

Dominique Méry
Stephan Merz (Eds.)

LNCS 6396

Integrated Formal Methods

8th International Conference, IFM 2010
Nancy, France, October 2010
Proceedings

 Springer

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

Alfred Kobsa

University of California, Irvine, CA, 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

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Dominique Méry Stephan Merz (Eds.)

Integrated Formal Methods

8th International Conference, IFM 2010
Nancy, France, October 11-14, 2010
Proceedings

Volume Editors

Dominique Méry

Stephan Merz

INRIA Nancy-Grand Est & LORIA, Bâtiment B, équipe MOSEL

615 rue du Jardin Botanique, 54602 Villers-lès-Nancy cédex, France

E-mail: {Dominique.Mery; Stephan.Merz@loria.fr}

Library of Congress Control Number: 2010935597

CR Subject Classification (1998): D.2, F.3, D.3, D.2.4, F.4.1, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743

ISBN-10 3-642-16264-9 Springer Berlin Heidelberg New York

ISBN-13 978-3-642-16264-0 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.com

© Springer-Verlag Berlin Heidelberg 2010

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper 06/3180

Preface

This volume contains the proceedings of IFM 2010, the 8th International Conference on Integrated Formal Methods. The conference took place October 12–14, 2010, at the INRIA research center and the LORIA laboratory in Nancy, France. Previous editions were held in York, Dagstuhl, Turku, Canterbury, Eindhoven, Oxford, and Düsseldorf. The IFM conference series seeks to promote research into the combination of different formal methods, including the combination of formal with semiformal methods, for system development. Such combinations are useful in order to apprehend different aspects of systems, including functional correctness, security, performance, and fault-tolerance. The conference provides a forum for discussing recent advances in the state of the art and for disseminating the results among the academic and industrial community.

IFM 2010 received 59 submissions, covering the spectrum of integrated formal methods and ranging from formal and semiformal notations, semantics, refinement, verification, and model transformations to type systems, logics, tools, and case studies. Each submission was reviewed by at least three members of the Program Committee. The committee decided to accept 20 papers. The conference program also included invited talks by Christel Baier, John Fitzgerald, and Rajeev Joshi. The conference was preceded by a day dedicated to the Workshop on Formal Methods for Web Data Trust and Security (WTS 2010) and two tutorials, one on the verification of *C#* programs using *Spec#* and *Boogie 2*, by Rosemary Monahan, and the other on the *TLA⁺* proof system, by Denis Cousineau and Stephan Merz.

We are grateful to the members of the Program Committee and the external reviewers for their care and diligence. The reviewing process and the preparation of the proceedings were facilitated by the EasyChair system that we highly recommend to every program chair. We thank the INRIA Nancy research center for organizational and logistic support, and gratefully acknowledge the financial support by CNRS (through GDR GPL), Nancy University, GIS 3SGS, the Lorraine Region, and the Greater Nancy.

October 2010

Dominique Méry
Stephan Merz

Conference Organization

Program Chairs

Dominique Méry University of Nancy, France
Stephan Merz INRIA Nancy, France

Program Committee

Yamine Aït-Ameur ENSMA Poitiers, France
Jean-Paul Bodeveix University of Toulouse, France
Bernard Boigelot University of Liège, Belgium
Eerke Boiten University of Kent, UK
Jim Davies University of Oxford, UK
David Déharbe UFRN Natal, Brazil
John Derrick University of Sheffield, UK
Jin Song Dong University of Singapore, Singapore
Wan Fokkink Free University of Amsterdam, The Netherlands
Martin Fränzle University of Oldenburg, Germany
Andy Galloway University of York, UK
Hubert Garavel INRIA Grenoble, France
Diego Latella CNR Pisa, Italy
Stefan Leue University of Konstanz, Germany
Michael Leuschel University of Düsseldorf, Germany
Heiko Mantel Technical University of Darmstadt, Germany
Jun Pang University of Luxemburg, Luxemburg
David Pichardie INRIA Rennes, France
Wolfram Schulte Microsoft Research, USA
Graeme Smith University of Queensland, Australia
Martin Steffen University of Oslo, Norway
Kenji Taguchi NII Tokyo, Japan
Helen Treharne University of Surrey, UK
Elena Troubitsyna Åbo Akademi, Finland
Heike Wehrheim University of Paderborn, Germany

Local Organization

Nicolas Alcaraz
Anne-Lise Charbonnier
Rachida Kasmı
Dominique Méry
Stephan Merz

External Reviewers

Erika Abraham	Mohammad M. Jaghoori	Neeraj Singh
Markus Aderhold	Suresh Jagannathan	Heiko Spiess
Maurice H. ter Beek	Michael Jastram	Barbara Sprick
Nazim Benaïssa	Matthias Kuntz	Dominik Steenken
Yves Bertot	Hironobu Kuruma	Volker Stolz
Sandrine Blazy	Peter Ladkin	Martin Strecker
Andrea Bracciali	Linas Laibinis	Henning Sudbrock
Erik Burger	Florian Leitner-Fischer	Toshinori Takai
Taolue Chen	Peter Lindsay	Anton Tarasyuk
Véronique Cortier	Yang Liu	Tino Teige
Frédéric Dabrowski	Michele Loreti	Thi Mai Thuong Tran
Mohammad T. Dashti	Alexander Lux	Nils Timm
Henning Dierks	Mieke Massink	Sebastian Uchitel
Xinyu Feng	Alexander Metzner	Chen-wei Wang
Mamoun Filali-Amine	Martin Musicante	James Welch
Pascal Fontaine	Anantha Narayanan	Bernd Westphal
Richard Gay	Khanh Nguyen Truong	Anton Wijs
Stefan Hallerstede	Daniel Plagge	Kirsten Winter
Ian J. Hayes	Jean-Baptiste Raclet	Peng Wu
Keijo Heljanko	Thomas Ruhroth	Shaojie Zhang
Alexei Iliasov	Christoph Scheben	Xian Zhang
Ethan Jackson	Rudi Schlatte	Huiquan Zhu

Sponsoring Institutions

- Centre de Recherche INRIA Nancy-Grand Est
- Nancy Université: Université Henri Poincaré Nancy 1, Institut National Polytechnique de Lorraine
- CNRS: GDR GPL – Génie de la Programmation et du Logiciel
- GIS 3SGS: Surveillance, Sûreté et Sécurité des Grands Systèmes
- Communauté Urbaine du Grand Nancy
- Région Lorraine

Table of Contents

On Model Checking Techniques for Randomized Distributed Systems (Invited Talk)	1
<i>Christel Baier</i>	
Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems (Invited Talk)	12
<i>John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, and Sune Wolff</i>	
Programming with Miracles (Invited Talk)	27
<i>Rajeev Joshi</i>	
An Event-B Approach to Data Sharing Agreements	28
<i>Alvaro E. Arenas, Benjamin Aziz, Juan Bicarregui, and Michael D. Wilson</i>	
A Logical Framework to Deal with Variability	43
<i>Patrizia Asirelli, Maurice H. ter Beek, Alessandro Fantechi, and Stefania Gnesi</i>	
Adding Change Impact Analysis to the Formal Verification of C Programs	59
<i>Serge Autexier and Christoph Lüth</i>	
Creating Sequential Programs from Event-B Models	74
<i>Pontus Boström</i>	
Symbolic Model-Checking of Optimistic Replication Algorithms	89
<i>Hanifa Boucheneb, Abdessamad Imine, and Manal Najem</i>	
From Operating-System Correctness to Pervasively Verified Applications	105
<i>Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt</i>	
A Compositional Method for Deciding Equivalence and Termination of Nondeterministic Programs	121
<i>Aleksandar Dimovski</i>	
Verification Architectures: Compositional Reasoning for Real-Time Systems	136
<i>Johannes Faber</i>	

Automatic Verification of Parametric Specifications with Complex Topologies	152
<i>Johannes Faber, Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans</i>	
Satisfaction Meets Expectations: Computing Expected Values of Probabilistic Hybrid Systems with SMT	168
<i>Martin Fränzle, Tino Teige, and Andreas Eggers</i>	
Showing Full Semantics Preservation in Model Transformation – A Comparison of Techniques	183
<i>Mathias Hülsbusch, Barbara König, Arend Rensink, Maria Semenyak, Christian Soltzenborn, and Heike Wehrheim</i>	
Specification and Verification of Model Transformations Using UML-RSDS	199
<i>Kevin Lano and Shekoufeh Kolahdouz-Rahimi</i>	
Multiformalism and Transformation Inheritance for Dependability Analysis of Critical Systems	215
<i>Stefano Marrone, Camilla Papa, and Valeria Vittorini</i>	
Translating Pi-Calculus into LOTOS NT	229
<i>Radu Mateescu and Gwen Salaün</i>	
Systematic Translation Rules from ASTD to Event-B	245
<i>Jérémy Milhau, Marc Frappier, Frédéric Gervais, and Régine Laleau</i>	
A CSP Approach to Control in Event-B	260
<i>Steve Schneider, Helen Treharne, and Heike Wehrheim</i>	
Towards Probabilistic Modelling in Event-B	275
<i>Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis</i>	
Safe Commits for Transactional Featherweight Java	290
<i>Thi Mai Thuong Tran and Martin Steffen</i>	
Certified Absence of Dangling Pointers in a Language with Explicit Deallocation	305
<i>Javier de Dios, Manuel Montenegro, and Ricardo Peña</i>	
Integrating Implicit Induction Proofs into Certified Proof Environments	320
<i>Sorin Stratulat</i>	
Author Index	337

On Model Checking Techniques for Randomized Distributed Systems

Christel Baier

Technische Universität Dresden, Faculty of Computer Science, Germany

Abstract. The automata-based model checking approach for randomized distributed systems relies on an operational interleaving semantics of the system by means of a Markov decision process and a formalization of the desired event E by an ω -regular linear-time property, e.g., an LTL formula. The task is then to compute the greatest lower bound for the probability for E that can be guaranteed even in worst-case scenarios. Such bounds can be computed by a combination of polynomially time-bounded graph algorithm with methods for solving linear programs. In the classical approach, the “worst-case” is determined when ranging over all schedulers that decide which action to perform next. In particular, all possible interleavings and resolutions of other nondeterministic choices in the system model are taken into account. The worst-case analysis relying on this general notion of schedulers is often too pessimistic and leads to extreme probability values that can be achieved only by schedulers that are unrealistic for parallel systems. This motivates the switch to more realistic classes of schedulers that respect the fact that the individual processes only have partial information about the global system states. Such classes of partial-information schedulers yield more realistic worst-case probabilities, but computationally they are much harder. A wide range of verification problems turns out to be undecidable when the goal is to check that certain probability bounds hold under all partial-information schedulers.

Probabilistic phenomenon appear rather natural in many areas of computer science. Randomized algorithms, performance evaluation, security protocols, control theory, stochastic planning, operations research, system biology or resilient systems are just a few examples. Although a wide range of different stochastic models are used in these areas, it is often possible to deal with Markovian models. These rely on the memoryless property stating that the future system behavior only depends on the current state, but not on the past. If the state space is finite, then Markovian models can be viewed as a variant of classical finite automata augmented with distributions which makes them best suited for the application of model checking techniques.

In this extended abstract, we summarize the main steps of the automata-based model checking approach for the quantitative analysis of Markov decision processes in worst-case scenarios, and point out the difficulties that arise when taking the local views of the processes into account.

Markov decision processes (MDPs) can be understood as a probabilistic extension of labeled transition systems. Nondeterminism can be represented in an MDP by the choice between different actions. The actions of an MDP can have a probabilistic effect, possibly depending on the state in which they are executed.

The coexistence of nondeterminism and probabilism in an MDP allows for representing concurrent (possibly randomized) activities of different processes by interleaving, i.e., the choice which process performs the next step. Besides interleaving, the nondeterminism in an MDP can also be useful for abstraction purposes, for underspecification to be resolved in future refinement steps, or for modeling the interface with an unknown environment. Formally, an MDP is a tuple $\mathcal{M} = (S, \text{Act}, P, s_0, \dots)$ where

- S is a finite nonempty set of states,
- Act is a finite nonempty set of actions,
- $P : S \times \text{Act} \times S \rightarrow [0, 1]$ is a function, called *transition probability function*, such that for all states $s \in S$ and actions $\alpha \in \text{Act}$, the function $s \mapsto P(s, \alpha, \cdot)$ is either the null-function or a probabilistic distribution, i.e.,

$$\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\} \text{ for all states } s \in S \text{ and actions } \alpha \in \text{Act},$$

- $s_0 \in S$ is the initial state.

Alternatively, one can deal with a distribution over initial states. Further components can be added, such as reward or cost functions or atomic propositions. MDPs with a rewards for the states and actions can be useful to model sojourn times in states or other quantitative measures such as energy consumption. Atomic propositions can serve as state predicates and are often used to formalize properties in some temporal or modal logic.

If $s \in S$ then $\text{Act}(s)$ denotes the set of actions that are *enabled* in state s , i.e.,

$$\text{Act}(s) \stackrel{\text{def}}{=} \{\alpha \in \text{Act} : P(s, \alpha, s') > 0 \text{ for some } s' \in S\}.$$

For technical reasons, it is often useful to assume that there are no terminal states, i.e., for each state $s \in S$ the set $\text{Act}(s)$ is nonempty.

The intuitive operational behavior of an MDP \mathcal{M} as above is the following. The computation starts in the initial state s_0 . If after n steps the current state is s_n then first an enabled action $\alpha_{n+1} \in \text{Act}(s_n)$ is chosen nondeterministically. The effect of action α_{n+1} in state s_n is given by the distribution $P(s_n, \alpha_{n+1}, \cdot)$. Thus, the next state s_{n+1} belongs to the support of $P(s_n, \alpha_{n+1}, \cdot)$ and is chosen probabilistically. The resulting sequence of states $\pi = s_0 s_1 s_2 \dots \in S^\omega$ is called a path of \mathcal{M} .

Markov decision processes are widely used as an operational model for parallel systems where some components behave probabilistically, e.g., if they rely on a randomized algorithms or communicate via an unreliable fifo channel that loses or corrupts messages with some small (fixed) probability.

Example 1 (MDP for a randomized mutual exclusion protocol). Figure 1 shows an MDP modeling a simple mutual exclusion protocol for two processes P_1 , P_2 that compete to access a certain shared resource. Process P_i is represented by three local states n_i, w_i, c_i . Local state n_i stands for the non-critical phase of process P_i , w_i represents the location where process P_i is waiting and c_i denotes the critical section. The local states n_i and c_i can be left by performing a request or release action, respectively. If P_i is waiting, i.e., its current local state is w_i , and the other process P_j is not its critical section then P_i can enter the local state c_i (action $enter_i$). If both P_1 and P_2 are waiting then the competition is resolved by a randomized arbiter who tosses a fair coin to decide whether P_1 will enter its critical section (if the outcome is head) or P_2 gets the grant (if the outcome is tail). All other actions $request_i$, $release_i$, and $enter_i$ have a deterministic effect in the sense that, in all states s where they are enabled, the associated distribution is a Dirac distribution, i.e., assigns probability 1 to the unique successor state.

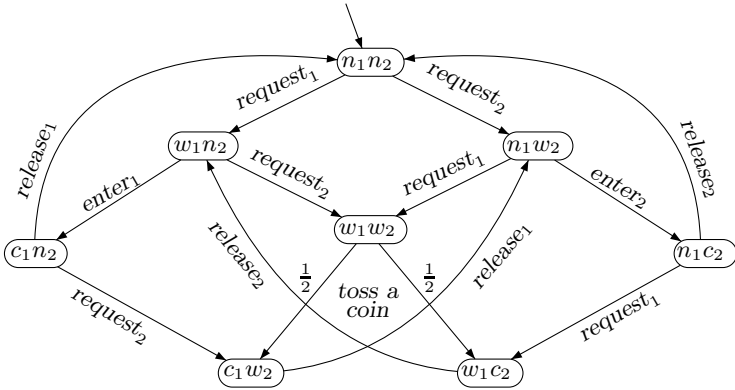


Fig. 1. MDP for mutual exclusion with randomized arbiter

Schedulers. Reasoning about the probabilities for path events (i.e., conditions that might or might not hold for a path) in an MDP requires the concept of *schedulers*, also often called *policies* or *adversaries*. A scheduler refers to any, possibly history-dependent, strategy for resolving the nondeterministic choices. Formally, a scheduler for an MDP $\mathcal{M} = (S, \text{Act}, \text{P}, s_0, \dots)$ can be defined as a function

$$D : S^+ \rightarrow \text{Act} \text{ such that } D(s_0 s_1 \dots s_n) \in \text{Act}(s_n).$$

The input $s_0 s_1 \dots s_n$ of D stands for the “history”, i.e., the sequence of states that have been visited in the past. The last state s_n represents the current state s_n . (The values of D are only relevant for finite D -paths, i.e., sequences $s_0 s_1 \dots s_n$ such that $\text{P}(s_i, D(s_0, \dots, s_n), s_{i+1}) > 0$ for $0 \leq i < n$.)

