA A that nondeterministically selects a path and on this path checks that the acceptance condition of B does not hold. Formally, $A = \langle \{q\} \cup Q, \Sigma, \delta', Q_0, \{\{q\} \cup Q \backslash F_i \mid i = 1, \ldots, k\}\rangle$ be a GCTA, where $q \notin Q$ and δ' is defined as follows. For each $(s, \sigma, s', s'') \in \delta$, the transition relation δ' contains (s, σ, q, s'') and (s, σ, s', q); moreover, δ' contains (q, σ, q, q). A tree $t \notin L(B)$ if and only if the only run r in $Run_B(t)$ (B is deterministic) is not successful. That is, r contains at least a path π such that $Inf(r/\pi) \cap F_i = \emptyset$, for some i. Thus, there exists a run r in $Run_A(t)$ such that for each path π, either $Inf(r/\pi) = \{q\}$ or there is an i such that $Inf(r/\pi) \subseteq Q \backslash F_i$. Hence, $t \notin L(B)$ if and only if $t \in L(A)$. □

From the above construction, notice that co–DGBTA can be linearly characterized by GCTA. In the next lemma, we show that co–DGBTA can be polynomially characterized by BTA (notice that it is linear starting from co–DBTA).

Lemma 8. *Given a DGBTA B, there exists a BTA A accepting $\overline{L(B)}$, whose size is polynomial in the size of B.*

Proof. By [Cho74], we can restrict to DBTA. Let L be a language whose complement is accepted by a DBTA $B = \langle Q, \Sigma, \delta, Q_0, F\rangle$. We build a BTA A that nondeterministically selects a path and on this path checks that the acceptance condition of B does not hold. Formally, $A = \langle Q', \Sigma, \delta', Q'_0, F'\rangle$ is such that *(i)* $Q' = Q \times \{0, 1, 2\}$; *(ii)* $Q'_0 = Q_0 \times \{0\}$; *(iii)* $F' = Q \times \{1, 2\}$; *(iv)* if $(s, \sigma, s', s'') \in \delta$, the transition relation δ' contains: $((s, 0), \sigma, (s', h), (s'', 1))$ and $((s, 0), \sigma, (s', 1), (s'', h))$, for $h \in \{0, 2\}$, $((s, 1), \sigma, (s', 1), (s'', 1))$, $((s, 2), \sigma, (s', 2), (s'', 1))$ for $s' \in Q \setminus F$, and $((s, 2), \sigma, (s', 1), (s'', 2))$ for $s'' \in Q \setminus F$. First, observe that size of A is linear in the size of B. Moreover, A accepts a tree t if and only if, for the only run r of B on t (B is deterministic), there exists a path π of r on which final states of B occur only finitely often. This is done by nondeterministically selecting a path π (unselected paths are marked with 1 in the second component of the states) and then checking that the property holds on π. For this purpose, on the selected path the second component of the states is nondeterministically set to 2. Once 2 is entered the run stops unless only states in $Q \setminus F$ are met on the selected path. Thus, $\overline{L(B)} = L(A)$, and the lemma is shown. □

Let us observe that the characterizations of co–DGBTA given in Lemmas 7 and 8 yield solutions to the emptiness problem for co–DGBTA via reductions to the same problem for GCTA and BTA, respectively. Since the best known upper bounds on the time complexity of the emptiness problem for GBTA and GCTA

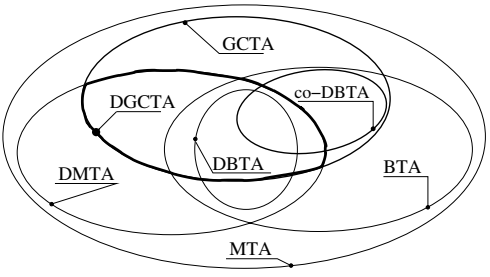


Fig. 4. Summary of the comparisons involving co–DBTA

are both quadratic in the number of states and linear in the index of the automaton [Rab70, LMN02], the time complexity resulting from these approaches is asymptotically worse than the upper bound stated in Theorem 7.

In the following lemmas, we complete the comparisons involving co–DBTA and the classes GCTA, (D)BTA and DMTA. The complete picture of the comparisons is given in Figure 4.

Lemma 9. *co–DBTA* $\subset$ *GCTA* $\cap$ *BTA.*

Proof. The inclusion co–DBTA $\subseteq$ GCTA $\cap$ BTA is a direct consequence of Lemmas 7 and 8. To prove the strict inclusion, we can use the language $L = \{t \in T_\Sigma^\omega \mid t_0 \in L_1 \text{ and } t_1 \in L_1\}$, where L_1 is given in Figure 2. □

Lemma 10

1. *(a)* $co\text{–}DBTA \cap DBTA \neq \emptyset$*;*
   (b) $co\text{–}DBTA \cap (DMTA \setminus DBTA) \neq \emptyset$*;*
   (c) $co\text{–}DBTA \setminus DMTA \neq \emptyset$*.*
2. *(a)* $((BTA \cap GCTA \cap DMTA) \setminus DBTA) \not\subseteq co\text{–}DBTA$*;*
   (b) $((BTA \cap GCTA) \setminus DMTA) \not\subseteq co\text{–}DBTA$*.*

Proof. Consider first part 1. To prove statement (a), we use the language $L = \{t \in T_\Sigma^\omega \mid \forall\ x \in \pi_0,\ t(x) = a\}$. For statements (b) and (c), we respectively use L_3 and L_1 given in Figure 2.

Consider now part 2. To prove statement (a) we use $L = \{t \in T_\Sigma^\omega \mid t_0 \in L_3 \text{ and } t_1 \in L_3\}$. Finally, for statement (b) we use $L = \{t \in L \mid t_0 \in L_1 \text{ and } t_1 \in L_1\}$, where L_1 and L_3 are given in Figure 2. □

6 Conclusion

Büchi and co–Büchi conditions are of interest for expressing requirements over nonterminating computations [GTW02]. For example, consider a drink-dispenser machine, we may want to express a requirement such as "users can always choose in the future coffee or tea" (typically a Büchi condition). In system verification,

we may want to prove that the computations of a system do not violate a requirement. In particular, in the automata theoretic approach, given a system model S and its specification φ, we can construct an automaton A capturing the computations of S and an automaton B capturing the negation of φ. Thus, S is correct with respect to ψ if $L(A) \cap L(B)$ is empty [VW86, VW94]. In the above example, the negation of the assertion consists of requiring that "users can be prevented from choosing both coffee and tea from a given point on" (a co–Büchi condition). Thus, to prove a model A of the drink-dispenser correct with respect to the first requirement, we can model the second requirement as a tree automaton with co–Büchi acceptance and check if its intersection with A is empty.

In this paper, we have dealt with co–Büchi acceptance for branching time specifications. As a characterization of this class we have considered alternating generalized co–Büchi tree automata (AGCTA). We have compared the corresponding class of tree languages with the main classes of languages accepted by tree automata, showing interesting relationships. In particular, it is worth to remark that this class strictly contains the class accepted by co–Büchi tree automata and is not comparable with that characterized via Büchi tree automata. As a consequence of the first result we obtain that the simulation theorem does not hold for the co–Büchi acceptance condition on tree automata.

We have also investigated the emptiness problem for AGCTA and its subclass of languages whose complement is accepted by deterministic generalized Büchi tree automata (co–DGBTA). For the general class, using a simple translation to parity automata, we have proved that the emptiness for AGCTA is in EXPTIME. This results is also complete since the emptiness problem for weak alternating Büchi tree automata is EXPTIME-hard. For the class co–DGBTA, we have shown a better bound, that is, the emptiness problem is decidable in quadratic time. For this purpose, we have used a linear-time characterization of this class of languages via generalized co-Büchi tree automata. In particular, given a deterministic generalized Büchi tree automaton A with n states and index k, we can check the emptiness for the complement of $L(A)$ in time $\mathcal{O}(nk)$.

References

[Büc62] J.R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

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

[EJ88] E.A. Emerson and C.S. Jutla. The complexity of tree automata and logics of programs. In *Proceedings 29th Annual IEEE Symp. on Foundations of Computer Science, FOCS'88*, pages 328 – 337, 1988.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings 32nd Annual IEEE Symp. on Foundations of Computer Science, FOCS'91*, pages 368–377, 1991.

[EL85] E.A. Emerson and C.L. Lei. Modalities for model-checking: Branching time logic strikes back. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[GPVW95] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3 – 18. Chapman & Hall, 1995.

[GTW02] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*, 2002.

[Jur98] Marcin Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.

[KV04] O. Kupferman and M.Y. Vardi. From complementation to certification. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS. Springer-Verlag, 2004.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[Lan69] L. H. Landweber. Decision problems for ω-automata. *Mathematical System Theory*, 3:376–384, 1969.

[LMN02] S. La Torre, A. Murano, and M. Napoli. Weak muller acceptance condition for tree automata. In *3rd Workshop in Verification, Model Checking, and Abstract Interpretation, VMCAI 2002. LNCS*, volume 2294: 240-254, 2002.

[McN66] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[MS87] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[MS95] D. E. Muller and P. E. Schupp. Simulating alternating tree automata by nondeterministic automata: new results and proofs of theorems of Rabin, McNaughton, and Safra. *Theoretical Computer Science*, 141:69–107, 1995.

[Rab69] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1 – 35, 1969.

[Rab70] M.O. Rabin. Weakly definable relations and special automata. *Mathematical Logic and Foundations of Set theory*, 1970.

[Rab72] M.O. Rabin. Automata on infinite objects and church's problem. *Trans. Amer. Math. Soc.*, 1972.

[Sei90] H. Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19:424–437, 1990.

[Tho90] W. Thomas. Automata on infinite objects. In J.van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol.B*, pages 133 – 191. 1990.

[VW86] M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32:182 – 211, 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 15:1 – 37, 1994.

[Wil01] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Soc. Math. Belg.*, 8(2), May 2001.

Foundations for the Run-Time Monitoring of Reactive Systems

- *Fundamentals of the MaC Language*

Mahesh Viswanathan[1] and Moonzoo Kim[2]

[1] CS Dept. University of Illinois at Urbana Champaign
vmahesh@cs.uiuc.edu
[2] CSE Dept. Pohang University of Science and Technology
moonzoo@postech.ac.kr

Abstract. As the complexity of systems grows, the correctness of systems becomes harder to achieve. This difficulty promotes a run-time monitoring technique as a promising complementary methodology for higher system assurance. To formalize and understand the computational nature of run-time monitoring is a key to utilize this valuable technique. In this paper, we formalize the notion of run-time monitoring of reactive systems in terms of ω-languages and show that the language of *Monitoring and Checking (MaC) architecture*, called MEDL, is expressive enough for the run-time monitoring.

First, we provide a descriptive theory for the class of monitorable languages and show that this class of languages coincides with the class Π_1^0 of the Arithmetic hierarchy. Second, we introduce a class of automata with storage that can be used to describe the class of monitorable languages using connections to the Arithmetic hierarchy. Finally, we show that MEDL can express the class of monitorable languages via the correspondence between MEDL and the automata with storage.

1 Introduction

Reactive systems are systems which perform ongoing interaction with an environment rather than generate output with given input. Computation is, therefore, typically seen as being non-terminating. Such systems are notorious for its complex behavior and difficulty of testing. As the complexity of systems grows, the correctness of systems becomes harder to achieve. This difficulty promotes a run-time monitoring technique not only as a performance measurement method, but also as a promising complementary method for higher system assurance. The monitor examines interaction of systems, rather than a result at the end of computation and determines whether the behavior is correct.

It is customary to model the behavior of such systems as an infinite sequence of letters from some finite alphabet. This sequence can be seen as either the sequence of program states visited by the reactive system, or as the sequence

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 543–556, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of request-response pairs that is generated by the system's interaction with its environment. Certifying the correctness of reactive systems, therefore, involves checking to see if the set of execution sequences of the reactive system satisfies certain constraints/properties.

Past research in monitoring [1] has tried to identify the class of monitorable properties[1] with the class of *safety* properties. We instead identify the class of properties that can be monitored with the class Π_1^0 in the Arithmetic hierarchy. Π_1^0 consists of properties whose violation can be detected by a Turing machine by examining a finite prefix of the errant behavior. We will also introduce a class of automata that can be used to specify these properties, and that can serve as monitors for such properties.

Section 2 shows that the class of languages run-time monitoring can determine, say $\mathcal{M}$, is a strict subset of the class of safety languages. Section 3 describes the class of monitorable languages $\mathcal{M}$ in the Arithmetic hierarchy. Section 4 introduces the model of finite state machines with storage which can specify $\mathcal{M}$. Section 5 briefly describes the Monitoring and Checking (MaC) architecture and the specification language Meta Event Definition Language (MEDL) of the MaC architecture. Then, we show that MEDL is expressive enough for $\mathcal{M}$. Finally, we enumerate related works in Sec 6 and Sec 7 concludes this paper.

2 A Class of Monitorable Languages $\mathcal{M}$

It is obvious that run-time monitoring cannot evaluate liveness properties because a monitor decides the correctness of system based on what has been observed. We generally presume that the class of properties which run-time monitoring can evaluate is safety properties. In this section, however, we study the class of properties run-time monitoring can evaluate more precisely.

2.1 Notations

We use standard notations of ω-languages. Σ is a finite alphabet. The set of finite words over Σ, including the empty word ϵ, is denoted by Σ^*, while the set of ω-words is Σ^ω; $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. A subset of Σ^* is called a finitary language, and a subset of Σ^ω is an infinitary language (or ω-language).

For a (finite or infinite) word α, $\alpha(i)$ (or α_i) denotes the $(i+1)$st letter of α. Segments of words are denoted as follows: $\alpha(m,n) = \alpha(m) \cdot \alpha(m+1) \cdots \alpha(n-1)$ and $\alpha(m,\omega) = \alpha(m) \cdot \alpha(m+1) \cdots$. The concatenation of a finite word u with another word (finite or infinite) α, $u \cdot \alpha$, is defined by $u \cdot \alpha(i) = u(i)$ if $i \leq |u|$ and $u \cdot \alpha(i) = \alpha(i - |u|)$ otherwise. A finite word u is said to be a prefix of another word α if there is $\beta \in \Sigma^\infty$ such that $u \cdot \beta = \alpha$. $\mathrm{pref}(\alpha)$ is the set of all finite prefixes of α, and for a (finite or infinite) language L, $\mathrm{pref}(L) = \cup_{\alpha \in L} \mathrm{pref}(\alpha)$.

[1] In this paper, we use two terms "property" and "language" for the same meaning depending on its context.