In the literature, more general types of schedulers have been defined, e.g., randomized schedulers that assign distributions of actions to finite paths rather

than single actions. Furthermore, the input of a scheduler can also include the actions that have been taken in the past. Vice versa, one can also restrict the class of schedulers to those that are realizable by a finite-state machine. For reasoning about probabilities for ω -regular path events in worst-case scenarios, these differences in the definition of schedulers are irrelevant, as long as they rely on complete information about the states of \mathcal{M} .

Given a scheduler D , the operational behavior of \mathcal{M} under D can be unfolded into a tree-like infinite Markov chain. This allows to apply standard techniques for Markov chains to define a σ -algebra over infinite paths and the probability of path events, i.e., measurable sets of infinite paths. Details can be found in any textbook on Markov decision processes, see e.g. [27].

Quantitative worst-case analysis. The typical task of the *quantitative analysis* of an MDP is to determine the probabilities for a certain path event E in a worst-case scenario, i.e., the maximal or minimal probabilities for E when ranging over all schedulers. Thus, the purpose of a quantitative analysis is to provide lower or upper probability bounds that are guaranteed for all interleavings of concurrent activities, all refinements of nondeterministic choices that have been obtained from abstraction techniques, and no matter how the environment behaves, provided that the environment has been modeled nondeterministically. When Markov decision processes are augmented with cost functions, then the quantitative analysis can also establish lower or upper bounds on expected values (e.g., costs for reaching a certain goal set or long-run averages) that can be guaranteed for all schedulers. The notion *qualitative analysis* refers to the task where one has to check whether a given event E holds almost surely, i.e., with probability 1 for all schedulers, or whether E holds with zero probability, no matter which scheduler is used.

For an example, let us regard the mutual exclusion protocol modeled by the MDP shown in Figure 11 again.

- The safety property E_{safe} stating that the two processes are never simultaneously in their critical section needs no probabilistic features and can be established by standard (non-probabilistic) model checking techniques, as the mutual exclusion property holds along all paths. Note that the state $\langle c_1, c_2 \rangle$ is not reachable.
- The liveness property E_{live} stating that each waiting process will eventually enter its critical section does not hold along all paths. E.g., in any infinite path that runs forever through the cycle $\langle n_1, w_2 \rangle \langle w_1, w_2 \rangle \langle c_1, w_2 \rangle \langle n_1, w_2 \rangle$ the second process is waiting forever. However, such paths have probability measure 0 under all schedulers. Hence, event E_{live} holds almost surely (i.e., with probability 1) under each scheduler. This yields that the minimal probability for E_{live} is 1.
- Suppose now that process P_2 is waiting in the current state, i.e., we treat state $\langle n_1, w_2 \rangle$ as initial state.

For each scheduler, the probability that P_2 will enter its critical section within the next n rounds is at least $1 - \frac{1}{2^n}$. Here, by a “round” we mean

any simple cycle containing the state $\langle w_1, w_2 \rangle$ where the randomized arbiter tosses a coin. The worst case scenario for process P_2 is a scheduler that always schedules action $request_1$ in state $\langle n_1, w_2 \rangle$. Under this scheduler, process P_1 is the winner of the first n coin tossing experiment with probability $\frac{1}{2^n}$. It should be noticed that under other schedulers, process P_2 can have better chances to enter its critical section within the next n rounds. For example, if action $enter_2$ is scheduled in state $\langle n_1, w_2 \rangle$ then process P_2 will enter its critical section in the first round.

The expected number of rounds that P_2 has to wait after having requested its critical section is less or equal

$$\sum_{n=1}^{\infty} n \cdot \frac{1}{2^n} = 2$$

for each scheduler. Value 2 is obtained under the scheduler which always schedules the action $request_1$ in state $\langle n_1, w_2 \rangle$. Hence, the MDP in Figure 1 enjoys the property that the minimal expected number of rounds that P_2 has to wait before entering its critical section is 2.

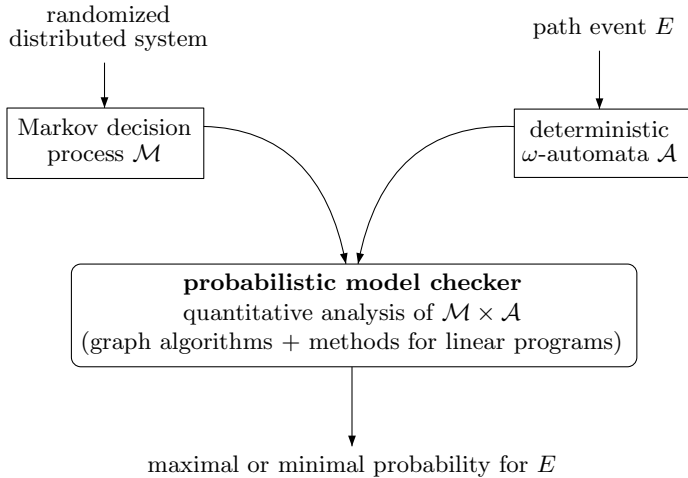


Fig. 2. Schema for the quantitative analysis

Probabilistic model checking. Probabilistic model checking techniques for finite-state Markov decision processes have been designed for verifying qualitative and quantitative properties specified in probabilistic computation tree logic [8,15,17] or computing extremal probabilities for ω -regular path events [32,33,13,5]. The schema of the standard model checking approach for computing extremal (i.e., minimal or maximal) probabilities for ω -regular path events in MDPs is shown in Figure 2. The main idea relies on an analysis of the product that results from the given MDP \mathcal{M} with a deterministic ω -automaton \mathcal{A} representing the

path event. (See, e.g., [31][19] for an overview of automata over infinite structures.) As \mathcal{A} is deterministic, the product $\mathcal{M} \times \mathcal{A}$ can be understood as an MDP that behaves operationally as \mathcal{M} and additionally mimicks \mathcal{A} 's behavior when scanning a path of \mathcal{M} . The extremal probabilities for the given path event E in \mathcal{M} agree with the extremal probabilities for the acceptance condition of \mathcal{A} in the product-MDP \mathcal{M} . The latter can be computed by means of a graph analysis and linear programming techniques for calculating minimal or maximal reachability probabilities. The details of this procedure can be found in the above mentioned literature.

Several efficient probabilistic model checking tools for MDPs are available that have been used in many application areas. The most prominent one that uses a symbolic approach with BDD-variants is PRISM [20].

Anomalies when ranging over all schedulers. The approach sketched above computes “worst-case” probabilities for path events when ranging over all schedulers. In particular, all possible interleavings and resolutions of other nondeterministic choices in the system model are taken into account. However, the computed worst-case probabilities are often too pessimistic since there are unrealistic schedulers that might yield extremal probabilities.

As in the non-probabilistic case, *fairness assumption* might be necessary to rule out schedulers that treat some processes in an unfair way. E.g., for the MDP in Figure 1, the scheduler that never takes an action of process P_2 and only schedules the actions *request*₁, *enter*₁ and *release*₁ can be viewed to be unrealistic since the request-operation of process P_2 is always enabled. The schema sketched in Figure 2 can be refined to compute the minimal and maximal probabilities of ω -regular path events when ranging over fair schedulers only [7][6]. The major difference is that the graph-based analysis has to be revised.

But there are still other curious phenomena when ranging over all (possibly fair) schedulers in MDPs that can contort the extremal probabilities.

Example 2. The MDP in Figure 3 arises through the parallel composition of two processes P_1 and P_2 with local integer variables x and y , respectively, that have initial value 0.

Process P_1 consists of a nondeterministic choice between actions β and γ , representing the deterministic assignments $x := 1$ (action β) and $x := 2$ (action γ). Process P_2 probabilistically assigns value 1 or 2 to y , depending on the outcome of a coin tossing experiment (action α). To ensure that no state is terminal, self-loops can be added to all states that have no outgoing transition in Figure 3. These self-loops might represent an internal step performed by a third process.

The maximal probability that a state where $x = y \in \{1, 2\}$ will be reached is 1, since we might first schedule α and then, depending on the outcome of α , we can choose β or γ to ensure that finally $x = y \in \{1, 2\}$. This scheduler, however, is not realizable if there is no central control and process P_1 cannot access the current value of P_2 's local variable y . Indeed, intuitively we might expect the answer $\frac{1}{2}$ (rather than 1) as the maximal probability for reaching a

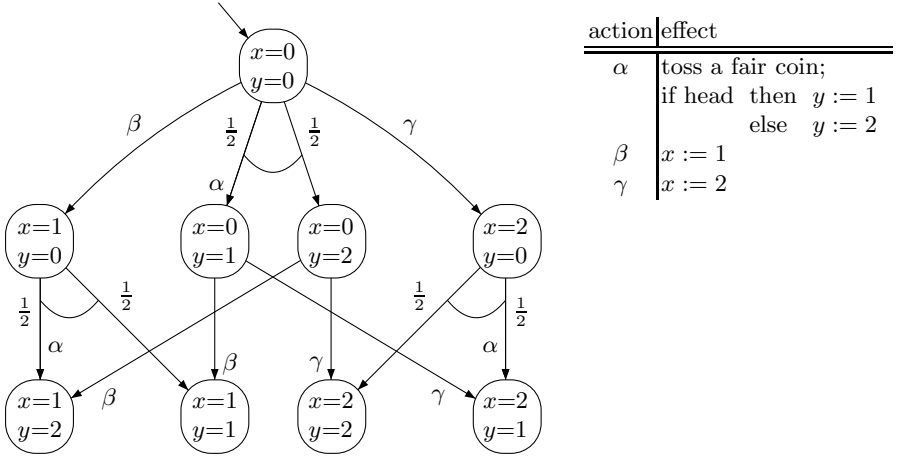


Fig. 3. MDP for $P_1 \parallel P_2$ where $P_1 = \beta + \gamma$ and $P_2 = \alpha$

state where $x = y \in \{1, 2\}$ is $\frac{1}{2}$ when ranging over all strategies to resolve the nondeterministic choice between β and γ in process P_1 .

Partially-observable Markov decision processes. The above observation motivates to study the worst-case behavior of MDPs for restricted classes of schedulers that take the local view of the processes that run in parallel into account. In the literature, several classes of schedulers have been considered that are more adequate for reasoning about distributed systems [12, 18, 1]. Partially-observable MDPs, briefly called POMDPs, can be seen as a simple variant that can serve to model the view of one process.

POMDPs are formally defined as MDPs that are augmented by an equivalence relation \sim on the state space which identifies those states that cannot be distinguished by an observer. A scheduler $D : S^+ \rightarrow \text{Act}$ for an POMDP is called *observation-based* iff D 's decisions only depend on the observables of the history, i.e., if $\pi_1 = s_0 s_1 \dots s_n$ and $\pi_2 = t_0 t_1 \dots t_n$ are finite paths of the same length such that $[s_i] = [t_i]$ for $0 \leq i \leq n$ then $D(\pi_1) = D(\pi_2)$. Here, $[s] = \{s' \in S : s \sim s'\}$ denotes the \sim -equivalence class of state s .

POMDPs are used in many application areas, see e.g. [9], and many algorithms have been proposed for the analysis of the behavior up a fixed number of steps ("finite-horizon") [23, 25, 21, 24]. However, many difficulties arise for path events of the standard safety-liveness spectrum where no restrictions are imposed on the number of relevant steps. The design of algorithms for a quantitative analysis that determines worst-case probabilities for, e.g., a reachability condition, under all observation-based schedulers is impossible. This is due to the close link between POMDPs with a reachability objective, say $\diamond F$ where F is a set of states and \diamond denotes the eventually operator of LTL, and *probabilistic finite automata* (PFA) [28, 26]. Note that in the extreme case of an POMDP \mathcal{M} where

\sim identifies all states, the observation-based schedulers can be viewed as functions $D : \mathbb{N} \rightarrow \text{Act}$, and hence, as infinite words in Act^ω . But then the question whether there exists an observation-based scheduler D for \mathcal{M} such that the probability for $\diamond F$ under D is larger than a given a threshold $\lambda \in]0, 1[$ is equivalent to the *non-emptiness problem* for the PFA that results from \mathcal{M} by treating F as the set of final states and λ as threshold for the accepted language; and the latter is known to be undecidable [26,22].

There is even no approximation algorithm for maximal reachability probabilities in POMDPs and the verification problem for almost all interesting quantitative properties for POMDPs and related models are undecidable [22,18]. Even worse, undecidability results can be established for certain instances of the model checking problem for POMDPs and qualitative properties, such as the question whether there exists an observation-based scheduler such that a repeated reachability condition $\square \diamond F$ (“visit infinitely often some state in F ”) holds with positive probability. This problem is a generalization of the non-emptiness problem for *probabilistic Büchi automata* (PBA) with the probable semantics [3] which is known to be undecidable [2].

However, some qualitative model checking problems for POMDPs are decidable. Examples for decidable verification problems for POMDPs are the question whether there exists an observation-based scheduler such that an invariance $\square F$ (“always F ”) holds with positive probability [16], or whether the maximal probability under all observation-based schedulers for a reachability condition $\diamond F$ (“eventually F ”) or a repeated reachability condition $\square \diamond F$ (“infinitely often F ”) is 1 [2], see also [10,11]. The algorithms for such qualitative model checking problems for POMDPs rely on variants of the powerset construction that has been introduced for incomplete-information games [29].

Besides the observation that ranging over the full class of schedulers can yield too pessimistic worst-case probabilities, also several other techniques suffer from the power of general schedulers.

In the context of *partial order reduction* for MDPs, it has been noticed in [4,14] that the criteria that are known to be sound for non-probabilistic systems are not sufficient to preserve extremal probabilities for ω -regular path events. In this setting, the problem is that the commutativity of independent probabilistic actions is a local property that does not carry over to a global setting. Consider again the MDP in Example 2. Action α is independent from both β and γ , as α accesses just variable y , while β and γ operate on x , without any reference to y . Indeed if we consider the parallel execution of α and β (or α and γ) in isolation, then the order of α and β (or α and γ) is irrelevant for the probabilities of the final outcome. But the commutativity of α and β resp. α and γ does not carry over to the nondeterministic choice between β and γ (process P_1).

- The scheduler which first chooses α and then β if $x=0 \wedge y = 1$ and γ if $x=0 \wedge y = 2$ yields probability 1 for a final outcome where $x = y \in \{1, 2\}$ hold.

- For the schedulers that first choose one of the actions β or γ and then perform α , an outcome where $x = y \in \{1, 2\}$ hold is obtained with probability $\frac{1}{2}$.

This observation causes some care for the design of partial order reduction techniques for MDPs and either requires an extra condition [414] or to restrict the class of schedulers for the worst-case analysis [17].

Another problem that arises from the power of the full class of schedulers is the lack of compositionality of trace distribution equivalence in MDP-like models [30]. This problem can be avoided by introducing some appropriate concept of distributed scheduling [12]. But as the series of undecidability results for POMDPs shows, the price one has to pay when switching from the full class of schedulers to more realistic ones is the loss of model checking algorithms for the quantitative analysis against infinite-horizon path events, and partly also verification algorithms for qualitative properties.

Nevertheless, further restrictions on the scheduler types might be possible to overcome the limitations due to undecidability results.