2.2 Safety Languages and Monitorable Languages

Informally speaking, safety languages are languages that require that nothing bad happens during an execution; if an execution is faulty, then the monitor should be able to reject it after looking at a *finite* prefix. Safety languages are formally defined by [2] as

Definition 1 (Safety language). *A language $L \subseteq \Sigma^\omega$ is a safety language if for every $\sigma \in \Sigma^\omega$, $\sigma \in L$ if and only if $\forall i \exists \beta \in \Sigma^\omega (\sigma(0,i) \cdot \beta \in L)$.*

It is clear from Definition 1 that a monitorable property is a safety language. A safety language, however, is *not* necessarily a monitorable language. The definition of safety language makes no computational assumptions. It is possible to define a language that is a safety language, but which is unlikely monitorable. For example, safety closure of the halting problem is a safety language but *not* a monitorable language.

Example 1. *Let $\Sigma = \{0, 1, a, b\}$. Consider a finite language $H_* = \{x \cdot a \cdot y \mid x, y \in \{0,1\}^*$, the Turing Machine encoded by x halts on input $y\}$. We define a language $H_\omega = H_* \cdot b^\omega \cup \{0,1\}^* \cdot a \cdot \{0,1\}^\omega \cup \{0,1\}^\omega$.*

The language H_ω, defined above is a safety language. In order to see this, we only need to observe that for any execution not in H_ω, there is a finite prefix when this violation can be detected. Executions not in H_ω are those that are not in the "right format", or where the finite prefix before the sequence of b's is not in H_*; in both cases there is a finite prefix that provides evidence of the execution not being in the language.

However, in order to detect that an execution σ is not in H_ω, we have to check for membership in H_*. Since membership in H_* (or the Halting problem) is not decidable, it is impossible for us to design monitors that would be able to detect a violation of this language. This suggests that the class of *monitorable language* is a strict subset of a class of safety languages; they should be such that sequences not in the languages should be recognizable by a Turing Machine, after examining a finite prefix. Therefore, we can define a monitorable language as follows.

Definition 2 (Monitorable language). *A language $L \subseteq \Sigma^\omega$ is said to be* monitorable *if and only if L is a safety language and $\Sigma^* \setminus \mathrm{pref}(L)$ is recursively enumerable. The class of monitorable languages is denoted by $\mathcal{M}$.*

3 $\mathcal{M}$ in the Arithmetic Hierarchy

In our study of ω-languages, we will find it useful to discuss definability relative to classical hierarchies in recursion theory and descriptive set theory. Such hierarchies have been extensively studied in the context of formal languages[3, 4, 5]. In the language theoretic context, the usual set-up of these hierarchies is modified slightly. The relations that we consider are not defined over natural numbers

and functions over natural numbers, but are rather over the finite and infinite words over a finite alphabet. While this change is irrelevant due to the presence of standard recursive encodings from Σ^* to $\mathbb{N}$, it provides a cleaner presentation for questions arising in automata theory.

A relation R is said to be *finitary* over Σ, if $R \subseteq (\Sigma^*)^m$. We write $Ru_1u_2 \dots u_m$ instead of $(u_1, u_2, \dots u_m) \in R$. We will define our hierarchy in terms of a class of finitary relations, $\mathcal{C}$. This class $\mathcal{C}$ will be assumed to be closed under boolean operations.

First, we will consider finitary languages over Σ. A languages $L \subseteq \Sigma^*$ is said to be in $\Sigma_n^0(\mathcal{C})$ if and only if for some relation $R \in \mathcal{C}$,

$$L = \{u \mid \exists v_1 \forall v_2 \dots Q_n v_n R v_1 v_2 \dots v_n u\}$$

where Q_n is either $\exists$ (if n is odd) or $\forall$ (if n is even). The languages in $\Pi_n^0(\mathcal{C})$ are defined analogously. $L \subseteq \Sigma^*$ is in $\Pi_n^0(\mathcal{C})$ if and only if for some relation $R \in \mathcal{C}$,

$$L = \{u \mid \forall v_1 \exists v_2 \dots Q_n v_n R v_1 v_2 \dots v_n u\}$$

where Q_n is either $\forall$ (if n is odd) or $\exists$ (if n is even).

The hierarchy of infinitary languages over $\mathcal{C}$ is defined as follows. A language $L \subseteq \Sigma^\omega$ is in $\Sigma_n^0(\mathcal{C})$ if and only if for some $R \in \mathcal{C}$,

$$L = \{\alpha \mid \exists v_1 \forall v_2 \dots Q_{n-1} v_{n-1} Q'_n i R v_1 v_2 \dots v_{n-1} \alpha(0, i)\}$$

where, once again, Q_{n-1} and Q'_n are quantifiers, and i is an natural number. The languages in $\Pi_n^0(\mathcal{C})$ are defined similarly in terms of logical formulae with alternating quantifiers, with the leading quantifier being $\forall$. Though we use the same notation for the hierarchy of infinitary languages, as in the case of finitary languages, it will often be clear from the context which hierarchy we are referring to.

By instantiating $\mathcal{C}$ to specific families, we obtain the classical hierarchies from recursion theory and descriptive set theory. If $\mathcal{C}$ is taken to be the class of recursive relations (REC), then we get the *arithmetic hierarchy*. For finitary languages, $\Sigma_1^0(REC)$ coincides with the class of recursively enumerable (R.E.) languages, while $\Pi_1^0(REC)$ is the class of co-R.E. languages. For notational convenience, we will denote the classes $\Sigma_n^0(REC)$ and $\Pi_n^0(REC)$, simply as, Σ_n^0 and Π_n^0, respectively.

Now we are ready to describe the class of monitoring languages $\mathcal{M}$ in terms of the Arithmetic hierarchy.

Proposition 1. $\mathcal{M} = \Pi_1^0$

Proof. [$\mathcal{M} \Rightarrow \Pi_1^0$] Consider $L \in \mathcal{M}$. From the definition of $\mathcal{M}$, we know that $\Sigma^* \setminus \text{pref}(L)$ is recursively enumerable. Therefore, there is a recursive relation R such that $u \in \Sigma^* \setminus \text{pref}(L)$ if and only if $\exists v R v u$. In other words, $u \in \text{pref}(L)$ if and only if $\forall v R' v u$, where $R' = \neg R$. Furthermore, we know that L is a safety language, which implies that $L \in adh(\text{pref}(L))$ where $adh(L) = \{\alpha \in \Sigma^\omega | \text{pref}(\alpha) \subseteq$

$\text{pref}(L)\}$ [6]. Hence, $\alpha \in L$ if and only if $\forall i \alpha(0,i) \in \text{pref}(L)$ if and only if $\forall i \forall v R' v \alpha(0,i)$. By contracting the quantifiers we can see that $L \in \Pi_1^0$.

$[\Pi_1^0 \Rightarrow \mathcal{M}]$ Let $L \in \Pi_1^0$. Hence $\alpha \in L$ if and only if $\forall i R \alpha(0,i)$, for some recursive relation R. From the definition of safety language (see Sect 2.2), it is clear that L is a safety language. Also, $u \in \Sigma^* \setminus \text{pref}(L)$ if and only if $\neg Ru$. Thus $\Sigma^* \setminus \text{pref}(L)$ is recursively enumerable, and $L \in \mathcal{M}$. □

4 ω-Automata with Storage

Finite state machines on infinite words, are very similar to those which accept finite words. On an ω-word, α, the machine works as if α were a "very large" finite word. The only difference is the criteria that these machines use to accept a language (clearly, acceptance by final state cannot be used).

The general notion of an automaton on ω-words, using some kind of storage, was first introduced and studied Engelfriet and Hoogeboom [7]. We use definitions and concepts described there, to develop our theory. Before defining finite state machines on ω-words formally, we first define the notion of a storage type, and give an example.