References

1. Andres, M., Palamidessi, C., van Rossum, P., Sokolova, A.: Information hiding in probabilistic concurrent systems. In: Proc. of the 7th International Conference on Quantitative Evaluation of SysTems (QEST 2010). IEEE Computer Society Press, Los Alamitos (to appear 2010)
2. Baier, C., Bertrand, N., Grösser, M.: On decision problems for probabilistic Büchi automata. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 287–301. Springer, Heidelberg (2008)
3. Baier, C., Grösser, M.: Recognizing ω -regular languages with probabilistic automata. In: Proc. of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005), pp. 137–146. IEEE Computer Society Press, Los Alamitos (2005)
4. Baier, C., Grösser, M., Ciesinski, F.: Partial order reduction for probabilistic systems. In: Proc. of the First International Conference on Quantitative Evaluation of SysTems (QEST), pp. 230–239. IEEE Computer Society Press, Los Alamitos (2004)
5. Baier, C., Größer, M., Ciesinski, F.: Model checking linear time properties of probabilistic systems. In: Handbook of Weighted Automata, pp. 519–570 (2009)
6. Baier, C., Größer, M., Ciesinski, F.: Quantitative analysis under fairness constraints. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 135–150. Springer, Heidelberg (2009)
7. Baier, C., Kwiatkowska, M.: Model checking for a probabilistic branching time logic with fairness. Distributed Computing 11 (1998)
8. Bianco, A., de Alfaro, L.: Model checking of probabilistic and non-deterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
9. Cassandra, A.R.: A survey of POMDP applications. Presented at the AAAI Fall Symposium (1998), <http://pomdp.org/pomdp/papers/applications.pdf>
10. Chadha, R., Sistla, P., Viswanathan, M.: Power of randomization in automata on infinite strings. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 229–243. Springer, Heidelberg (2009)

11. Chatterjee, K., Doyen, L., Henzinger, T.: Qualitative analysis of partially-observable markov decision processes. In: Proc. Mathematical Foundation of Computer Science. LNCS, Springer, Heidelberg (2010)
12. Cheung, L., Lynch, N., Segala, R., Vaandrager, F.: Switched PIOA: Parallel composition via distributed scheduling. *Theoretical Computer Science* 365(1-2), 83–108 (2006)
13. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. *Journal of the ACM* 42(4), 857–907 (1995)
14. D’Argenio, P.R., Niebert, P.: Partial order reduction on concurrent probabilistic programs. In: Proc. of the First International Conference on Quantitative Evaluation of SysTems (QEST), pp. 240–249. IEEE Computer Society Press, Los Alamitos (2004)
15. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University, Department of Computer Science (1997)
16. de Alfaro, L.: The verification of probabilistic systems under memoryless partial-information policies is hard. In: Proc. of the 2nd International Workshop on Probabilistic Methods in Verification (ProbMiV 1999), pp. 19–32. Birmingham University (1999), Research Report CSR-99-9
17. Giro, S., D’Argenio, P., Maria Ferrer Fioriti, L.: Partial order reduction for probabilistic systems: A revision for distributed schedulers. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 338–353. Springer, Heidelberg (2009)
18. Giro, S., D’Argenio, P.R.: Quantitative model checking revisited: neither decidable nor approximable. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 179–194. Springer, Heidelberg (2007)
19. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002)
20. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)* 6(2), 128–142 (2004)
21. Littman, M.: Algorithms for Sequential Decision Making. PhD thesis, Brown University, Department of Computer Science (1996)
22. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence* 147(1-2), 5–34 (2003)
23. Monahan, G.: A survey of partially observable Markov decision processes: Theory, models and algorithms. *Management Science* 28(1), 1–16 (1982)
24. Mundhenk, M., Goldsmith, J., Lusena, C., Allender, E.: Complexity of finite-horizon Markov decision process problems. *Journal of the ACM* 47(4), 681–720 (2000)
25. Papadimitriou, C., Tsitsiklis, J.: The complexity of Markov decision processes. *Mathematics of Operations Research* 12(3) (1987)
26. Paz, A.: Introduction to probabilistic automata. Academic Press Inc., London (1971)
27. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons, Chichester (1994)
28. Rabin, M.O.: Probabilistic automata. *Information and Control* 6(3), 230–245 (1963)

29. Reif, J.H.: The complexity of two-player games of incomplete information. *Journal of Computer System Sciences* 29(2), 274–301 (1984)
30. Segala, R.: *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Massachusetts Institute of Technology (1995)
31. Thomas, W.: Languages, automata and logic. In: Rozenberg, G., Salomaa, A. (eds.) *Handbook of Formal Languages*, vol. 3, pp. 389–455. Springer, New York (1997)
32. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: *Proc. of the 26th Symposium on Foundations of Computer Science (FOCS)*, pp. 327–338. IEEE Computer Society Press, Los Alamitos (1985)
33. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: *Proc. of the 1st IEEE Symposium on Logic in Computer Science (LICS)*, pp. 332–345. IEEE Computer Society Press, Los Alamitos (1986)

Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems

John Fitzgerald¹, Peter Gorm Larsen², Ken Pierce¹,
Marcel Verhoef³, and Sune Wolff^{2,4}

¹ Newcastle University, UK

{John.Fitzgerald,K.G.Pierce}@ncl.ac.uk

² Aarhus School of Engineering, Denmark

{pgl, swo}@iha.dk

³ Chess, Haarlem, The Netherlands

Marcel.Verhoef@chess.nl

⁴ Terma A/S, Denmark

sw@terma.dk

Abstract. This paper presents initial results of research aimed at developing methods and tools for multidisciplinary collaborative development of dependable embedded systems. We focus on the construction and analysis by co-simulation of formal models that combine discrete-event specifications of computer-based controllers with continuous-time models of the environment with which they interact. Basic concepts of collaborative modelling and co-simulation are presented. A pragmatic realisation using the VDM and Bond Graph formalisms is described and illustrated by means of an example, which includes the modelling of both normal and faulty behaviour. Consideration of a larger-scale example from the personal transportation domain suggests the forms of support needed to explore the design space of collaborative models. Based on experience so far, challenges for future research in this area are identified.

1 Introduction

Whether viewed from a technical or a commercial perspective, the development of embedded systems is a demanding discipline. Technical challenges arise from the need to develop complex, software-rich products that take the constraints of the physical world into account. Commercial pressures include the need to innovate rapidly in a highly competitive market and to offer products that are simultaneously resilient to faults and highly efficient.

Traditional development approaches are mono-disciplinary in style, in that separate mechanical, electronic and software engineering groups handle distinct aspects of product development and often do so in sequence. Contemporary concurrent engineering strategies aim to improve the time to market by performing these activities in parallel. However, cross-cutting system-level requirements that cannot be assigned to a single discipline, such as performance and dependability, can cause great problems, because their impact on each discipline is exposed late in the development process, usually during integration. Embedded systems, in which the viability of the product depends on

the close coupling between the physical and computing disciplines, therefore calls for a more multidisciplinary approach.

High-tech mechatronic systems are complex (a high-volume printer typically consists of tens of thousands of components and millions of lines of code) and so is the associated design process. There are many design alternatives to consider but the impact of each design decision is difficult to assess. This makes the early design process error-prone and vulnerable to failure as downstream implementation choices may be based on it, causing a cascade of potential problems. Verhoef identifies four causes of this problem [24]. First, the disciplines involved have distinct methods, languages and tools. The need to mediate and translate between them can hamper development by introducing opportunities for imprecision and misunderstanding. The inconsistencies that result are difficult to detect because there is usually no structured process for analysing system-level properties. Second, many design choices are made implicitly, based on previous experience, intuition or assumptions, and their rationale is not recorded. This can lead to locally optimal but globally sub-optimal designs. Third, dynamic aspects of a system are complex to grasp and there are few methods and tools available to support reasoning about time varying aspects in design, in contrast to static or steady-state aspects. Fourth, embedded systems are often applied in areas where dependability is crucial, but design complexity is compounded by the need to take account of faults in equipment or deviation by the environment (plant or user) from expected norms. Non-functional properties associated with dependability can be hard to capture and assess. This typically leads to designs that are over-dimensioned, making implementation impractical due to the associated high costs.

A strong engineering methodology for embedded systems will be *collaborative*¹. It will provide notations that expose the impact of design choices early, allow modelling and analysis of dynamic aspects and support systematic analysis of faulty as well as normal behaviour. The hypothesis underpinning our work is that lightweight formal and domain-specific models that capture system-level behaviour can supply many of these characteristics, provided they can be combined in a suitable way and be evaluated rapidly. Formal techniques of system modelling and analysis allow the precise modelling of desired behaviour upstream of expensive commitments to hardware and code. The support for abstraction in formal modelling languages allows the staged introduction of additional sources of complex behaviour, such as fault modelling. A recent review of the use of formal methods [27] suggests that successful industry applications often make use of tools that offer analysis with a high degree of automation, are lightweight in that they are targeted at particular system aspects, are robust and are readily integrated with existing development practices.

We conjecture that a collaborative methodology based on lightweight formal modelling improves the chances of closing the design loop early, encouraging dialogue between disciplines and reducing errors, saving cost and time. Throughout the paper, we term this approach “collaborative modelling” or “co-modelling”. In previous work [24],

¹ Collaboration is “United labour, co-operation; esp. in literary, artistic, or scientific work.” [23]. Simple coordination between disciplines, for example, bringing mechatronic and software engineers together in the same space, or enabling communication between them, is necessary but not sufficient to achieve collaboration.

Verhoef has demonstrated that significant gains are feasible by combining VDM and Bond Graphs, using co-simulation as the means of model assessment. Andrews et al. have suggested that collaborative models are suited to exploring fault behaviours [1]. In this paper, we build upon these results, indicating how a collaborative modelling approach can be realised in existing formally-based technology, how models can be extended to describe forms of faulty behaviour, and identifying requirements for design space exploration in this context.

In Section 2 we introduce concepts underpinning co-modelling and co-simulation. Section 3 shows how we have so far sought to realise these ideas using the VDM discrete event formalism and the Bond Graph continuous-time modelling framework. We outline some of the design decisions that face collaborative teams in the area of fault modelling in particular. Section 4 looks towards the application of co-simulation in exploring the design space for larger products. Section 5 identifies research challenges that spring from our experience so far.

2 Collaborative Modelling and Design Space Exploration

In our approach to collaborative development, a *model* is a more or less abstract representation of a system or component of interest. We regard a model as being *competent* for a given analysis if it contains sufficient detail to permit that analysis. We are primarily concerned with analysis by execution, so we will generally be interested in formal models that, while they are abstract, are also directly executable. A test run of a model is called a *simulation*. A *design parameter* is a property of a model that affects its behaviour, but which remains constant during a given simulation. A simulation will normally be under the control of a *script* that determines the initial values of modelled state variables and the order in which subsequent events occur. A script may force the selection of alternatives where the model is underspecified and may supply external inputs (e.g. a change of set point) where required. A *test result* is the outcome of a simulation over a model.

A goal of our work is to support the modelling of faults and resilience mechanisms. Adopting the terminology of Avizienis et al. [2], we regard a *fault* as the cause of an *error* which is part of the system state that may lead to a *failure* in which a system's delivered service deviates from specification. *Fault modelling* is the act of extending the model to encompass faulty behaviours. *Fault injection* is the act of triggering faulty behaviour during simulation and is the responsibility of a script.

A *co-model* (Figure 1(a)) is a model composed of:

- Two component models, normally one describing a computing subsystem and one describing the plant or environment with which it interacts. The former model is typically expressed in a discrete event (DE) formalism and the latter using a continuous-time (CT) formalism.
- A *contract*, which identifies shared design parameters, shared variables, and common events used to effect communication between the subsystems represented by the models.

A co-model is itself a model and may be simulated under the control of a script. The simulation of a co-model is termed *co-simulation*. A co-model offers an interface that

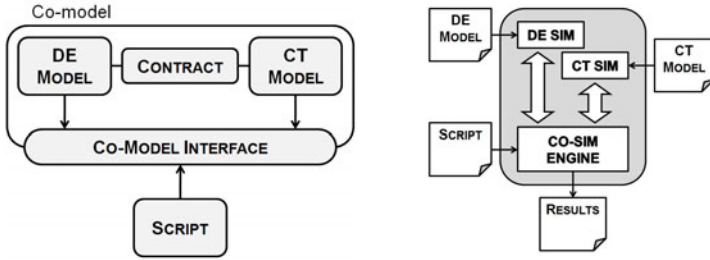


Fig. 1. (a) conceptual view of a co-model (left) and (b) execution of a co-model realised using a co-simulation engine (right)

can be used to set design parameters and to run scripts to set initial values, trigger faulty behaviour, provide external inputs and observe selected values as the simulation progresses. Our goal is to provide modelling and simulation techniques that support *design space exploration*, by which we mean the (iterative) process of constructing co-models, co-simulation and interpretation of test results governing the selection of alternative models and co-models as the basis for further design steps.

In a co-simulation, a *shared variable* is a variable that appears in and can be accessed from both component models. Predicates over the variables in the component models may be stated and may change value as the co-simulation progresses. The changing of the logical value of a predicate at a certain time is termed an *event*. Events are referred to by name and can be propagated from one component model to another within a co-model during co-simulation. The semantics of a co-simulation is defined in terms of the evolution of these shared variable changes and event occurrences while co-model time is passing. In a co-simulation, the CT and DE models execute as interleaved threads of control in their respective simulators under the supervision of a *co-simulation engine* (Figure 1(b)). The DE simulator calculates the smallest time Δt it can run before it can perform the next possible action. This time step is used by the co-simulation engine in the communication to the CT simulator which then runs the solver forward by up to Δt . If the CT simulator observes an event, for example when a continuously varying value passes a threshold, this is communicated back to the DE simulator by the co-simulation engine. If this event occurred prior to Δt , then the DE simulator does not complete the full time step, but it runs forward to this shorter time step and then re-evaluates its simulator state. Note that it is not possible (in general) to roll the DE simulation back, owing to the expense of saving the full state history, whereas the CT solver can work to specified times analytically. Verhoef et al. [25] provide an integrated operational semantics for the co-simulation of DE models with CT models. Co-simulation soundness is ensured by enforcing strict monotonically increasing model time and a transaction mechanism that manages time triggered modification of shared variables.

3 Co-modelling and Co-simulation in 20-Sim and VDM

The work reported in this paper is aimed at demonstrating the feasibility of multidisciplinary collaborative modelling for early-stage design space exploration. As a proof

of concept, methods and an open tools platform are being developed to support modelling and co-simulation, with explicit modelling of faults and fault-tolerance mechanisms from the outset. This activity is undertaken as part of the EU FP7 Project DESTECs [4]².

The proof of concept work uses continuous-time models expressed as differential equations in Bond Graphs [17] and discrete event models expressed using the Vienna Development Method (VDM) [16][10] notation. The simulation engines supporting the two notations are, respectively, 20-sim [6]³ and Overture [18]⁴. Complementary work investigates the extension of the co-simulation approach to other modelling languages [26]. An open, extensible tools platform will be developed, populated with plugins to support static analysis, co-simulation, testing and fault analysis. Trials will be conducted on industrial case studies from several domains, including document handling, heavy equipment and personal transportation. Aspects of the latter study are introduced in Section 4. In this section we first introduce the two modelling notations and their tools (Sections 3.1-3.2) before discussing a simple example of co-simulation (Section 3.3) and fault modelling (Section 3.4).

3.1 VDM

VDM is a model-oriented formal method that permits the description of functionality at a high level of abstraction. The base modelling language, VDM-SL, has been standardised by ISO [15]. Extensions have been defined for object-orientation (VDM++ [11]) and real-time embedded and distributed systems (VDM-RT [19]). VDM-RT includes primitives for modelling deployment to a distributed hardware architecture and support for asynchronous communication.

VDM is supported by industrial strength tools: VDMTools [8][12] and the open source Overture tool [18] (being developed with the Eclipse Platform). Both tools have been extended with the capability to generate logfiles derived from execution of VDM-RT models [9][19].

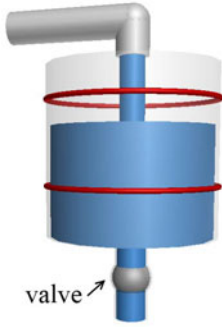
3.2 20-Sim

20-sim [6], formerly CAMAS [5], is a tool for modelling and simulation of dynamic systems including electronics, mechanical and hydraulic systems. All models are based on Bond Graphs [17] which is a non-causal technology, where the underlying equations are specified as equalities. Hence variables do not initially need to be specified as inputs or outputs. In addition, the interface between Bond Graph elements is port-based where each port has two variables that are computed in opposite directions, for example voltage and current in the electrical domain. 20-sim also supports graphical representation of the mathematical relations between signals in the form of block diagrams and iconic diagrams (building blocks of physical systems like masses and springs) as more user friendly notations. Combination of notations is also possible, since Bond Graphs provide a common basis. It is possible to create sub-models of multiple components or even multiple sub-models allowing for a hierarchical model structure.