Definition 1. *A storage type is a 5-tuple $X = (C, C_0, P, F, [\![\cdot]\!])$, where*

- *C is a set of storage configurations,*
- *$C_0 \subseteq C$ is a set of initial storage configurations,*
- *P is a set of predicate symbols,*
- *F is a set of function symbols, and*
- *$[\![\cdot]\!]$ is a function that defines the semantics of the predicate and function symbols. For each $p \in P$, $[\![p]\!] : C \to \{\textbf{true}, \textbf{false}\}$, and for each $f \in F$, $[\![f]\!] : C \to C$, is a partial function.*

The set of all Boolean expressions over P, built using connectives $\wedge, \vee$, and $\neg$, constants $\{\textbf{true}, \textbf{false}\}$, and the predicates in P, is denoted by $BE(P)$. The function $[\![\cdot]\!]$ is extended to $BE(P)$ in the standard way. $[\![\cdot]\!]$ is also extended to finite words over F, by interpreting concatenation as function composition. In other words, $[\![f \cdot \varphi]\!] = [\![\varphi]\!] \circ [\![f]\!]$, where $\varphi \in F^*$ and $f \in F$.

Example 2. *The storage type,* accumulator*, is $AC = (\mathbb{N}, \{0\}, \{zero\}, \{+_k, -_k | k \in \mathbb{N}\}, [\![\cdot]\!])$. It is the storage type of integers with a test for zero, and ability to add and subtract constants. More precisely,*

$$[\![zero]\!](c) = \textbf{true} \text{ if and only if } c = 0$$
$$[\![+_k]\!](c) = c + k$$
$$[\![-_k]\!](c) = c - k, \text{ if } c \geq k, \text{ and undefined otherwise.}$$

We will now define the notion of the product of storage types. It is a way obtaining a new storage type that combines two storage types and uses them independently.

Definition 2. *Let* $X_1 = (C_1, C_{10}, P_1, F_1, [\![\cdot]\!]_1)$ *and* $X_2 = (C_2, C_{20}, P_2, F_2, [\![\cdot]\!]_2)$ *be two storage types with* $P_1 \cap P_2 = \emptyset$ *and* $F_1 \cap F_2 = \emptyset$*. The product of these two storage types,* $X_1 \times X_2$*, is the type* $(C, C_0, P, F, [\![\cdot]\!])$*, where* $C = C_1 \times C_2$*,* $C_0 = C_{10} \times C_{20}$*,* $P = P_1 \cup P_2$ *and* $F = F_1 \cup F_2$*. The function* $[\![\cdot]\!]$ *is then defined naturally, as follows.*

$$[\![p]\!](c_1, c_2) = \begin{cases} [\![p]\!]_1(c_1) \text{ if } p \in P_1 \\ [\![p]\!]_2(c_2) \text{ if } p \in P_2 \end{cases}$$
$$[\![f]\!](c_1, c_2) = \begin{cases} ([\![f]\!]_1(c_1), c_2) \text{ if } f \in F_1 \\ (c_1, [\![f]\!]_2(c_2)) \text{ otherwise} \end{cases}$$

We will often use the above definition to get finitely many copies of the same storage type. In such a case, we first rename the predicate and function symbols of the storage type, by adding subscripts, and then taking repeated products. The n-fold product ($n \geq 1$) of a storage type X will be denoted by X^n. In order to extend the definition consistently, we take $X^0 = (\{c\}, \{c\}, \emptyset, \emptyset, \emptyset)$.

We are now ready to define automata with storage type X. We will consider only one acceptance condition for such machines (see Def 4) [2]

Definition 3. *Let* $X = (C, C_0, P, F, [\![\cdot]\!])$ *be a storage type. An* X*-automata is a* 5*-tuple* $\mathcal{A} = (Q, \Sigma, \delta, q_0, c_0)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite input alphabet,*
- δ *is the transition function, which is a* finite *subset of* $Q \times (\Sigma \cup \{\epsilon\}) \times BE(P) \times Q \times F^*$*,*
- $q_0 \in Q$ *is the initial state, and*
- $c_0 \in C_0$ *is the initial storage configuration.*

The instantaneous description of such a machine $\mathcal{A}$, is a tuple $(q, \alpha, i, c) \in Q \times \Sigma^\omega \times \mathbb{N} \times C$, where q is the current state of the machine, α is the input to the machine, i is the position of the symbol being currently scanned, and c is the current configuration. In one step the machine either reads a symbol from the input or makes a "silent" transition, according to the transition function δ. More precisely, we say $(q, \alpha, i, c) \vdash (q', \alpha, i', c')$, if there exists a transition (q, a, φ, q', h), such that $[\![\varphi]\!](c) = \mathbf{true}$, $[\![h]\!](c)$ is defined, and $[\![h]\!](c) = c'$. Furthermore, we require that, either $a = \epsilon$ and $i = i'$, or $a = \alpha(i)$ and $i' = i + 1$. An infinite run of the automaton $\mathcal{A}$, on an input α, is an infinite sequence $\langle I_i \rangle_{i \in \mathbb{N}}$ of instantaneous descriptions, such that $I_0 = (q_0, \alpha, 0, c_0)$, $I_i \vdash I_{i+1}$, for each $i \in \mathbb{N}$, and for every $j \in \mathbb{N}$, there is a k such that I_k is scanning a position beyond j.

Definition 4. *An* ω*-word,* $\alpha \in \Sigma^\omega$*, is said to be accepted by an* X*-automaton* $\mathcal{A}$*, if there is an infinite run of the automaton on the input* α*. The language accepted by* $\mathcal{A}$*,* $L_{\mathcal{A}}$*, is the set of all* ω*-words accepted by* $\mathcal{A}$*.*

[2] For a discussion of the relative power of the various other acceptance conditions, readers are directed to [3, 4].

The above definition of acceptance coincides with Landweber's [8] 1′-acceptance and with the "always" acceptance of [7].

An automaton $\mathcal{A}$ is *deterministic* if for any state and storage configuration there is at most one possible next state and storage configuration. More formally, for any two tuples $(q_1, a_1, \varphi_1, q_1', h_1)$ and $(q_2, a_2, \varphi_2, q_2', h_2)$ in δ, with $q_1 = q_2$, either $a_1 \neq a_2$ and $a_1, a_2 \neq \epsilon$, or $[\![\varphi_1 \wedge \varphi_2]\!](c) = \mathbf{false}$, for every $c \in C$. Automata of particular interest to us will be what are called *real-time* automata. $\mathcal{A}$ is a real-time automata if it has no ϵ-transition, i.e., $\delta \subseteq Q \times \Sigma \times BE(P) \times Q \times F^*$. A slightly more general class of automata than real-time automata is the *finite delay* automata. An automaton $\mathcal{A}$ is said to be finite-delay, if there is no infinite run of the automaton on a finite word.

The class of ω-languages accepted by X-automata will be denoted by $X\mathcal{L}$; $X^*\mathcal{L} = \cup_n X^n\mathcal{L}$, where X^n is the n-fold product of the storage type X. The prefixes *d-*, *r-*, and *f-* will be used to denote the class of languages accepted by deterministic, real-time, and finite delay automata respectively. Similarly the prefix *dr-* (and *df-*) will be used for languages accepted by automata that are both deterministic and real-time (deterministic and finite delay).

Before presenting the automata theoretic characterization of $\mathcal{M}$, we define a storage type that will play an important role. This is the type of storage where one has finitely many integer locations that one can manipulate using addition, subtraction, multiplication and division. We will then prove a result relating the powers of real-time and finite delay automata with such a storage.