² <http://www.destecs.org/>

³ <http://www.20sim.com/>

⁴ <http://www.overturetool.org/>



$$\frac{dV}{dt} = \varphi_{in} - \varphi_{out} \quad (1)$$

$$\varphi_{out} = \begin{cases} \frac{\rho * g}{A * R} * V & \text{if valve open} \\ 0 & \text{if valve closed} \end{cases} \quad (2)$$

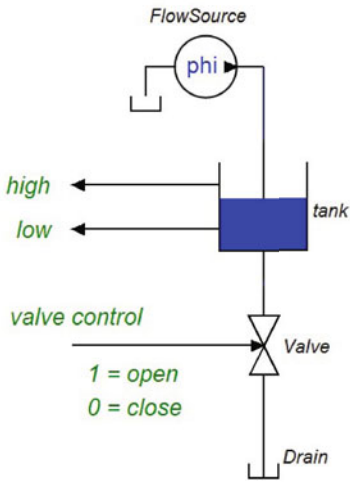
Fig. 2. Water tank level controller case study system overview

3.3 Basic Co-simulation in 20-Sim and VDM

In this section, co-simulation between a VDM and 20-sim model is illustrated by means of a simple example based on the level controller of a water tank (Figure 2). The tank is continuously filled by the input flow φ_{in} , and can be drained by opening the valve, resulting in the output flow φ_{out} . The output flow through the valve when this is opened or closed is described by Equation 2 in Figure 2, where ρ is the density of the water, g is acceleration due to gravity, A is the surface area of the water tank, R is the resistance in the valve and V is the volume. An iconic diagram model of this system created in 20-sim is shown in Figure 3(a). There are two simple requirements for the discrete-event controller: when the water reaches the “high” level mark the valve must be opened, and when the water reaches the “low” level mark, the valve must be closed. A VDM model of the controller is in Figure 3(b).

The controller model is expressed in VDM-RT. An instance variable represents the state of the valve and the asynchronous `Open` and `Close` operations set its value. Both operations are specified explicitly in the sense that they are directly executable. In order to illustrate the recording of timing constraints in VDM-RT, the **duration** and **cycles** statements constrain the time taken by the operations to 50ms in the case of `Open` and 1000 processor cycles in the case of `Close`. The time taken for a `Close` operation is therefore dependent on the defined speed of the computation unit (CPU) on which it is deployed (described elsewhere in the model). The synchronisation constraints state that the two operations are mutually exclusive.

A co-model can be constructed consisting of the 20-sim model and VDM model shown above. The co-simulation contract between them identifies the events from the CT model that are coupled to the operations in the DE model and indicates that `valve` is shared between the two models. The contract indicates which state event triggers which operations. In the case the water level rises above the upper sensor, the `Open` operation shall be triggered and respectively when the water level drops below the lower sensor, the `Close` operation shall be called. Note that `valve` represents the actual state of the valve, not merely the controller’s view of it. These facets are explored in Section 3.4 below.



```

class Controller

instance variables
  private i : Interface

operations
  async public Open:() ==> ()
  Open() == duration(50)
  i.SetValue(true);

  async public Close:() ==> ()
  Close() == cycles(1000)
  i.SetValue(false);

sync
  mutex(Open, Close);
  mutex(Open); mutex(Close)

end Controller

```

Fig. 3. (a) 20-Sim model (left) and (b) event-driven controller in VDM (right)

3.4 Modelling of Faults

The water tank case study has so far considered only the nominal behaviour of the environment and controller. This section considers the types of faults that one might wish to explore using co-simulation and how they might be modelled, taking the water tank as an example. The choice of which faults to model depends on the purpose of the overall modelling process. For example, where the purpose is to identify mechanisms for recovering from or tolerating faults that lead to significant system-level failures (those with unacceptable likelihood or severity), an analytic technique such as Fault Tree Analysis might be used to identify particular faults that lead to significant system failures. However faults of interest are identified, the model must be made competent to express them and to describe any recovery or tolerance measure to be put in place.

Sensor and actuator faults are particularly interesting because they typically cross the boundary between components of the co-model. In the water tank example, the sensors (high and low) and the actuator (valve) are not modelled as distinct units. Their connection is realised purely through the co-simulation framework and is essentially perfect — the controller is always notified of events and the valve always reacts correctly to the controller.

The designer faces a choice of alternative ways in which to represent sensor and actuator faults. One approach is to specify failures of communication in the co-simulation contract. This is somewhat disconnected from reality however, since a contract will not necessarily map directly to the physical sensors and actuators of the real system. In our current work, we advocate keeping the contract pure and instead introducing explicit

models of sensors and actuators to either the controller or plant model (or both where necessary). These models can then exhibit faulty behaviour as required. Since the controller and plant are modelled in far richer languages than the contract, this approach provides scope for describing complex faults. In addition, the capability to exhibit faults then exists statically in the controller and/or plant model and not simply dynamically as part of the co-simulation.

Bearing this in mind, we can introduce sensors and actuators into the model shown above. Let us first consider a faulty valve that can become *stuck*. That is, if the valve is open it will not close when commanded and if it is closed it will not open. We make the co-model competent to exhibit this fault by introducing a `ValveActuator` class into the VDM controller model, representing the actuator (Figure 4).

```

class ValveActuator

types
  ValveCommand = <OPEN> | <CLOSE>;

instance variables
  private i : Interface;
  private stuck : bool := false

operations
  public Command: ValveCommand ==> ()
  Command(c) == duration(50)
    if not stuck then
      cases c:
        <OPEN> -> i.SetValve(true),
        <CLOSE> -> i.SetValve(false)
      end
    pre not stuck
    post i.ReadValve() <=> c = <OPEN> and
      not i.ReadValve() <=> c = <CLOSE>
    errs STUCK : stuck -> i.ReadValve() = ~i.ReadValve();

  private SetStuckState: bool ==> ()
  SetStuckState(b) == stuck := b
  post stuck <=> b and not stuck <=> not b;

end ValveActuator

```

Fig. 4. Explicit model of a valve in VDM, which can exhibit a stuck fault

First, note the instance variable `stuck`. This represents an internal error state: when `stuck` is false, the valve actuator will behave correctly. An operation called `SetStuckState` is defined to control when the valve actuator becomes stuck. In this

model, no logic is included to control exactly when faults occur. A more complex fault model could include parameters to tune when faults occurred, for example, based on stochastic measures. Alternatively, the `SetStuckState` operation could be exposed to the co-simulation tool, which could then activate the fault during a co-simulation, in response to some scripted behaviour. Both methods have benefits and drawbacks, so it is suggested that the DESTTECS approach will allow both, leaving it up to the user to decide.

Second, consider the main operation of the class called `Command`. This operation should open and close the valve (using `i.SetValve`), depending on the command given. Note that in Figure 3(b), the controller was directly responsible for operating the valve. In this new model however, the controller must call the `Command` operation. The body of `Command` gives an explicit definition for the operation, which is reasonably intuitive — if the valve is not stuck, the valve will open if the command given was `<OPEN>` and close if the command was `<CLOSE>`. If the valve is stuck, nothing will happen.

In addition to the explicit definition, a precondition, postcondition and errors clause are also given. The precondition records assumptions about the state and input parameters when an operation is invoked. In this case, the precondition states that the operation will only behave correctly if the valve is not stuck. The postcondition captures the behaviour of the operation as a relation between the initial and final state. Postconditions must hold if the precondition holds. It is obvious however that the precondition may not be met, i.e. when the valve is stuck. In this case, it is not necessary for the operation to meet the postcondition and its behaviour is typically undefined. We can however introduce an errors clause to capture the behaviour when the valve is stuck. Here, the `errs` clause records that the valve's state will remain unchanged (note `~`-prefixed instance variable names indicate the initial value of the variable).

Another valve fault that might be considered is a leak. A simple model of a leaky valve would be one in which a constant amount of water flows out of the tank, even when the valve is closed. It is much more natural to model this fault in 20-sim, since it involves altering the flow rate (part of the plant model). Reproducing this on the VDM side would (at least) involve modifying the co-simulation contract to allow direct modification of the flow rate, which is an inelegant solution. Thus a modification of the 20-sim plant model is required. A diagram of the modified model is given in Figure 5 (based on Figure 3).

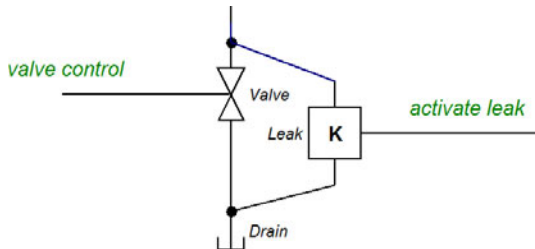


Fig. 5. Block diagram model of a valve which can exhibit a leak

The leak is modelled as an alternative route for water to flow from the tank to the drain, bypassing the valve. The rate of flow is a constant (K). As with the VDM model, the fault can be active or dormant (zero flow). In 20-sim however, activation of the fault is modelled as an input signal to the leaky component, in much the same way that the valve control activates the flow of water through the valve itself. As with the previous fault, this activation signal could be sent by the co-simulation framework executing a script to allow for fault injection. Using 20-sim to model this fault also allows for more complex models, for example a leak where the rate of flow depends on the pressure.

A scenario that could be explored with this leaky valve in the plant model would be to see whether or not the controller could discover that such a leak existed, based on the sensors with which it can observe the plant. Currently, the controller would only discover a potential leak if the low water sensor was activated while the valve was closed. Based on this discovery, we might suggest additional sensors that would allow the controller to discover a leak sooner.

4 Towards the Exploration of Design Alternatives

As indicated in Section 1, our practical approach to formal modelling of embedded systems must address the need to support the comparison and selection of design alternatives. Section 3.4 discussed some of the choices that developers can face in constructing a co-model for a simple control system that includes fault behaviour. In this section, we look towards the support for design space exploration based on the use of co-models, using a more substantial case study.

The ChessWay is an industrial challenge problem originated by Chess. It is a self-balancing personal transporter, much akin to the well-known Segway⁵. The device has two wheels, mounted on either side of a base platform on which the rider can stand, holding on to a handlebar (Figure 6). The systems weight is mostly positioned above the two powerful, direct drive wheels. As such, it acts like an inverted pendulum and is therefore unstable. In order to stop the ChessWay from falling over and perhaps injuring the rider, it must be actively balanced by driving the wheels. The aim of the system controller therefore is to keep the ChessWay upright, even while stationary. It can do this by applying torque to each wheel independently, such that the base of the ChessWay is always kept directly underneath the centre of gravity of the entire system. The rider can move forward (and backward) by leaning forward (or backward). The controller measures the deviation angle of the handlebar and performs an immediate control action in order to keep the ChessWay stable, similar to the way that you might try to balance a pencil on the tip of your finger.

The control laws for this kind of system are relatively simple, but the ChessWay remains a challenging control problem because the desired nominal system state is in fact metastable. Furthermore, the system dynamics require high frequency control in order to guarantee smooth and robust handling. Safety plays a crucial (complicating) role: there are circumstances in which the safest thing for the controller to do is to *allow* the ChessWay to fall over. For example, if the ChessWay is lying on the floor (90 degrees deviation from upright), then the controller needs a dangerously large torque

⁵ <http://www.segway.com/>



Fig. 6. The ChessWay personal transporter

to correct this. This would result in the handlebar swinging suddenly upright, possibly hitting the user. In fact, any sudden deviation exceeding 10 degrees from upright could result in similarly violent control correction subjected to the user. This obviously diminishes the driving experience, which should be smooth and predictable. Moreover, it is intuitively clear that even small failures of the hardware or software could easily lead to the ChessWay malfunctioning.

The challenge is to find a modelling methodology that allows the system developer to define a controller strategy (based on several possible user scenarios), while reasoning about the suitability of the system under possibly changing environmental conditions and in the presence of potential faults. The need for this methodology is easily demonstrated by the well-known public debate regarding the legality of allowing the Segway on public roads. For the device to become street legal, its usability had to be demonstrated to several third parties (such as government road safety inspectors) and to private insurance companies. This is of course a significant challenge and representative for many industrial products being developed today.

In the ChessWay case study, we should be able to specify multi-modal controller behaviour. For example, the controller should contain a start-up procedure in which the user must manually hold the ChessWay upright for a certain amount of time, before the controller will begin to balance the device actively. Similarly, the user may step off the platform and the controller needs to be turned off at some point. Furthermore, we wish to model an independent safety controller which monitors and intervenes in the case of extreme angles, hardware failure, sensor failure and so on. In addition, we wish to model: a joystick, allowing the user to turn the ChessWay; degraded behaviour, based on low battery level; a safety key, in case the user falls off; a parking key, allowing the user to safely stop the ChessWay; and feedback to the user, in the form of LED indicators. It is clear that even this simple case study demonstrates the intrinsic complexity of modern real-time control systems.

There are numerous faults which we would hope to explore by developing and co-simulating a ChessWay co-model. These include sensor failures (e.g. missing, late, or

jittery data) of the accelerometer, gyroscope, safety key and steering joystick. Other issues can arise with hardware, such as battery degradation, communication bus faults, and CPU crashes. In addition, complex environmental factors are not currently modelled. For example, uneven surfaces or those where the wheels experience different friction; or scenarios in which the ChessWay collides with obstacles or loses ground contact. Users can also cause faults, such as rapidly leaning forward on the ChessWay, which can lead to a dangerous overreaction of the controller⁶.

The trade-off between safety, functionality and cost price is a typical bottleneck during system design. There are numerous factors to be explored in the ChessWay study. Low-cost accelerometers may be sufficient to meet the basic system requirements, while a more expensive IMU (Inertial Measurement Unit) could deliver a wider safety margin, reduced complexity and better performance, but at a higher cost. Choice of motor, desired top speed and desired running time will affect choice of battery capacity. In turn, battery capacity affects the size and weight of the battery, which affects the design of the frame and so on. Electronics and processors must be selected to meet the timing requirements. Deciding between the myriad options is envisioned as a typical design space exploration task in the DESTTECS project.

The large variety in usage scenarios, functional requirements, environmental conditions and fault types described above makes it clear that suitable analysis of any design can only be sensibly done semi-automatically. It is also clear that current state of the art modelling technology does not provide efficient means to do so (at the appropriate level of abstraction). Rapid co-model analysis by simulation will be used in the DESTTECS project to explore the design space. This iterative process rates each possible design on a number of predefined quantitative and qualitative (and possibly conflicting) design objectives for predefined sets of scenarios, environment conditions and faults. This ranking provides objective insight into the impact of specific design choices and this guides, supports and logs the decision making process.

5 Concluding Remarks

We have presented an approach to collaborative modelling and co-simulation with an emphasis on exploring the design space of alternative fault models. Our work is in very early stages, but we have already demonstrated the feasibility of coupling discrete event models in VDM with continuous-time models in 20-Sim using existing tools. We believe that we have viable apparatus on which to build tools that will allow design exploration in terms co-models (and explicit fault models in particular).

Several authors have argued that the separation of the physical and abstract software worlds impedes the design of embedded systems. Henzinger and Sifakis conclude that a new discipline of embedded systems design, developed from basic theory up, is required [14]. Lee argues that new (time-explicit) abstractions are needed [20]. Our approach brings relevant timing requirements into otherwise conventional formal controller models. We expect that developing methods and tools for collaborative modelling and co-simulation, especially for fault-tolerant systems, will yield insights into the mathematical frameworks required for a unified discipline.

⁶ Search www.youtube.com for “segway crashes” for examples of malicious users.

Our work aims for a pragmatic, targeted exploitation of formal techniques to get the best value from currently under-exploited formal methods and tools. Note that in early design stages, we are not interested in a design that is provably correct under all circumstances, but in finding the class of system models that is very likely to have that property when studied in more detail. This is sound engineering practice, because in reality the cost involved in performing this detailed analysis for a specific design is usually significant and can be performed at most once during the design of a system.

The concept of co-simulation has also attracted interest. Nicolescu et al. [22][21] propose CODIS, a co-simulation approach based on a generic “co-simulation bus” architecture. Verhoef’s semantics [24] appears to differ from CODIS in that there is a direct connection between the CT interface and the operational semantics of the DE system. The COMPASS project [3] aims to support co-engineering of critical on-board systems for the space domain, using probabilistic model checking of AADL models extended with an explicit notion of faults. The MODELISAR⁷ project shares many goals with the DESTECES project, particularly in using co-simulation to aid collaborative design. It is focused on the automotive industry and uses the Modelica [13] modelling language. The main output is the definition of a “Function Mock-up Interface” (FMI), which is essentially a specification for co-simulation. Models can be co-simulated by generating C-code which implement this FMI. There is less focus on explicit fault modelling.