Definition 5. *The storage type of m integer variables $\mathbb{N}_m$ is given by $\mathbb{N}_m = (C, C_0, P, F, [\![\cdot]\!])$. $C = \mathbb{N}^m$ is the set of m-tuples of natural numbers, and $C_0 = \langle 0, \ldots, 0\rangle$. P consists of predicates $zero_i$, which test if the ith element of the current configuration is 0, i.e., $[\![zero_i]\!](\langle c_0, \ldots, c_{m-1}\rangle) = \mathbf{true}$ if and only if $c_i = 0$. There are various operations that one can perform on these configurations; one can add, subtract, and multiply integers to some element of the tuple, find the quotient or remainder when dividing an entry by an integer, and also add and subtract one entry in the tuple to another. The operation $ADR_{i,j}$ (add register) adds the ith entry to the jth entry; $SBR_{i,j}$ (subtract register) subtracts the ith entry from the jth entry. $ADC_{i,k}$ adds constant k to ith entry; similarly, $SBC_{i,k}$ and $MLC_{i,k}$ subtract and multiply constants k, while $QC_{i,k}$ and $RMC_{i,k}$ find the quotient and remainder when divided by k.*

As usual, $\mathbb{N}_m\mathcal{L}$ is the class of languages accepted by automata with storage type $\mathbb{N}_m$. By $\mathbb{N}_*\mathcal{L}$ we denote the class of languages $\cup_m \mathbb{N}_m\mathcal{L}$.

Theorem 1. *The following classes of ω-languages are equivalent.*

1. $\mathcal{M} = \Pi_1^0$
2. *df-*$\mathbb{N}_*\mathcal{L}$
3. *dr-*$\mathbb{N}_*\mathcal{L}$

Proof. [(1) $\Rightarrow$ (2)] For a language $L \in \Pi_1^0$, we know that $\alpha \in L$ if and only if $\forall i R\alpha(0, i)$, where R is a recursive language. Since R is a recursive language, there

exists a deterministic finite delay $\mathbb{N}_m$-automaton, $\mathcal{A}$, that has runs on exactly the same finite words as R. It is easy to see that the language accepted by $\mathcal{A}$ is$L$.

[(2) $\Rightarrow$ (3)] A real-time automaton may not have the "time" to do the computation performed by a finite delay machine. However, if the real-time machine can simulate a buffer, it has enough time to do everything done by the finite delay machine. The real-time automaton, then, reads an input symbol every time and puts it into the buffer, while the actual computation is then performed on the buffered input.

We basically show that $df\text{-}\mathbb{N}_m\mathcal{L} \subseteq dr\text{-}\mathbb{N}_{m+2}\mathcal{L}$. The two extra integer locations will be used by the real-time machine to simulate a buffer. The operations for manipulating a queue can be performed in one step using two locations, one storing the contents of the queue and the other storing some measure of the number of elements in the queue. Detailed proof is omitted.

[(3) $\Rightarrow$ (1)] Let $\mathcal{A}$ be the deterministic real-time automaton that accepts L. Observe that since $\mathcal{A}$ is real-time, it cannot distinguish between infinite runs that read the whole input and runs that do not read the whole input (because there are no such runs). Thus $\alpha \in L$ if and only if $\forall i \alpha(0, i)$ has a run. Hence, $L \in \Pi_1^0$. □

5 The Language of the Monitoring and Checking Architecture

5.1 Overview of the MaC Architecture

The *Monitoring and Checking (MaC)* architecture [9, 10] is a framework for monitoring and checking a running system with the aim of ensuring that the target program is running correctly with respect to a formal requirement specification. Fig 1 shows the overview of the MaC architecture.

The MaC architecture consists of three components: *filter, event recognizer*, and *run-time checker*. The filter extracts low-level information (such as values of program variables and time when variables change their values) from the instrumented code. The filter sends this information to the event recognizer, which detects primitive events and conditions where primitive events are changes of values, beginnings of functions, and endings of functions and primitive conditions are boolean variables or boolean statements composed by primitive typed variables. These events and conditions are then sent to a run-time checker. The run-time checker determines whether the current execution history satisfies the requirement specification.

Monitoring and checking as well as target program instrumentation are automatically performed from a given requirement specification, which makes the run-time analysis rigorous. In addition, monitoring program-dependent low-level behavior and checking high-level behavioral requirements are separated. This separation allows the specification of high-level requirements independent of the

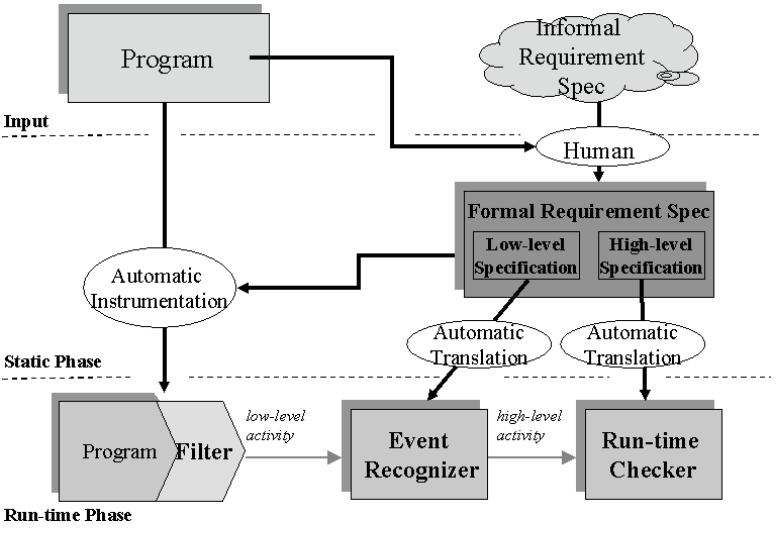


Fig. 1. Overview of the MaC architecture

implementation since implementation specific details are confined to the low-level specification. Furthermore, this modularity of the MaC architecture and well-defined interfaces among the components makes it easy to extend the architecture to incorporate third-party tools.

We have demonstrated the effectiveness of the MaC architecture using Java-MaC, a prototype implementation of the MaC architecture for Java programs, through several case studies [11, 12].

5.2 Specification Languages of the MaC Architecture

In this section, we give a brief overview of the formal specification languages used to describe specifications. The language for low-level specification is called Primitive Event Definition Language (PEDL). PEDL is used to define what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level specification by the event recognizer. High-level specifications are written in Meta Event Definition Language (MEDL). This separation ensures that the architecture is portable to different implementation languages and specification formalisms. Before presenting the two languages, we first define the notions of *event* and *condition*, which are fundamental to the MaC languages.

Events and Conditions. The MaC architecture assumes that it is possible to observe the behavior of the target system and evaluate the observed behavior to check whether required properties are satisfied or not. The observation is based on the occurrence of "interesting" state change in the target system. We use the notions of event and condition to capture interesting state changes.

Events occur instantaneously during the system execution, whereas *conditions* represent information that holds for a duration of time. For example, an event denoting return from method `RaiseGate` occurs at the instant the control returns from the method, while a condition `(position == 2)` holds as long as the variable `position` does not change its value from 2. The distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. The monitor can conclude that an event does not occur at any moment except when it receives an update from the filter. By contrast, once the monitor receives a message from the filter that variable `position` has been assigned the value 2, we can conclude that `position` retains this value until the next update.

We assume a countable set $\mathcal{C} = \{c_1, c_2, \ldots\}$ of primitive conditions. For example, these primitive conditions can be Java boolean expressions built from the monitored variables. In MEDL (see Sec 5.2), these will be conditions that were recognized by the event recognizer and sent to the run-time checker. We also assume a countable set $\mathcal{E} = \{e_1, e_2, \ldots\}$ of primitive events. Primitive events correspond to updates of monitored variables and calls/returns of monitored methods. The primitive events in MEDL are those that are reported by the event recognizer. Table 1 shows the syntax of conditions (C) and events (E).