Ptolemy [7] is a radically different actor-based framework for the construction of models of complex systems with a mixture of heterogeneous components, where different models of computation can be used at different hierarchical modelling levels. The Ptolemy execution semantics provides facilities that address similar goals to co-simulation – this is especially true for the built-in tool HyVisual.

Our exploration of co-models is already beginning to raise interesting questions for future work. Even the simple water tank example shows the choices facing modellers – some of which might not be explicit in more conventional development processes. For example, faults can be modelled on either side of the co-model and there is flexibility in how much fault behaviour is encoded within the model, as opposed to the external script driving a co-simulation. Practical experience will yield guidelines for modellers in future. Further, our goal is to develop patterns that allow the automatic (or semi-automatic) enhancement of normative models with descriptions of faulty behaviour.

We have argued that co-simulation should not be decoupled from design space exploration. A necessary activity is to develop forms of ranking and visualisation for test outcomes in order to support model selection. Further, a means of validating system-level timing properties by stating validation conjectures is required [9]. Regarding the co-simulation framework, there are open questions. Here there are open questions about how open the co-model’s interface to the script should be. For example, should private variables and operations be made available to the script, or should the information hiding (in the DE models in particular) be enforced on the script? These are currently open points.

Finally, collaborative modelling and co-simulation have the same strengths and weaknesses as all formal modelling. The predictive accuracy of models depends in turn on the accuracy with which properties, especially timing properties, of components can be

⁷ <http://www.modelisar.org/fmi.html>

determined. The linking of heterogeneous models by co-simulation contracts brings into the light many of the choices and conflicts that are currently only poorly understood, or are made implicitly or with weak justification. It is to be hoped that exposing these decisions to scrutiny will help to reduce the late feedback and rework that characterises so much embedded systems development today.

Acknowledgements. We are grateful to our colleagues in the EU FP7 project DESTTECS, and we especially acknowledge the contributions of Jan Broenink. In addition we would like to thank Nick Battle for providing input on this paper. Fitzgerald's work is also supported by the EU FP7 Integrated Project DEPLOY and by the UK EPSRC platform grant on Trustworthy Ambient Systems (TrAmS).

References

1. Andrews, Z.H., Fitzgerald, J.S., Verhoef, M.: Resilience Modelling through Discrete Event and Continuous Time Co-Simulation. In: Proc. 37th Annual IFIP/IEEE Intl. Conf. on Dependable Systems and Networks, vol. (Supp.), pp. 350–351. IEEE Computer Society, Los Alamitos (June 2007)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 11–33 (2004)
3. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: The compass approach: Correctness, modelling and performability of aerospace systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 173–186. Springer, Heidelberg (2009)
4. Broenink, J.F., Larsen, P.G., Verhoef, M., Kleijn, C., Jovanovic, D., Pierce, K., Wouters, F.: Design support and tooling for dependable embedded control software. In: Proc. of Serene 2010 International Workshop on Software Engineering for Resilient Systems. ACM, New York (2010)
5. Broenink, J.F.: Computer-aided physical-systems modeling and simulation: a bond-graph approach. Ph.D. thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands (1990)
6. Broenink, J.F.: Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD* 38(3), 22–25 (1997)
7. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE* 91(1), 127–144 (January 2003)
8. Elmstrøm, R., Larsen, P.G., Lassen, P.B.: The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices* 29(9), 77–80 (1994)
9. Fitzgerald, J.S., Larsen, P.G., Tjell, S., Verhoef, M.: Validation Support for Real-Time Embedded Systems in VDM++. In: Cukic, B., Dong, J. (eds.) Proc. HASE 2007: 10th IEEE High Assurance Systems Engineering Symposium, pp. 331–340. IEEE, Los Alamitos (November 2007)
10. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development, 2nd edn. Cambridge University Press, Cambridge (2009), ISBN 0-521-62348-0
11. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, New York (2005), <http://www.vdmbook.com>

12. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices* 43(2), 3–11 (2008)
13. Fritzson, P., Engelson, V.: Modelica - a unified object-oriented language for system modelling and simulation. In: Jul, E. (ed.) *ECOOP 1998*. LNCS, vol. 1445, pp. 67–90. Springer, Heidelberg (1998)
14. Henzinger, T., Sifakis, J.: *The Discipline of Embedded Systems Design*. *IEEE Computer* 40(10), 32–40 (2007)
15. Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language (December 1996)
16. Fitzgerald, J.S., Larsen, P.G., Verhoef, M.: Vienna Development Method. In: Wah, B. (ed.) *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., Chichester (2008)
17. Karnopp, D., Rosenberg, R.: *Analysis and simulation of multiport systems: the bond graph approach to physical system dynamic*. MIT Press, Cambridge (1968)
18. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: *The Overture Initiative – Integrating Tools for VDM*. *ACM Software Engineering Notes* 35(1) (January 2010)
19. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Systems using VDM. *International Journal of Software and Informatics* 3(2-3) (October 2009)
20. Lee, E.A.: Computing needs time. *Communications of the ACM* 52(5), 70–79 (2009)
21. Nicolescu, G., Boucheneb, H., Gheorghe, L., Bouchhima, F.: Methodology for efficient design of continuous/discrete-events co-simulation tools. In: Anderson, J., Huntsinger, R. (eds.) *High Level Simulation Languages and Applications*, SCS, San Diego, CA, pp. 172–179 (2007)
22. Nicolescu, G., Bouchhima, F., Gheorghe, L.: CODIS – A Framework for Continuous/Discrete Systems Co-Simulation. In: Cassandras, C.G., Giua, A., Seatzu, C., Zaytoon, J. (eds.) *Analysis and Design of Hybrid Systems*, pp. 274–275. Elsevier, Amsterdam (2006)
23. *Oxford English Dictionary Online*. Oxford University Press (2010)
24. Verhoef, M.: *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Ph.D. thesis, Radboud University Nijmegen (2008), ISBN 978-90-9023705-3
25. Verhoef, M., Visser, P., Hooman, J., Broenink, J.: Co-simulation of Real-time Embedded Control Systems. In: Davies, J., Gibbons, J. (eds.) *IFM 2007*. LNCS, vol. 4591, pp. 639–658. Springer, Heidelberg (2007)
26. Wolff, S., Larsen, P.G., Noergaard, T.: Development Process for Multi-Disciplinary Embedded Control Systems. In: *EuroSim 2010*, EuroSim (September 2010)
27. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. *ACM Computing Surveys* 41(4), 1–36 (2009)

Programming with Miracles

Rajeev Joshi

Laboratory for Reliable Software*
NASA Jet Propulsion Laboratory, Pasadena, CA, USA

Abstract. In his seminal book, *A Discipline of Programming* [EWD 76], Dijkstra proposed that all sequential programs satisfy four laws for their weakest preconditions. By far the catchiest name was reserved for the Law of the Excluded Miracle, which captured the intuition that, started in a given state, a program execution must either terminate or loop forever. In the late 1980s, both Nelson [GN 89] and Morgan [CCM 90] noted that the law was unnecessarily restrictive when writing programs to be used as specifications. In the years since, “miracles” have become a standard feature in specification languages (for instance, the `assume` statement in JML [LLP+00] and BoogiePL [DL 05]).

What is perhaps surprising is that miracles are not as commonly used in programs written as implementations. This is surprising because for many everyday tasks, programming in a language with miracles is often far superior to the popular scripting languages that are used instead. In this talk, we build upon pioneering work by Burrows and Nelson [GN 05] who designed the language LIM (“Language of the Included Miracle”). We describe a language LIMe (“LIM with extensions”), and discuss its application in the context of flight software testing, including the analysis of spacecraft telemetry logs.

References

- [EWD 76] Dijkstra, E.W.: *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs (1976)
- [GN 89] Nelson, G.: A Generalization of Dijkstra’s Calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11(4) (1989)
- [CCM 90] Carroll Morgan, C.: *Programming from specifications*, Prentice Hall International Series in Computer Science, NJ, USA (1990), 2nd edition (1994)
- [LLP+00] Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: *OOPSLA 2000 Companion*, pp. 105–106. ACM, New York (2000)
- [DL 05] DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs, Microsoft Research Technical Report MSR-TR-2005-70 (March 2005)
- [GN 05] Nelson, G.: LIM and Nanoweb, Hewlett-Packard Laboratories Technical Report HPL-2005-41 (February 2005)

* The research described in this talk was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

An Event-B Approach to Data Sharing Agreements

Alvaro E. Arenas, Benjamin Aziz, Juan Bicarregui, and Michael D. Wilson

e-Science Centre, STFC Rutherford Appleton Laboratory
Oxfordshire OX11 0QX, U.K.

{alvaro.arenas, benjamin.aziz, juan.bicarregui,
michael.wilson}@stfc.ac.uk

Abstract. A Data Sharing Agreement (DSA) is a contract among two or more principals regulating how they share data. Agreements are usually represented as a set of clauses expressed using the deontic notions of obligation, prohibition and permission. In this paper, we present how to model DSAs using the Event-B specification language. Agreement clauses are modelled as temporal-logic formulas that preserve the intuitive meaning of the deontic operators, and constrain the actions that a principal can execute. We have exploited the ProB animator and model checker in order to verify that a system behaves according to its associated DSA and to validate that principals' actions are in agreement with the DSA clauses.

Keywords: Data Sharing Agreements; Formal Analysis; Event-B.

1 Introduction

Data sharing is increasingly important in modern organisations. Every organisation requires the regular exchange of data with other organisations. Although the exchange of this data is vital for the successful inter-organisational process, it is often confidential, requiring strict controls on its access and usage. In order to mitigate the risks inherent in sharing data between organisations, Data Sharing Agreements (DSAs) are used to ensure that agreed data policies are enforced across organisations [1].

A DSA is a legal agreement among two or more parties regulating who can access data, when and where, and what they can do with it. DSAs either include the data policies explicitly as clauses, or include existing organisational data policies by reference. DSA clauses includes deontic notions stating permissions for data access and usage, prohibitions on access and usage which constrain these permissions, and obligations that the principles to the agreement must fulfil. DSA can be created between an organisation and each of many collaborators. A single data set may result from multiple sources, so that the clauses in the DSA with each contributor need to be combined together and applied to it. DSAs are represented in natural language with its concomitant ambiguities and potential conflicts, which are exacerbated by DSA combination. Natural language DSA clauses can be enforced by transforming them into executable policy languages [5]. However, support is required to ensure this transformation conveys the intention of the natural language, while avoiding its potential problems. The formal representation and analysis of agreement clauses could provide this support.

This paper presents a formalisation of DSAs using the Event-B specification language [2]. This involves incorporating normative deontic notions (obligations, permissions and prohibitions) into the Event-B modelling process. The main contribution of the paper is the development of an approach to model agreements using the Event-B specification language. The process itself forces the explicit statement of implicit assumptions underlying the natural language which are required in the formal modelling. The contribution includes a method to transform natural-language data-sharing agreement clauses, including deontic notions, into linear temporal logic predicates suitable for verification and validation.

The paper is structured as follows. Section 2 introduces DSAs and their main components. Then, section 3 summarises the Event-B method. Section 4 presents our approach to model contracts in Event-B, and section 5 applies that approach to a DSA for scientific collaborations. Section 6 discusses the verification and validation of contracts in Event-B. Finally, section 7 relates our work with others, and section 8 presents concluding remarks and highlights future work.

2 An Overview of Data Sharing Agreements

To introduce DSAs, we take as an example data sharing in scientific collaborations [4]. Large-scale research facilities such as particle accelerators and synchrotrons are used by teams of scientists from around the world to produce data about the structure of materials which can be used in many ways, including to create new products, change industrial methods or identify new drugs. There are many individuals and organisations involved in these processes who make agreements with each other to share the data from the facilities, within specified time limits, and for use in particular ways.

Data sharing requirements are usually captured by means of collaboration agreements among the partners, typically established during the scientific proposal preparation. They usually contains clauses defining what data will be shared, the delivery/transmission mechanism, the processing and security framework, among others. Following [12], a DSA consists of a definition part and a collection of agreement *Clauses*. The definition part usually includes the list of involved *Principals*; the start and end dates of the agreement; and the list of *Data* covered by the agreement. Three type of clauses are relevant for DSAs: *Authorisation*, *Prohibitions* and *Obligation* clauses. Authorisations indicate that specified roles of principal are authorised to perform actions on the data within constraints of time and location. Prohibitions act as further constraints on the authorisations, prohibiting actions by specific roles at stated times and locations. Obligations indicate that principals, or the underlying infrastructure, are required to perform specified actions following some event, usually within a time period. The DSA will usually contain processes to be followed, or systems to be used to enforce the assertions, and define penalties to be imposed when clauses are breached.

For instance, in the case of scientific collaborations, principals typically involve *investigators* and *co-investigators* from universities or industrial research departments, and a *scientific facility provider*, whose facilities provide the infrastructure for

performing scientific experiments. Data generated from experiments is usually held on a *experimental database*, which is exposed via a web service API. The following are examples of DSA clauses, taken from existing DSAs:

1. During the embargo period, access to the experimental data is restricted to the principal investigator and co-investigators.
2. After the embargo period, the experimental data may be accessed by all users.
3. Access to data must be denied to users located in un-secure locations such as the cafeteria.
4. System must notify principal investigator when a public user access experimental data, within 2 days after the access.
5. User must renew use license within 30 days if it has expired before the three year embargo period.

Clauses 1 and 2 correspond to authorisation clauses. Clause 3 is a prohibition. Clauses 4 and 5 are examples of obligation clauses.

2.1 Main Components of Data Sharing Agreements

The set-up of DSAs requires technologies such as DSA authoring tools [8], which may include controlled natural language vocabularies to define unambiguously the DSAs conditions and obligations; and translators of DSAs clauses into enforceable policies. Our work aims at providing formal reasoning support to DSA authoring tools such as the one presented in [8], focusing mainly on the clauses part of a DSA.

We represent DSA clauses as guarded actions, where the guard is a predicate characterising environmental conditions, such as time and location, or restrictions for the occurrence of the event, such as "user is registered" or "data belongs to a project".

Definition 1. (Action). *An action is a tuple consisting of three elements $\langle p, an, d \rangle$, where p is the principal, an is an action name, and d is the data.*

Action $\langle p, an, d \rangle$ expresses that the principal p performs action name an on the data d . Action names represent atomic permissions, where actions are built from by adding the identity of the principal performing the action name and the data on which the action name is performed. Actions are analogous to *fragments* defined in [8]. We assume that actions are taken from a pre-defined list of actions, possibly derived from an ontology. An example of an action is "Alice accesses experimental data", where "Alice" is the principal, "accesses" is the action and "experimental data" is the data.

We are interested in four types of clauses: permissions, prohibitions, bounded obligations, and obligations. Clauses are usually evaluated within a specific context, which is represented by a predicate characterising environmental conditions such as location and time.

Definition 2. (Agreement Clause). *Let G be a predicate, n an integer denoting a time unit, and $a = \langle p, an, d \rangle$ be an action. The syntax of agreement clauses is defined as follows:*

$$\begin{aligned}
C ::= & \text{IF } G \text{ THEN } \mathcal{P}(a) \quad | \quad \text{IF } G \text{ THEN } \mathcal{F}(a) \\
& | \quad \text{IF } G \text{ THEN } \mathcal{O}(a) \quad | \quad \text{IF } G \text{ THEN } \mathcal{O}_n(a)
\end{aligned}$$

In the following, we provide an intuitive explanation of the clause syntax. A more precise meaning will be given later when representing agreement clauses in the Event-B language. A permission clause is denoted as $\text{IF } G \text{ THEN } \mathcal{P}(a)$, which indicates that provided the condition G holds, the system may perform action a . A prohibition clause is denoted as $\text{IF } G \text{ THEN } \mathcal{F}(a)$, which indicates that the system must not perform action a when condition G holds. An obligation clause is denoted as $\text{IF } G \text{ THEN } \mathcal{O}(a)$, which indicates that provided the condition G holds, the system eventually must perform action a . Finally, a bounded-obligation clause is denoted as $\text{IF } G \text{ THEN } \mathcal{O}_n(a)$, which indicates that provided the condition G holds, the system must perform action a within n time units.

A data sharing agreement can be defined as follows.

Definition 3. (Data Sharing Agreement). A DSA is a tuple $\langle Principals, Data, ActionNames, fromTime, endTime, \mathbb{P}(C) \rangle$.

Principals is the set of principals signing the agreement and abiding by its clauses. *Data* is the data elements to be shared. *ActionNames* is a set containing the name of the actions that a party can perform on a data. *fromTime* and *endTime* denotes the starting and finishing time of the agreement respectively; this is an abstraction representing the starting and finishing date of the agreement. Finally, $\mathbb{P}(C)$ is the set of clauses of the agreement.

3 Event-B with Obligations

3.1 Introduction to Event-B

Event-B [2] is an extension of Abrial's B method [1] for modelling distributed systems. In Event-B, machines are defined in a context, which has a unique name and is identified by the keyword `CONTEXT`. It includes the following elements: `SETS` defines the sets to be used in the model; `CONSTANTS` declares the constants in the model; and finally, `AXIOMS` defines some restrictions for the sets and includes typing constraints for the constants in the way of set membership.

An Event-B machine is introduced by the `MACHINE` keyword, it has a unique name and includes the following elements. `VARIABLES` represents the variables (state) of the model. `INVARIANT` describes the invariant properties of the variables defined in the clause `VARIABLES`. Typing information and general properties are described in this clause. These properties shall remain true in the whole model and in further refinements. Invariants need to be preserved by the initialisation and events clauses. `INITIALISATION` allows to give initial values to the variables of the system. `EVENTS` cause the state to change by updating the values of the variables as defined by the *generalised substitution* of the event. Events are guarded by a *condition*, which when satisfied implies that the event is permitted to execute by applying its generalised substitution in the current state of the machine.

Event-B also incorporates a refinement methodology, which can be used by software architects to incrementally develop a model of a system starting from the initial most abstract specification and following gradually through layers of detail until the model is close to the implementation.

In Event-B, an event is defined by the syntax: `EVENT e WHEN G THEN S END`, where G is the guard, expressed as a first-order logical formula in the state variables, and S is any number of generalised substitutions, defined by the syntax $S ::= x := E(v) \mid x := z : |P(z)$. The deterministic substitution, $x := E(v)$, assigns to variable x the value of expression $E(v)$, defined over set of state variables v . In a non-deterministic substitution, $x := z : |P(z)$, it is possible to choose non-deterministically local variables, z , that will render the predicate $P(z)$ true. If this is the case, then the substitution, $x := z$, can be applied, otherwise nothing happens.

3.2 Linear Temporal Logic in Event-B

The Event-B Rodin platform¹ has associated tools, such as ProB² and Atelier B³ for expressing and verifying Linear Temporal Logic (LTL) formulae properties of Event-B specifications. LTL formulae are defined based on a set of propositional variables, $p_1, p_2 \dots$, the logical operators, \neg (not), \wedge (and), \vee (or), \rightarrow (implication), as well as future temporal operators: \square (always), \diamond (eventually), \circ (next), \mathcal{U} (until), \mathcal{W} (weak until) and \mathcal{R} (release). It also includes dual operators for modelling the past.

The propositions themselves can be *atomic*, which include predicates on states written as $\{\dots\}$, and the event enabled predicate (in the case of ProB), written as `enabled(\bar{a})`, which states that event \bar{a} is enabled in the current state. They can also be *transition* propositions, such as $[\bar{a}]$, to state that the next event to be executed in the path is \bar{a} . Finally, $(P \Rightarrow Q)$ is often written to denote the formula $\square(P \rightarrow Q)$.

3.3 Modelling Obligations in Event-B

Classical Event-B does not include a notion of obligation. When the guard of an event is true, there is no obligation to perform the event and its execution may be delayed as a result of, for example, interleaving it with other permitted events. The choice of scheduling permitted events is made non-deterministically. In [3], we describe how obligations can be modelled in Event-B as events, interpreting the guard of an event as a *trigger* condition. An obliged event is written as `EVENT e WHEN T WITHIN n NEXT S END`, where T is the trigger condition such that when T becomes true, the event must be executed within at most $n + 1$ number of time units, provided the trigger remains true. If the trigger changes to false within that number, the obligation to execute the event is canceled. This type of event represents a bounded version of the leads-to modality, represented by the obligation $(T \Rightarrow \diamond_{\leq n}(T \rightarrow e))$. The obliged event is a syntactic sugar and can be encoded and refined in the standard Event-B language (see [3]). In the rest of the paper, we adopt Event-B with obligations as our specification language.

¹ www.event-b.org/platform.html

² www.stups.uni-duesseldorf.de/ProB

³ www.atelierb.eu

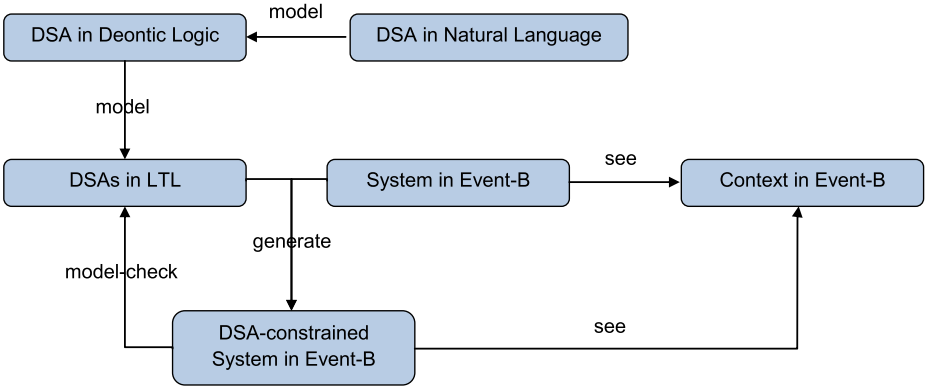


Fig. 1. A Lifecycle for Modelling DSAs in Event-B

4 Formal Modelling of Data Sharing Agreements in Event-B

This section introduces our method for modelling DSAs in Event-B. The method consists of a number of stages, as shown in Figure 1.

First, the system domain and variables are defined, and actions are identified (see subsections 4.1 and 4.2). Second, once defined the system vocabulary, the agreement clauses are modelled using the deontic operators introduced in definition 2 (see subsection 4.3). Finally, each agreement clause is modelled as a logical formula that holds as an invariant in the Event-B system (see subsections 4.4, 4.5 and 4.6).

4.1 Defining System Domains

The first step in the modelling of DSAs in Event-B consists of defining the domains to be used in the model. This corresponds to the definition of primitive sets and constant values. A DSA includes three main sets: *PRINCIPALS* indicating the principals participating in the DSA; *DATA* indicating the data to be shared, and *PACTIONNAMES* denoting the actions that principals may perform on data. Following definition 1, an action is defined as a tuple consisting of a principal, an action name, and a data. In addition, we are interested in registering the actions performed by the system, therefore we introduce set *SACTIONNAMES* to indicate such actions. The main motive in distinguishing user and system action names stems from our later treatment of obligations where we distinguish between the non-enforceable obligations on users and the enforceable obligations on systems.

Our example follows a role-based model, hence we identify the roles to be used in the system; *LOCATIONS* indicates the set of location of a principal may be located. Below, we present the context for our DSA example. We are using a slightly different syntax to the one used in Event-B for readability purpose, associating each set with its constant elements.

CONTEXT

$$\begin{aligned}
PRINCIPALS &= \{ Alice, Bob, Charlie, \dots \} \\
PACTIONNAMES &= \{ access, renewlicence, \dots \} \\
DATA &= \{ ExpData, \dots \} \\
PACTION &= PRINCIPALS \times PACTIONNAMES \times DATA \\
SACTIONNAMES &= \{ notify \} \\
MESSAGE &= PRINCIPALS \times DATA \\
SACTION &= SACTIONNAMES \times PRINCIPAL \times MESSAGE \\
LOCATIONS &= \{ room1, room2, cafeteria, \dots \} \\
ROLES &= \{ PI, CoI, PublicUser \}
\end{aligned}$$
4.2 Modelling System Variables

The following are the main variables associated to a DSA. *action* is the input to the system. *actionLog* represents the actions performed by principals, where each action is labelled with the occurrence time. Those actions performed by the system are logged into the *systemLog*. Variables *fromTime*, *endTime* and *currentTime* denote the starting time of the agreement, final time, and the current time respectively. In addition, there are domain-specific variables related to specific DSAs. For our example of DSA in the scientific domain, we require variables such as *embargoTime*, indicating the end of the embargo on the data being shared; *location* associates each principal with his current location and *safeLocation* indicates if a location is safe. The relation *roleAssig* associates principals with their role in the system; and *getPI* is a function that given a data returns its associated principal investigator. Finally, *licenceExpTime* indicates the time when the licence of a principal for accessing a data expires. Below, we present the variables of our DSA, their domain, and their main properties.

INVARIANTS

$$\begin{aligned}
action &\in PACTION \cup SACTION \\
actionLog &\in \mathbb{P}(\mathbb{N} \times PACTION) \\
systemLog &\in \mathbb{P}(\mathbb{N} \times SACTION) \\
fromTime &\in \mathbb{N} \\
endTime &\in \mathbb{N} \\
currentTime &\in \mathbb{N} \\
location &\in PRINCIPALS \rightarrow LOCATIONS \\
safeLocation &\in LOCATIONS \rightarrow \mathbb{B} \\
embargoTime &\in DATA \rightarrow \mathbb{N} \\
getPI &\in DATA \rightarrow PRINCIPALS \\
roleAssig &\in (PRINCIPALS \times DATA) \rightarrow ROLES \\
licenceExpTime &\in (PRINCIPALS \times DATA) \rightarrow \mathbb{N} \\
fromTime &\leq currentTime \leq endTime \\
\forall t \in \text{ran } licenceExpTime \cdot t &\leq endTime \\
\forall d \in DATA \cdot roleAssig(getPI(d), d) &= PI
\end{aligned}$$

4.3 Initial Modelling of Agreement Clauses

Having defined system variables enables us to model the guard associated to each clause, and to identify the actions of the system. The next step consists in representing agreement clauses in the style presented in Definition 2. For instance, the clauses described in Section 2 can be modelled as follows.

$ \begin{aligned} C_1 : & \forall p \in PRINCIPALS, d \in DATA \cdot \\ & \text{IF } (d = ExpData \wedge \\ & \quad currentTime \leq embargoTime(d) \wedge roleAssig(p, d) \in \{PI, CoI\}) \\ & \text{THEN } \mathcal{P}(\langle p, access, d \rangle) \\ C_2 : & \forall p \in PRINCIPALS, d \in DATA \cdot \\ & \text{IF } (d = ExpData \wedge currentTime > embargoTime(d)) \\ & \text{THEN } \mathcal{P}(\langle p, access, d \rangle) \\ C_3 : & \forall p \in PRINCIPALS, d \in DATA \cdot \\ & \text{IF } \neg safeLocation(location(p)) \\ & \text{THEN } \mathcal{F}(\langle p, access, d \rangle) \\ C_4 : & \forall p \in PRINCIPALS, d \in DATA \cdot \\ & \text{IF } (\langle p, access, d \rangle \in ran(actionLog) \wedge roleAssig(p, d) = PublicUser) \\ & \text{THEN } \mathcal{O}_2(\langle notify, getPI(d), (p, d) \rangle) \\ C_5 : & \forall p \in PRINCIPALS, d \in DATA \cdot \\ & \text{IF } currentTime > licenceExpTime(p, d) \\ & \text{THEN } \mathcal{O}_{30}(\langle p, renewlicence, d \rangle) \end{aligned} $

4.4 Modelling Permission and Prohibition Clauses

We proceed now to represent agreement clauses in Event-B. As a starting point, we assume that principal actions are modelled in the system as Event-B events. For instance, if $\langle p, a, d \rangle$ is an arbitrary principal action, we assume then there is an event called \bar{a} that models the effect of principal p performing action a on data d as a guarded substitution.

<pre> EVENT $\bar{a1}$ ANY <i>action</i> WHERE <i>action</i> = $\langle p1, a1, d1 \rangle$ THEN ... /* Execution of action <i>a1</i> */ END EVENT $\bar{a2}$ ANY <i>action</i> WHERE <i>action</i> = $\langle p2, a2, d2 \rangle$ THEN ... /* Execution of action <i>a2</i> */ END ... </pre>
--

We represent agreement clauses as assertions that the system must validate. Let $\langle p, a, d \rangle$ be an arbitrary principal action and \bar{a} be its associated Event-B event. Let $\text{IF } G \text{ THEN } \mathcal{P}(\langle p, a, d \rangle)$ be an arbitrary permitted clause. The following assertion must be valid in the system, indicating that the event associated to the clause may be executed when condition G holds:

$$((action = \langle p, a, d \rangle \wedge G) \Rightarrow enabled(\bar{a}))$$

where $enabled$ is the event-B enable proposition, indicating that an event is enabled in the current state.

Let $\text{IF } G \text{ THEN } \mathcal{F}(\langle p, a, d \rangle)$ be an arbitrary prohibition clause. The following assertion must be valid in the system, indicating that the event associated to the clause must not be executed when condition G holds:

$$((action = \langle p, a, d \rangle \wedge G) \Rightarrow \neg \text{enabled}(\bar{a}))$$

In order to validate agreement clauses, some changes are needed in the structure of the events associated to actions. First, the permitted clauses indicate when an action may be executed, then the guard of the associated actions should be strengthened with the disjunction of the clauses conditions. Second, the prohibition clauses indicate when an action must not be executed, then the guard of the associated actions is strengthened with the conjunction of the negation of the clauses. Formally, let $\langle p, a, d \rangle$ be an arbitrary principal action and \bar{a} be its associated event. Let $\text{IF } G_i \text{ THEN } \mathcal{P}(\langle p_i, a, d_i \rangle)$, for $i = 1, \dots, k$ be all the permitted clauses associated to action a , and let $\text{IF } G'_j \text{ THEN } \mathcal{F}(\langle p_j, a, d_j \rangle)$, for $j = 1, \dots, l$ be all the prohibition clauses associated to action a . The event is transformed as follows to meet the agreement clauses.

$$\begin{array}{l} \text{EVENT } \bar{a} \text{ ANY } action \text{ WHERE } action = \langle p, a, d \rangle \wedge \bigvee_{i=1}^k G_i \wedge \bigwedge_{j=1}^l \neg G'_j \text{ THEN} \\ \quad \dots \parallel actionLog := actionLog \cup \{currentTime \mapsto (p \mapsto a \mapsto d)\} \\ \text{END} \end{array}$$

4.5 Modelling Obligations on the System

Without loss of generality, we will consider only bounded-obligation clauses. Any unbounded obligation in a DSA can be transformed into a bounded one by limiting it by the duration of the contract. Let $\langle b, c, d \rangle$ be an arbitrary system action; and $\mathcal{C} : \text{IF } G \text{ THEN } \mathcal{O}_k(\langle b, c, d \rangle)$ be an arbitrary obligation clause. This imposes an obligation on the system expressed by the following temporal logic formula.

$$\begin{array}{l} (G \wedge currentTime < clock_c + k) \Rightarrow \\ \quad \diamond ((\neg G \vee (currentTime \mapsto (b \mapsto c \mapsto d)) \in systemLog) \wedge \\ \quad \quad currentTime \leq clock_c + k) \end{array}$$

In above LTL formula, variable $clock_c$ indicates the time when trigger condition G becomes true. The formula expresses that condition G holds for at most k time units, until a new action $\langle b, c, d \rangle$ is registered in the $systemLog$. This is indeed the intuitive meaning of the obligation clause expressed using deontic operators. The above obligation could be enforced by adding a new event performing the associated change in the system state, as described in subsection [5.4](#).

4.6 Modelling Obligations on Users

Let $\langle p, a, d \rangle$ be a principal action and $\mathcal{C} : \text{IF } G \text{ THEN } \mathcal{O}_k(\langle p, a, d \rangle)$ be an arbitrary obliged clause. In general, the system cannot enforce users' obligations, but it can detect when a user's obligation has not been fulfilled, as illustrated by the formula below.

$$\begin{array}{l} (G \wedge currentTime < clock_c + k) \Rightarrow \\ \quad \diamond ((\neg G \vee (currentTime \mapsto (p \mapsto a \mapsto d)) \in actionLog) \wedge \\ \quad \quad currentTime \leq clock_c + k) \end{array}$$

5 An Example of a DSA in Event-B

This section applies the method presented previously to our DSA example for scientific collaborations. The first two steps, defining system domain and variables and expressing the agreement clauses using deontic operators, were presented in section 4. We continue with the modelling of the clauses as logic formulae.