Table 1. The syntax of conditions and events

$\langle C\rangle ::= c \mid \texttt{defined}(\langle C\rangle) \mid [\langle E\rangle,\langle E\rangle) \mid\ !\langle C\rangle \mid \langle C\rangle \&\& \langle C\rangle \mid \langle C\rangle \| \langle C\rangle \mid \langle C\rangle \Rightarrow \langle C\rangle$
$\langle E\rangle ::= e \mid \texttt{start}(\langle C\rangle) \mid \texttt{end}(\langle C\rangle) \mid \langle E\rangle \&\& \langle E\rangle \mid \langle E\rangle \| \langle E\rangle \mid \langle E\rangle\ \texttt{when}\ \langle C\rangle$

During execution, variables routinely become undefined when they are out of scope. We choose to use a three-valued logic, where the third value is taken to represent undefined (Λ). We interpret conditions over three values, $true$, $false$, and Λ. The predicate $\texttt{defined}(c)$ is true whenever the condition c has a well-defined value, namely, $true$ or $false$. Negation ($!c$), disjunction ($c_1 || c_2$), and conjunction ($c_1 \&\& c_2$) are interpreted classically whenever c, c_1 and c_2 take values $true$ or $false$; the only non-standard cases are when these take the value Λ. In these cases, we interpret them as follows. Negation of an undefined condition is Λ. Conjunction of an undefined condition with $false$ is $false$, and with $true$ is Λ. Disjunction is defined dually; disjunction of undefined condition and $true$ is $true$, while disjunction of undefined condition and $false$ is Λ. Implication ($c_1 \Rightarrow c_2$) is taken to $!c_1 || c_2$. For events, conjunction ($e_1 \&\& e_2$) and disjunction ($e_1 || e_2$) are defined classically; so $e_1 \&\& e_2$ is present only when both e_1 and e_2 are present, whereas $e_1 || e_2$ is present when either e_1 or e_2 is present.

There are some natural events associated with conditions, namely, the instant when the condition becomes $true$ ($\texttt{start}(c)$), and the instant when the condition becomes $false$ ($\texttt{end}(c)$). Notice that the event corresponding to the instant when the condition becomes Λ can be described as $\texttt{end}(\texttt{defined}(c))$. Also, any pair of events define an interval of time, so forms a condition $[e_1, e_2)$ that is $true$ from

event e_1 until event e_2. Finally, the event (e `when` c) is present if e occurs at a time when condition c is *true*.

Notice that MaC reasons about temporal behavior and data behavior of the target program execution using events and conditions; events are abstract representation of time and conditions are abstract representation of data. For formal semantics of events and conditions, see [9].

Primitive Event Definition Language (PEDL). PEDL is the language for writing low-level specifications. The design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in PEDL specifications. Second, we want the process of event recognition to be as simple as possible. Therefore, we limit the constructs of PEDL to allow one to reason only about the current state in the execution trace. The name, PEDL, reflects the fact that the main purpose of PEDL specifications is to define primitive events of requirement specifications. All the operations on events can be used to construct more complex events from these primitive events. PEDL is dependent on its target programming language.

Meta Event Definition Language (MEDL). The safety requirements are written in MEDL. Primitive events and conditions in MEDL specifications are imported from PEDL specifications. The overall structure of a MEDL specification is given in Fig 2.

```
ReqSpec <spec_name>

  /* Import section */
  import event <e>;
  import condition <c>;

  /*Auxiliary variable declaration*/
  var int <aux_v>;

  /*Event and condition definition*/
  event <e> = ...;
  condition <c>= ...;

  /*Property and violation definition*/
  property <c>  = ...;
  alarm <e> = ...;

  /*Auxiliary variable update section*/
  <e> -> {  <aux_v'> := ... ; }
End
```

Fig. 2. Structure of MEDL

Importing events and conditions. A list of events and conditions to be imported from an event recognizer is declared.

Defining events and conditions. Events and conditions are defined using imported events, imported conditions, and auxiliary variables, whose role is explained later in this section. These events and conditions are then used to define safety properties and alarms.

Safety properties and alarms. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must be *always* true during the execution. Alarms, on the other hand, are events that must never be raised (all safety properties [13] can be described in this way). Also observe that alarms and safety properties are complementary ways of expressing the same thing. The reason that we have both of them is because some properties are easier to think of in terms of conditions, while others are in terms of alarms.

Auxiliary variables. The language described in Sec 5.2 has a limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the ith occurrence of an event. For this purpose, MEDL allows users to define auxiliary variables, whose values may then be used to define events and conditions. Updates of auxiliary variables are triggered by events. For example,

```
e1 -> {count_e1' := count_e1 + 1;}
```

counts occurrences of event `e1`.

5.3 Expressive Power of MEDL

In this section, we show that MEDL is expressive enough for the monitoring purpose. More specifically, we show that for every dr-$\mathbb{N}_*$-automaton $\mathcal{A}_M$, there exists a MEDL script $M_{\mathcal{A}}$ which accepts exactly the same strings.

Theorem 2. *MEDL is expressive enough for $\mathcal{M}$.*

Proof. Consider a dr-$\mathbb{N}_*$-automaton $\mathcal{A}$. The elements of Σ (the input alphabet of $\mathcal{A}$) will be all the imported events, and there will be an auxiliary variable corresponding to each of the m storing locations of the automaton $\mathcal{A}$. In addition, there will be an auxiliary variable `state` that will store the state of the automaton. Let Pr be the set of all boolean expressions that label the edges of the automaton $\mathcal{A}$. Corresponding to each such boolean expression $b \in Pr$, we will define a condition $C_b = b$ and an event $E_b = start(C_b)$; note, that the expression b contains no primed variable. A transition (q_1, a, b, q_2, f) is transformed into a guard

$$(a \&\& E_b) \texttt{ when } (\texttt{state} == q_1) \texttt{ -> } \{\texttt{state}' := q_2; f';\}$$

where, f' is the sequence of updates that produces the same result as function f. Finally, the automaton accepts only those strings that do not cause it to be stuck at any point; this is captured by defining the safety property of the MEDL script to be something that says if `state ==` q, then the boolean expression labeling one of the out-going transitions must be true. It is clear that this MEDL script will behave exactly like the automaton. □

6 Related Work

Monitoring systems at runtime to ensure correctness has received a lot of attention recently, and many systems have been developed. There are monitoring systems that analyze programs written in C [14, 15] and Java [16, 17, 18, 9], by instrumenting the program to extract information. Different specification languages with varying expressive powers have been used to specify monitoring requirements ranging from simple boolean expressions [14] to some versions of propositional temporal logic [17] to extensions of propositional temporal logic [9] and logics for partial-order traces [19]. However, there has been very little work in understanding the fundamental limitations of what properties can and cannot be monitored. In the seminal paper [1], monitorable properties are identified with safety properties. This was refined in [6]. More recently, Hamlen et. al. [20] have identified the class of properties that can be enforced; namely properties that can be detected and for which corrective action can be taken before a serious violation happens. The class of properties they identify as enforceable is strict subset of the class identified in this paper. The difference between these classifications stems from the fact that in this paper, we are only concerned with the problem of monitoring to detect errors (possibly after the violation has occurred) and not in enforceable properties.