5.1 The Agreement Clauses as Logic Formulae

Below we model the clauses introduced in subsection 4.3 as logic formulae, following the patterns presented in the previous section. Permission and prohibition clauses are modelled in predicate logic, and can be represented as invariants of the system. Obligations are modelled in LTL, and can be represented as theorems that the system can verify using model-checking techniques or validate via simulators. We assume that all clauses are universally quantified with the variable $p \in PRINCIPALS$ and $d \in DATA$.

$$\begin{aligned}
C_1 &: (action = \langle p, access, d \rangle \wedge d = ExpData \wedge \\
&\quad currentTime \leq embargoTime(d) \wedge roleAssig(p, d) \in \{PI, CoI\}) \Rightarrow \\
&\quad enabled(\overline{access}) \\
C_2 &: (action = \langle p, access, d \rangle \wedge d = ExpData \wedge currentTime > embargoTime(d)) \Rightarrow \\
&\quad enabled(\overline{access}) \\
C_3 &: (action = \langle p, access, d \rangle \wedge \neg safeLocation(location(p))) \Rightarrow \\
&\quad \neg enabled(\overline{access}) \\
C_4 &: ((clock_{c_4} \mapsto (p \mapsto access \mapsto d)) \in actionLog) \wedge \\
&\quad roleAssig(p, d) = PublicUser \wedge currentTime < clock_{c_4} + 2 \Rightarrow \\
&\quad \diamond((currentTime \mapsto (notify \mapsto getPI(d) \mapsto (p \mapsto d))) \in systemLog \wedge \\
&\quad\quad currentTime \leq clock_{c_4} + 2) \\
C_5 &: (currentTime > licenceExpTime(p, d) \wedge currentTime < clock_{c_5} + 30) \Rightarrow \\
&\quad \diamond((currentTime \mapsto (p \mapsto renewlicence \mapsto d)) \in actionLog) \wedge \\
&\quad\quad currentTime \leq clock_{c_5} + 30
\end{aligned}$$

5.2 Initialising the System

For completeness sake, we include here the initialisation of the system. The DSA is assumed to have a duration of 100 time units.

INITIALISATION

$$\begin{aligned}
actionLog, systemLog &:= \emptyset, \emptyset \\
fromTime, endTime, currentTime &:= 1, 100, 1 \\
location &:= \{ Alice \mapsto room1, Bob \mapsto room2, Charlie \mapsto cafeteria \} \\
safeLocation &:= \{ room1 \mapsto TRUE, room2 \mapsto TRUE, cafeteria \mapsto FALSE \} \\
embargoTime &:= \{ ExpData \mapsto 60 \} \\
getPI &:= \{ ExpData \mapsto Alice \} \\
roleAssig &:= \{ (Alice, ExpData) \mapsto PI, (Bob, ExpData) \mapsto CoI, \\
&\quad (Charlie, ExpData) \mapsto PublicUser \} \\
licenceExpTime &:= \{ (Alice, ExpData) \mapsto 50, (Bob, ExpData) \mapsto 30, \\
&\quad (Charlie, ExpData) \mapsto 30 \}
\end{aligned}$$

5.3 User Actions as Event-B Events

The system includes two events: $\overline{\text{access}}$, indicating that a principal will access a data, and $\overline{\text{renewlicence}}$, indicating that a principal has renewed the licence to access a data.

In the case of $\overline{\text{access}}$, the event guard is strengthened with the disjunction of the guards of its associated permitted clauses (clauses C_1 and C_2 in subsection 4.3) and the conjunction of the negation of the forbidden clauses (clause C_3).

```

EVENT  $\overline{\text{access}}$  ANY action WHERE
  action =  $\langle p, \text{access}, d \rangle \wedge d = \text{ExpData} \wedge$ 
   $((\text{currentTime} \leq \text{embargoTime}(d) \wedge (\text{roleAssig}(p, d) \in \{PI, CoI\})) \vee$ 
     $\text{currentTime} > \text{embargoTime}(d)) \wedge$ 
   $\text{safeLocation}(\text{location}(p))$  THEN
     $\text{actionLog} := \text{actionLog} \cup \{\text{currentTime} \mapsto (p \mapsto \text{access} \mapsto d)\}$ 
END

```

For $\overline{\text{renewlicence}}$, we introduce a new constant $RENEWTIME$ that indicates the constant time a licence is increased once the event occurs.

```

EVENT  $\overline{\text{renewlicence}}$  ANY action WHERE
  action =  $\langle p, \text{renewlicence}, d \rangle$  THEN
     $\text{licenceExpTime}(p, d) := \text{currentTime} + RENEWTIME$  ||
     $\text{actionLog} := \text{actionLog} \cup \{\text{currentTime} \mapsto (p \mapsto \text{renewlicence} \mapsto d)\}$ 
END

```

5.4 System Obligations as Events

The system is obliged to notify the principal investigator when a data item is accessed by a public user. This is modelled as the obligated Event-B event $\overline{\text{notify}}$, which must be executed within 2 time units after the guard holds.

```

EVENT  $\overline{\text{renewlicence}}$  ANY action WHERE
  action =  $\langle p, \text{renewlicence}, d \rangle$  THEN
     $\text{licenceExpTime}(p, d) := \text{currentTime} + RENEWTIME$  ||
     $\text{actionLog} := \text{actionLog} \cup \{\text{currentTime} \mapsto (p \mapsto \text{renewlicence} \mapsto d)\}$ 
END

```

5.5 User Obligations as Events

As mentioned before, user obligations cannot be enforced by the system, although they can be detected. Below, we show the $\overline{\text{checklicence}}$ event, which checks if a principal has renewed a licence for accessing a data within 30 time units after expiring the licence. If the licence is not renewed, the principal is *blacklisted* for using such data. To model such action, we assume the existence of a predicate $\text{blacklist} : \text{PRINCIPALS} \times \text{DATA} \rightarrow \mathbb{B}$, which is initialised as false for any principal and any data. We also assume that the $RENEWTIME$ constant is greater than 30 time units. The system designer could, for instance, use the blacklist information to restrict access to the associated data.

```

EVENT  $\overline{\text{checklicence}}$  WHEN
  ( $\text{currentTime} > \text{licenceExpTime}(p, d)$ ) WITHIN 30 NEXT
   $\text{blacklist}(p, d) := (\text{currentTime} > \text{licenceExpTime}(p, d))$ 
END

```

5.6 Dealing with the Environment

In our model of the DSAs in Event-B, we assume that the environment is modelled through time and location. Our notion of time can be based on some standard representation (e.g. ISO 8601), which could include a combined date and time representation. We assume, for simplicity, that currentTime is a natural number and that every event in the machine representing an action will perform its own time incrementation. Hence, the event representing action \bar{a} will become as follows, assuming that the event lasts for only one time unit,

```

EVENT  $\bar{a}$  WHEN ... THEN
  ... ||  $\text{currentTime} := \text{currentTime} + 1$  END

```

The other environment variable is location, where we have assumed the second solution, i.e. a separate event for changing locations of users, which has overrides the current value of the location function with a new location for some principal.

6 Formal Verification and Validation of DSA Properties

6.1 Verifying DSA Properties

The minimum global property that must hold for any DSA is that all permissions, prohibitions and system obligations stated in the DSA clauses must hold true.

One example of common conflicts is when an event corresponding to a principal action is both enabled (permitted) and disabled (prohibited) in the same state of the machine. Verifying that the model is free from this sort of conflicts corresponds to model-checking the following LTL formulae:

$$\square \neg (\text{enabled}(\bar{a}) \wedge \neg (\text{enabled}(\bar{a})))$$

Another example of conflicts is when obligations are not permitted. Hence, for example, a user obligation of the sort $\text{IF } G \text{ THEN } \mathcal{O}_n(\langle p, a, d \rangle)$ must be permitted,

$$\forall t \in \mathbb{N} \cdot (G \wedge \text{currentTime} = t) \Rightarrow \neg ((\neg \text{enabled}(\bar{a})) \mathcal{U} (\text{currentTime} > t + n))$$

which means that the obliged event \bar{a} must have been enabled at some stage prior to the current time passing the deadline $t + n$.

Healthiness properties are desirable aspects of the DSA that are not expressed by any of the previous properties we mentioned above. For example, healthiness property of our DSA for scientific collaborations state that all accesses to experimental data are performed by principals with a valid licence. This would correspond to the following formula, which was verified in our system.

$$\forall p \in \text{PRINCIPALS}, d \in \text{DATA}, n \in \mathbb{N} : \\ (n, (p, \text{access}, d)) \in \text{actionLog} \Rightarrow n \leq \text{licenceExpTime}(p, d)$$

Other possible healthiness properties would be: to check that no accesses occur at unsafe locations; to prove that system-notification events are idempotent; and to verify that any penalties associated with obligation violation (both for systems and users) are properly enforced. For example, in the case of violating user obligations, that the associated capabilities (certain events) are disabled if the user has not fulfilled their obligation.

6.2 Validating DSA Properties

The validation of the example DSA was carried out using the animation capabilities of the Pro-B plug-in for Rodin. This method of validation has a number of advantages for revealing problems with the specification, mainly:

- It helps monitor every value of the state variables as the machine executes each event. This may reveal certain under-specifications of the types of variables. For example, in one instance, it was discovered that sets are not sufficient to model logs (both action and system logs), since sets do not distinguish different instances of the same actions. Hence, timestamps were added to achieve that effect.
- It helps to view which of the events of the machine are currently active. In the case of modelling user obligations, this is quite helpful since it can reveal whether the obligation conditions are strong enough to disable other events in the case where the obliged event has not yet been executed. More generally, this gives an idea as to whether the activation of events is as expected or not.
- Simulating the machine allows understanding better traces that violate invariants, for those invariants that cannot be verified. For example, in the case of the obligations on users to renew the licence when it expires, these cannot be enforced. Hence, there are runs of the machine in which the user will not renew the licence. Therefore, simulation is beneficial since it will demonstrate the effects of not fulfilling such obligations.

7 Related Work

There have been other attempts to model and analyse contracts using event/state-based modelling approaches. [10] presents how standard conventional contracts can be described by means of Finite State Machines (FSMs). Rights and obligations extracted from the clauses of the contract are mapped into the states, transition and output functions, and input and output symbols of a FSM. The FSM representation is then used to guarantee that the clauses stipulated in the contract are observed when the contract is executed. The approach is more directed to contract monitoring than analysis, since no formal reasoning is included.

In [6], Daskalopulu discusses the use of Petri Nets for contract state tracking, and assessing contract performance. Her approach is best suited for contracts which can

naturally be expressed as protocols, or workflows. She model-checks a contract to verify desired properties. Our work has been inspired by hers, but it differs in that we separate the modelling of the contract – declarative approach by representing clauses as temporal-logic predicates – from the modelling of the system. Following similar objectives, [7] uses the Event Calculus to represent a contract in terms of how its state evolves according to a narrative of (contract-related) events.

An initial model of DSAs is proposed in [12]. The model is based on dataflow graphs whose nodes are principals with local stores, and whose edges are channels along which data flows. Agreement clauses are modelled as obligation constraints expressed as distributed temporal logic predicates over data stores and data flows. Although the model is formal, they do not exploit formal reasoning about agreements.

In [9], the authors propose an event language with deontic concepts like permissions, rights and obligations, which are relevant to the modelling of DSAs. However, their main focus is the access control framework, whereas we focus on a more general framework related to clauses that may occur in any DSA.

Our work has been influenced by the work by Matteucci *et al* [8], which presents an authoring tool for expressing DSAs in a controlled natural language. One of our aims is to provide formal reasoning support for such a tool. The underlying semantic model for DSAs in [8] is an operational semantics expressing how a system evolves when executed under the restriction of an agreement. Within this view, agreement clauses are encoded within the system. We consider that for formal reasoning, it is important to separate the agreement clauses from the system functionality, hence we have proposed to represent clauses as LTL assertions that the system must respect.

8 Conclusion and Future Work

This paper presents an approach for modelling contracts using the Event-B specification language. This involves incorporating normative deontic notions (obligations, permissions and prohibitions) into the Event-B modelling process. We have focused on one particular type of contract, the so-called data sharing agreements (DSAs).

The starting point of the proposed method is an informal DSA. In order to formalise it, the following steps are proposed. First, the system domain and variables are defined, and actions are identified. Second, agreements clauses are modelled using deontic logic. Third, each deontic-logic clause is represented in linear temporal logic (LTL), and the Event-B event dealing with the action is transformed so that the LTL predicate is valid in the model. The relation between the LTL clause and the Event-B model is established by applying verification (model-checking) and validation (animation) techniques.

DSAs typically follow a lifecycle comprising the following stages: (1) *contract drafting*, which includes the drafting of contracts with the aid of authoring tools; (2) *contract analysis*, which includes the formalisation and analysis of contracts in order to detect potential conflicts among contract clauses; (3) *policy derivation* from the contract clauses; and (4) finally, *monitoring and enforcement* of contract policies. This paper has concentrated on the second stage, contract analysis. As future work, we plan to study formally the policy derivation process. In addition, we will investigate the

problem of agreement evolution (changes in an agreement), and whether those changes can be verified/maintained using refinement techniques.

Acknowledgment

This work was partly supported by the EU FP7 project Consequence (Context-Aware Data-Centric Information Sharing), project grant 214859.

References

1. Abrial, J.-R.: *The B Book*. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae* 77(1-2), 1–28 (2007)
3. Bicarregui, J., Arenas, A.E., Aziz, B., Massonet, P., Ponsard, C.: Toward Modelling Obligations in Event-B. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 181–194. Springer, Heidelberg (2008)
4. Crompton, S., Aziz, B., Wilson, M.D.: Sharing Scientific Data: Scenarios and Challenges. In: *W3C Workshop on Access Control Application Scenarios* (2009)
5. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder Policy Specification Language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) *POLICY 2001*. LNCS, vol. 1995, pp. 18–38. Springer, Heidelberg (2001)
6. Daskalopulu, A.: Model Checking Contractual Protocols. In: *Legal Knowledge and Information Systems. Frontiers in Artificial Intelligence and Applications Series* (2001)
7. Farrell, A.D.H., Sergot, M.J., Sallé, M., Bartolini, C.: Using the Event Calculus for Tracking the Normative State of Contracts. *International Journal of Cooperative Information Systems* 14(2-3), 99–129 (2005)
8. Matteucci, I., Petrocchi, M., Sbodio, M.L.: CNL4DSA a Controlled Natural Language for Data Sharing Agreements. In: *25th Symposium on Applied Computing, Privacy on the Web Track*. ACM, New York (2010)
9. Méry, D., Merz, S.: Event Systems and Access Control. In: Gollmann, D., Jürjens, J. (eds.) *6th Intl. Workshop Issues in the Theory of Security*, Vienna, Austria. IFIP WG 1.7, pp. 40–54. Vienna University of Technology (2006)
10. Molina-Jimenez, C., Shrivastava, S., Solaiman, E., Warne, J.: Run-Time Monitoring and Enforcement of Electronic Contracts. *Electronic Commerce Research and Applications* 3(2), 108–125 (2004)
11. Sieber, J.E.: Data Sharing: Defining Problems and Seeking Solutions. *Law and Human Behaviour* 12(2), 199–206 (1988)
12. Swarup, V., Seligman, L., Rosenthal, A.: A Data Sharing Agreement Framework. In: Bagchi, A., Atluri, V. (eds.) *ICISS 2006*. LNCS, vol. 4332, pp. 22–36. Springer, Heidelberg (2006)

A Logical Framework to Deal with Variability*

Patrizia Asirelli¹, Maurice H. ter Beek¹,
Alessandro Fantechi^{1,2}, and Stefania Gnesi¹

¹ Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, Pisa, Italy
`{asirelli,terbeek,gnesi}@isti.cnr.it`

² Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
`fantechi@dsi.unifi.it`

Abstract. We present a logical framework that is able to deal with variability in product family descriptions. The temporal logic MHML is based on the classical Hennessy–Milner logic with Until and we interpret it over Modal Transition Systems (MTSs). MTSs extend the classical notion of Labelled Transition Systems by distinguishing possible (*may*) and required (*must*) transitions: these two types of transitions are useful to describe variability in behavioural descriptions of product families. This leads to a novel *deontic* interpretation of the classical modal and temporal operators, which allows the expression of both constraints over the products of a family and constraints over their behaviour in a single logical framework. Finally, we sketch model-checking algorithms to verify MHML formulae as well as a way to derive correct products from a product family description.