7 Conclusion and Future Work

Run-time monitoring can serve as a complementary method, in addition to formal verification and testing, for assurance of the systems' correctness. In this paper, we have formalized the computational nature of run-time monitoring, which is necessary for utilizing this valuable technique. We have provided a descriptive theory for the class of monitorable languages $\mathcal{M}$ and showed that MEDL, the specification language of the MaC architecture, is expressive enough for $\mathcal{M}$. We showed that $\mathcal{M}$ is a strict subset of the class of safety languages and $\mathcal{M}$ corresponds to Π_1^0 in the Arithmetic hierarchy. Also, we introduced a class of automata with storage which can specify $\mathcal{M}$, then showed that there exists a MEDL specification which can express such automaton. Therefore, the MaC architecture, whose specification language is MEDL, can be a general framework for run-time monitoring.

Although the MaC architecture provides an expressive language MEDL, it is sometimes awkward to express certain features like temporal ordering of complex events in MEDL. Extending MEDL for specifying requirements more easily could be one further research direction. For example, [21] extends MEDL for describing regular expressions more conveniently.

References

1. Schneider, F.B.: Enforceable security policies. Technical Report TR98-1664, Cornell University (1998)
2. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21** (1985) 181–185

3. Staiger, L.: Hierarchies of recursive ω-languages. Journal of Information Processing and Cybernetics EIK **22** (1986) 219–241
4. Thomas, W.: Automata on infinite objects. In Leeuwen, J.V., ed.: Handbook of Theoretical Computer Science. Volume B. Elsevier, Amsterdam (1990) 133–191
5. Staiger, L.: ω-languages. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Volume 3. Springer-Verlag, Berlin (1997) 339–387
6. Viswanathan, M.: Foundations for the Run-time Analysis of Software Systems. PhD thesis, Department of Computer and Information Science, University of Pennsylvania (2000)
7. Engelfriet, J., Hoogeboom, H.J.: X-automata on ω-words. Theoretical Computer Science **110** (1993) 1–51 Earlier version in Proceedings of the International Colloquium on Automata, Languages and Programming, pages 289–303, 1989.
8. Landweber, L.H.: Decision problems for ω-automata. Mathematical Systems Theory **3** (1969) 376–384
9. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design (2004)
10. Kim, M.: Information Extraction for Run-time Formal Analysis. PhD thesis, Department of Computer and Information Science, University of Pennsylvania (2001)
11. Bhargavan, K., Gunter, C., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal analysis of network simulations. IEEE Transaction on Software Engineering (2002)
12. Kim, M., Lee, I., Sammapun, U., Shin, J., Sokolsky, O.: Monitoring, checking, and steering of real-time systems. In: Runtime Verification, Copenhagen Denmark. (2002)
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, New York (1992)
14. Jeffery, C., Zhou, W., Templer, K., Brazell, M.: A lightweight architecture for program execution monitoring. In: ACM Workshop on Program Analysis for Software Tools and Engineering. (1998)
15. Templer, K.S., Jeffery, C.: A configurable automatic instrumentation tool for ansi c. In: Proceedings of the International Conference on Automated Software Engineering. (1998)
16. Mok, A.K., Liu, G.: Early detection of timing constraint violation at run-time. In: Proceedings of the IEEE Real-Time Systems Symposium. (1997)
17. Havelund, K., Rosu, G.: Monitoring Java Programs with JavaPathExplorer. In: Proceedings of the Workshop on Runtime Verification. Volume 55 of Electronic Notes in Theoretical Computer Science., Elsevier Publishing (2001)
18. Sen, K., Rosu, G., Agha, G.: Runtime Safety Analysis of Multithreaded Programs. In: Proceedings of the International Conference on Automated Software Engineering. (2003)
19. Sen, A., Garg, V.K.: Partial Order Trace Analyzer (POTA) for Distributed Systems. In: Proceedings of the Workshop on Runtime Verification. Electronic Notes in Theoretical Computer Science, Elsevier (2003)
20. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems (2004, to appear) Available from `http://www.cs.cornell.edu/fbs/publications/EnfClasses.pdf`.
21. Sammapun, U., Easwaran, A., Lee, I., , Sokolsky, O.: Simulation of simultaneous events in regular expressions for run-time verification. In: Runtime Verification, Barcelona, Spain. (2004)

A Summary of the Tutorials at ICTAC 2004

Zhiming Liu

International Institute for Software Technology,
The United Nations University, Macau
Z.Liu@iist.unu.edu

Abstract. ICTAC 2004 provided six tutorials on advanced topics related to the theme of ICTAC, given by recognized worldwide experts. Here, we include a brief summary of each tutorial.

1 Introduction

The tutorial program at ICTAC provides opportunities for conference attendees to get knowledge, insights and abilities on key subjects on theoretical aspects of computing. It is intended for practitioners, researchers, educators and students looking for a better and deeper understanding of up to date theories, methods and tools. We believe that this is particularly helpful for developing countries to strengthen their research, teaching and development in computer science and engineering.

The tutorial program at ICTAC 2004 was very popular and successful. The quality of the proposals was very high and it is a pity that we had space only for only six. As a result, a number of good proposals were not accepted. We hope the prospective speakers will be encouraged to submit again to ICTAC 2005.

In the six that were selected, there was a good coverage of areas in theories, practical formal engineering methods and tools, that had great appeal and relevance to the theoretical computer science community. In the following section, we summarise these tutorials. Further detailed information can be obtained at the ICTAC 2004 website: http://www.iist.unu.edu/ICTAC2004/tutorials.html.

2 The Tutorials

Tutorial 1: Theorem Proving with Isabelle/HOL

Lecturer: Tobias Nipkow, Techical University Munich

Abstract: Isabelle/HOL is an interactive theorem prover for HOL, a popular version of higher-order logic. The purpose of this 12-hour tutorial is to familiarise students, researchers and practitioners with the basics of specification and proof in Isabelle/HOL. The course combines traditional lectures, on-line demos, and practical exercises for the participants. At the end of the course participants should be able to formalize functional programs and set-theoretic system models and prove (with Isabelle's help) simple properties about them. The following topics are covered:

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 557–560, 2005.
© Springer-Verlag Berlin Heidelberg 2005

- Datatypes and recursive functions
- Proof by structural induction and by simplification
- Proofs in propositional logic and predicate logic
- Set theory and inductively defined sets
- A language of readable proofs.

Tutorial 2: Formal Theories of Software Testing

Lecturer: Hong Zhu, Oxford Brookes University, UK

Abstract: Software testing has been considered as in a lack of solid theoretical foundation for a long time. In the past thee decades, serious efforts have been attempted by researchers to lay a sound foundation of software testing. A number of formal theories have been advanced. In this half-day tutorial, we will present a systematic introduction the formal theories of software testing developed over the past decades. It will consist of the following three parts. Part 1 will give a brief introduction to the basic concepts and methods of software testing. Formal definitions of the concepts and testing methods will be presented. Part 2 will be devoted to the axiomatic studies of software test adequacy criteria. Various axiom systems will be discussed and analysed. The assessments of testing methods against axioms will be reviewed. The results of the investigations on the relationships between testing and software correctness and reliability through interpretations of the axiom systems by inductive inferences will be reported. Part 3 will focus on the problem of test oracle. The theories and methods of automated test oracle based on the concept of observation contexts in algebraic specification-based testing will be examined. A general theory of behaviour observation based on the domain theory of program semantics will be introduced. Axioms of well-defined behaviour observation methods will be discussed. Problems in testing concurrent systems and component-based systems will also be examined.

Tutorial 3: Formal Aspects of Software Architecture

Lecturer: José Luiz Fiadeiro, University of Leicester, England

Abstract: This half-day tutorial presents a formal approach to Software Architecture. The key architectural concepts - component, connector and configuration - are formalised in CommUnity, a prototype architectural description language that has a mathematical semantics and is supported by a graphical tool. We show how the principle of superposition can support the separation between three main architectural concerns - computation, coordination and distribution. Finally, we show how architectures enable the incremental development and compositional evolution of systems through graph-based reconfiguration techniques.

Tutorial 4: Formal Engineering Methods for Industrial Software Development - An Introduction to the SOFL Specification Language and Method

Lecturer: Shaoying Liu, Hosei University, Japan

Abstract: This half-day tutorial offers a systematic introduction to the SOFL specification language, method, process, and supporting tools. As a specification language, SOFL integrates VDM-SL, Data Flow Diagrams, and Petri Nets to provide an intuitive, rigorous, and comprehensible formal notation for specifications at different levels. Compared to UML (Unified Modeling Language), SOFL provides a simpler but systematic mechanism for precisely defining the functions of system units and their integration, and therefore avoids the difficulty in managing different kinds of diagrams and their consistency in UML.

The tutorial is divided into three parts. The first part includes the brief introduction to Formal Engineering Methods and the SOFL specification language. In particular, we will focus on the explanation of the idea of using the graphical notation, known as Condition Data Flow Diagram (or CDFD for short), to model the architecture of a system, while using the pre-post notation to define the functionality of processes occurring in CDFDs. The second part explains the SOFL method and process: how SOFL can be used to construct a formal specification by taking a three-step: informal, semi-formal, and formal specifications. It also explains how structured abstract design can be smoothly transformed into object-oriented detailed design and programs. Finally, the third part presents Rigorous Review and Specification Testing as two practical techniques for verification and validation of specifications, and demonstrates several tools we have built to support SOFL.

Tutorial 5: Program Transformation Systems: Theory and Practice for Software Generation, Maintenance and Reengineering

Lecturer: Hongjun Zheng , Semantic Designs, Inc., Austin, USA

Abstract: This half-day tutorial provides an integrated view, built over 20 years, of program transformation systems, on concepts, vocabulary, mechanisms, and discussion of some existing systems from this view. Software engineering and software maintenance automation support will come from such semantically founded tools. Of particular interest to this conference, the tutorial will elaborate a well-founded theory of software maintenance using the transformational perspective. It will describe a set of practical transformation systems, and provide some application experience based on DMS, the transformation toolset that Semantic Designs is building. A number of real-world applications of transformations will be described, including OO component reengineering (a task-specific refactoring), automated translation of JOVIAL to C (legacy software migration), test coverage and profiling analysis, and automated clone detection and removal (for million-line COBOL and Java systems).

Tutorial 6: Functional Predicate Calculus and Generic Functionals in Software Engineering

Lecturer: Raymond Boute, University of Ghent, Belgium

Abstract: By formal calculation we mean expression manipulation on the basis of syntax, “letting the symbols do the work”. This is a valuable complement to (some might

even say, replacement for) intuitive reasoning, especially when exploring new grounds where intuition is (still) clueless. By this tutorial, we wish to open for software engineers the same possibilities that calculus has offered to physicists and engineers in more classical areas (mechanical, electrical, ...). This half-day tutorial presents the principles of two interwoven formalisms and illustrates their applications to formal reasoning in various aspects of software engineering. The first formalism is a discipline for designing generic functionals, some of which constitute generalizations of often-used functionals in mathematics (composition, restriction, inverse, etc.), whereas others are new. A small collection of generic functionals and their algebraic properties appears sufficient for providing considerable expressive and calculational power in a wide diversity of fields, illustrated with examples in program transformation, data types, program semantics, databases. The second formalism is a functional predicate calculus, thus called because predicates and quantifiers are functions and all algebraic laws are derived from the basic quantifier axioms using the axioms for function equality. The laws are most elegantly expressed using generic functionals and, conversely, the predicate calculus is most convenient for proving the properties of the generic functionals, which explains the synergy. The collection of laws necessary for conveniently covering the diversity of logical arguments encountered in practice is larger than found in traditional logic textbooks (a fact also made evident by the extensive list in Gries and Schneider's "A Logical Introduction to Discrete Math"). However, it is still very manageable, especially by proper grouping. Its power resides in guiding the flow of the calculations by the shape of the formulas and, if used in this fashion, it makes the development of logical arguments in continuous mathematics and in discrete mathematics remarkably similar and, above all, easy and reliable. This is illustrated by (a) general methods for reasoning about functions, relations and proofs by induction (perhaps one of the most important proof techniques in software engineering) and (b) showing how theories of programming that are useful in program construction and verification can be calculationally derived from a very simple model based on "program equations" in a way similar to systems modelling in classical engineering. Other examples are kept available for illustration and discussion as time permits.

Author Index

Ábrahám, Erika 37
Aichernig, Bernhard K. 250
Aiguier, Marc 415
Alagar, Vasu 387
Arun-Kumar, S. 99

Barbosa, Luís Soares 52
Barbosa, Marco Antonio 52
Belli, Fevzi 220
Béroff, Stefan 415
Bonsangue, Marcello M. 37
Budnik, Christof J. 220

Cavalcanti, Ana 478
Chen, Yuting 235
Cheung, Ling 494
Chin, Wei Ngan 187
Colin, Samuel 431

Dan, Li 250
Daws, Conrado 280
de Boer, Frank S. 37
Dong, Jin Song 265
Dybjer, Peter 341

Feng, Yuzhang 265
Fiadeiro, José Luiz 1
Fissore, Olivier 356

Gnaedig, Isabelle 356

Ionescu, Mihai 68
Ishdorj, Tseren-Onolt 68

Janicki, Ryszard 84
Jiang, Ying 140
Jifeng, He 14, 478
Jin, Beihong 154

Kammüller, Florian 310
Kapur, Deepak 325
Kim, Moonzoo 543
Kirchner, Hélène 356
Korade, Neelesh 99
Krishnan, Padmanabhan 511

La Torre, Salvatore 527
Leino, K. Rustan M. 35
Li, Xiaoshan 170
Li, Yuan Fang 265
Lin, Huimin 36
Liu, Shaohua 154
Liu, Shaoying 235
Liu, Zhiming 170, 557
Long, Quan 170
Lynch, Nancy 494

Ma, Yinglong 154
Mariano, Georges 431
Mitchell, Bill 113
Murano, Aniello 527

Nagoya, Fumiko 235
Naiyong, Jin 14
Niaouris, Apostolos 447

Paquet, Joey 387
Poirriez, Vincent 431

Qi, Zhichang 204
Qiao, Haiyan 341
Qin, Shengchao 187

Ranise, Silvio 372
Reeves, Steve 128
Ringeissen, Christophe 372
Rodríguez-Carbonell, Enric 325

Sampaio, Augusto 478
Sanders, J.W. 310
Schäfer, Andreas 463
Schobbens, Pierre-Yves 415
Segala, Roberto 494
Sherif Adnan 478
Steffen, Martin 37
Streader, David 128

Takeyama, Makoto 341
Thai, Pham Hong 295
Tran, Duc-Khanh 372

Vaandrager, Frits 494
Van Hung, Dang 295
Viswanathan, Mahesh 543
Vu Tran, Viet-Anh 187

Wan, Kaiyu 387
Wang, Ji 204
Wang, Ya-Sha 140
Wang, Yisong 403
Wei, Jun 154
Wen, Yanjun 204

Xie, Bing 140

Yang, Jing 170

Zhang, Lu 140
Zhang, Mingyi 403
Zhao, Jun-Feng 140