1 Introduction

Product Line Engineering (PLE) is a paradigm to develop a family of products using a common platform and mass customisation [30,32]. This engineering approach aims to lower production costs of the individual products by letting them share an overall reference model of the product family, while at the same time allowing them to differ with respect to particular characteristics in order to serve, e.g., different markets. As a result, the production process in PLE is organised so as to maximise commonalities of the products and at the same time minimise the cost of variations.

Managing planned *variability* in product families has been the subject of extensive study in the literature on PLE, especially that concerning *feature modelling* [5,13,22], which provides compact representations of all the products of a PL in terms of their features. Variability modelling addresses how to explicitly define the features or components of a product family that are *optional*, *alternative*, or *mandatory*. Formal methods are then developed to show that a certain product belongs to a family, or to derive instead a product from a family, by means of a proper selection of the features or components.

* Research funded by the Italian project D-ASAP (MIUR–PRIN 2007) and by the RSTL project XXL of the CNR.

Many years after their introduction in [26], *Modal Transition Systems* (MTSs) and several variants have been proposed as a formal model for defining product families [1,16,17,19,25,33]. An MTS is a Labelled Transition System (LTS) with a distinction among so-called *may* and *must* transitions, which can be seen as optional or mandatory for the products of the family. Hence, given a family of products, an MTS allows one to model in a single framework:

1. the *underlying architecture*, by means of states and transitions, modelling the product platform shared by all products, and
2. the *variation points*, by means of possible and required transitions, modelling the variability among different products.

Deontic logic [2,29] has recently become popular in computer science for modelling descriptive and behavioural aspects of systems, mainly because of the natural way of formalising concepts like violation, obligation, permission, and prohibition. This makes deontic logic an obvious candidate for expressing the conformance of products of a family with respect to variation points. Such a conformance concerns both *static* requirements, which identify the features that constitute the different products, and *behavioural* requirements, which describe how products differ in their ability to deal with events in time.

Taking into account the Propositional Deontic Logic (PDL) that was proposed in [8,9] and which combines the expression of permission and obligation with concepts from temporal logics, in [3,4] we laid the basis for the application of deontic logic to model variability in product families. We showed how to characterise certain MTSs in terms of deontic logic formulae in [3]. In [4], we presented a first attempt at a logical framework capable of addressing both static and behavioural conformance of products of a family, by defining a deontic extension of an action- and state-based branching-time temporal logic interpreted over so-called doubly-labelled MTSs. Model checking with this logic was left as future work. Modelling and verifying static constraints over the products of a family usually requires separate expressions in a first-order logic [5,18,27], whereas modelling and verifying dynamic behavioural constraints over the products of a family is typically not addressed in feature modelling.

The first contribution of this paper is the introduction of the action-based branching-time temporal logic MHML, which allows expressing both constraints over the products of a family and constraints over their behaviour in a single logical framework. MHML is based on the “Hennessy–Milner logic with Until” defined in [14,24], but it is interpreted over MTSs rather than LTSs. This leads to a novel *deontic* interpretation of the classical modal and temporal operators.

The second contribution is a first step towards a modelling and verification framework based on model-checking techniques for MHML. We do so by providing a global model-checking algorithm to verify MHML formulae over MTSs.

Related Work

In [1,16,17,19,25,33], (variants of) MTSs have been proposed for modelling and verifying the behaviour of product families. We have extended MTSs in [17] to

allow modelling different notions of behavioural variability. A different, algebraic approach to behavioural modelling and verification of product lines instead has been developed in [20,21]. In this paper, we continue research we started in [3,4]. In [3], we showed how to finitely characterise certain MTSS by means of deontic logic formulae. In [4], we presented a first attempt at a logical framework capable of addressing both static and behavioural conformance of products of a family, by defining a deontic extension of an action- and state-based branching-time temporal logic interpreted over so-called doubly-labelled MTSS.

In [12], the authors present a model-checking technique over so-called Featured Transition Systems (FTSs), which are able to describe the combined behaviour of an entire product family. Their main purpose is to provide a means to check that whenever a behavioural property is satisfied by an FTS, then it is also satisfied by every product of the PL, and whenever a property is violated, then not only a counterexample is provided but also the products of the PL that violate the property. The main difference between their approach and ours is our use of a branching-time temporal logic with a deontic flavour that allows us to express and verify in a single framework both behavioural properties and the satisfiability of constraints imposed by features.

Outline

Section 2 contains a simple running example used throughout the paper. After a brief description of feature models in Section 2, we discuss how to use deontic logic to characterise them in Section 4. We introduce the behavioural modelling of product families by means of MTSS in Section 5. In Section 6, we define the temporal logic MHML and show that it can be used to express both static and behavioural requirements of product families. We provide a model-checking algorithm for MHML in Section 7 and we sketch how to use it to derive correct products from a family in Section 8. Section 9 concludes the paper.

2 Running Example: Coffee Machine Product Family

To illustrate the contribution of this paper we consider a family of (simplified) coffee machines as running example, with the following list of requirements:

1. The only accepted coins are the one euro coin (1€), exclusively for European products and the one dollar coin (1\$), exclusively for Canadian products;
2. After inserting a coin, the user has to choose whether (s)he wants sugar, by pressing one of two buttons, after which (s)he may select a beverage;
3. The choice of beverage (coffee, tea, cappuccino) varies for the products. However, delivering coffee is a must for all the family's products, while cappuccino is only offered by European products;
4. After delivering the appropriate beverage, optionally, a ringtone is rung. However, a ringtone must be rung whenever a cappuccino is delivered;
5. The machine returns to its idle state when the cup is taken by the user.

This list contains both static requirements, which identify the features that constitute the different products (see requirements 1, 3 and, partially, 4) and behavioural requirements, which describe the admitted sequences of operations (requirements 2, 5 and, partially, 4).

In the sequel, we will first distill the feature model of this family and provide a formal representation of it in terms of deontic logic formulae. We will then show how the behavioural requirements of this family can be described using an MTS. Finally, we will show how to combine the two approaches by defining a deontic logic framework to check the satisfiability of both static and behavioural requirements over products that should belong to this family.

3 Product Families: Feature Diagrams and Feature Models

Feature diagrams were introduced in [22] as a graphical *and/or* hierarchy of features; the features are represented as the nodes of a tree, with the product family as its root. Features come in several flavours. In this paper, we consider: **optional features** may be present in a product only if their parent is present; **mandatory features** are present in a product if and only if their parent is present; **alternative features** are a set of features among which one and only one is present in a product if their parent is present.

When additional constraints are added to a feature diagram, one obtains a *feature model*. Also these constraints come in several flavours. In this paper we consider:

- requires** is a unidirectional relation between two features indicating that the presence of one feature requires the presence of the other;
- excludes** is a bidirectional relation between two features indicating that the presence of either feature is incompatible with the presence of the other.

An example feature model for the Coffee Machine family of Section 2 is given in Fig. 1; the **requires** constraint obligates feature Ringtone to be present whenever

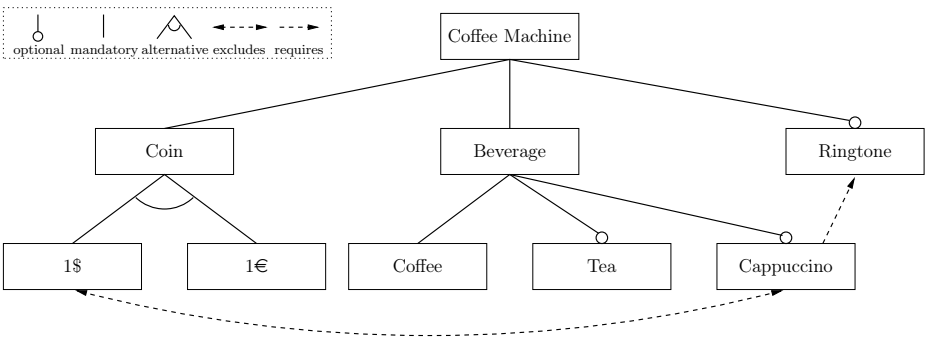


Fig. 1. Feature model of the Coffee Machine family

Cappuccino is and the **excludes** constraint prohibits features 1\$ and Cappuccino to both be present in any product of this family. Obviously, this feature model satisfies the static requirements (i.e. 1, 3 and, part of, 4) of our running example.

4 Deontic Logic Applied to Feature Models

Deontic logic has been an active field of research in computer science for many years now [2,29]. Most deontic logics contain the standard operators of classical propositional logic, i.e. negation (\neg), conjunction (\wedge), disjunction (\vee) and implication (\implies), augmented with deontic operators. In this paper, we consider only two of the most common deontic operators, namely *it is obligatory that* (O) and *it is permitted that* (P), which in the most classical versions of deontic logic enjoy the duality property

$$P(\alpha) = \neg O(\neg\alpha),$$

i.e. something is permitted if and only if its negation is not obligatory.

The way deontic logics formalise concepts such as violation, obligation, permission and prohibition is very useful for system specification, where these concepts arise naturally. In particular, deontic logics seem to be very useful to formalise product family specifications, since they allow one to capture the notions of optional and mandatory features.

4.1 A Deontic Characterisation of Feature Models

In [4], we have presented a deontic characterisation of feature models. Such a characterisation consists of a set of deontic formulae which, taken as a conjunction, precisely characterise the feature model of a product family. If we assume that a name of a feature A is used as the atomic proposition indicating that A is present, then the deontic characterisation is constructed as follows:

- If A is a feature, and A_1 and A_2 are two subfeatures (marked **alternative**, **optional** or **mandatory**), then add the formula $A \implies \Phi(A_1, A_2)$, where $\Phi(A_1, A_2)$ is defined as

$$\Phi(A_1, A_2) = (O(A_1) \vee O(A_2)) \wedge \neg(P(A_1) \wedge P(A_2))$$

if A_1 and A_2 are marked **alternative**, whereas $\Phi(A_1, A_2)$ is otherwise defined as

$$\Phi(A_1, A_2) = \phi(A_1) \wedge \phi(A_2),$$

in which $\phi(A_i)$, for $i \in \{1, 2\}$, is defined as:

$$\phi(A_i) = \begin{cases} P(A_i) & \text{if } A_i \text{ is } \mathbf{optional} \text{ and} \\ O(A_i) & \text{if } A_i \text{ is } \mathbf{mandatory}. \end{cases}$$

Moreover, since the presence of the root feature is taken for granted, the premise of the implication related to that feature can be removed. □

¹ Hence, we tacitly do not deal with trivially inconsistent graphs whose root is involved in an **excludes** relation with a feature.

- If A **requires** B , then add the formula $A \implies O(B)$.
- If A **excludes** B (and hence B **excludes** A), then add the formula $(A \implies \neg P(B)) \wedge (B \implies \neg P(A))$.

This deontic characterisation is a way to provide semantics to feature models. The resulting conjunction of deontic formulae, expressing features and the constraints between them, is called a *characteristic formula* and it can be used to verify whether or not a certain product belongs to a specific family.

5 Behavioural Models for Product Families

In this section we present a behavioural modelling framework able to deal with the variability notions characterising product families at different levels of detail. The underlying model of this framework is a Labelled Transition System (LTS).

Definition 1. A Labelled Transition System (LTS) is a quadruple $(Q, A, \bar{q}, \rightarrow)$, where Q is a set of states, A is a set of actions, $\bar{q} \in Q$ is the initial state, and $\rightarrow \subseteq Q \times A \times Q$ is the transition relation.

If $(q, a, q') \in \rightarrow$, then we also write $q \xrightarrow{a} q'$.

Since we are interested in characterising the dynamic behaviour of product families, we need a notion for the evolution of time in an LTS.

Definition 2. Let $(Q, A, \bar{q}, \rightarrow)$ be an LTS and let $q \in Q$. Then σ is a path from q if $\sigma = q$ (empty path) or σ is a (possibly infinite) sequence $q_1 a_1 q_2 a_2 q_3 \dots$ such that $q_1 = q$ and $q_i \xrightarrow{a_i} q_{i+1}$, for all $i > 0$.

A full path is a path that cannot be extended any further, i.e. which is infinite or ends in a state without outgoing transitions. The set of all full paths from q is denoted by $\text{path}(q)$.

If $\sigma = q_1 a_1 q_2 a_2 q_3 \dots$, then its i -th state q_i is denoted by $\sigma(i)$ and its i -th action a_i is denoted by $\sigma\{i\}$.

When modelling a product family as an LTS, the products of a family are considered to differ with respect to the actions that they are able to perform in any given state. This means that the definition of a family has to accommodate all the possibilities desired for each derivable product, predicating on the choices that make a product belong to the family.

An LTS representing all the possible behaviours conceived for the family of coffee machines described in Section 2 is presented in Fig. 2(a). Note that this LTS cannot distinguish **optional** transitions from **mandatory** ones, since variation points in the family definition are modelled as nondeterministic choices (i.e. alternative paths), independent from the type of variability.

5.1 Modal Transition Systems

To overcome the limitation pointed out earlier of using LTSs as modelling framework for product families, Modal Transition Systems (MTSs) [26] and several variants have been proposed to capture variability in product family specifications [11,16,17,19,25,33].

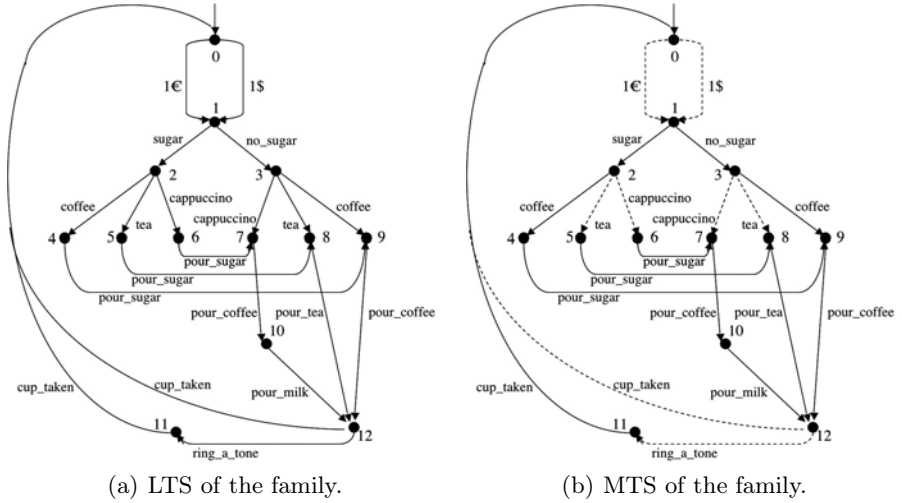


Fig. 2. (a)-(b) Modelling the family of coffee machines

Definition 3. A Modal Transition System (MTS) is a quintuple $(Q, A, \bar{q}, \rightarrow_{\square}, \rightarrow_{\diamond})$ such that $(Q, A, \bar{q}, \rightarrow_{\square} \cup \rightarrow_{\diamond})$ is an LTS, called its underlying LTS.

An MTS has two distinct transition relations: $\rightarrow_{\diamond} \subseteq Q \times A \times Q$ is the may transition relation, which expresses possible transitions, while $\rightarrow_{\square} \subseteq Q \times A \times Q$ is the must transition relation, which expresses required transitions.

By definition, any required transition is also possible, i.e. $\rightarrow_{\square} \subseteq \rightarrow_{\diamond}$.

In an MTS, transitions are either possible (*may*) or required (*must*), corresponding to the notion of **optional** or **mandatory** features in product families. This allows the distinction of a special type of path.

Definition 4. Let $(Q, A, \bar{q}, \rightarrow_{\square}, \rightarrow_{\diamond})$ be an MTS and let $\sigma = q_1 a_1 q_2 a_2 q_3 \dots$ be a full path in its underlying LTS. Then σ is a must path (from q_1) if $q_i \xrightarrow{a_i} \square q_{i+1}$, for all $i > 0$, in the MTS.

The set of all must paths from q_1 is denoted by $\square\text{-path}(q_1)$. A must path σ is denoted by σ^{\square} .

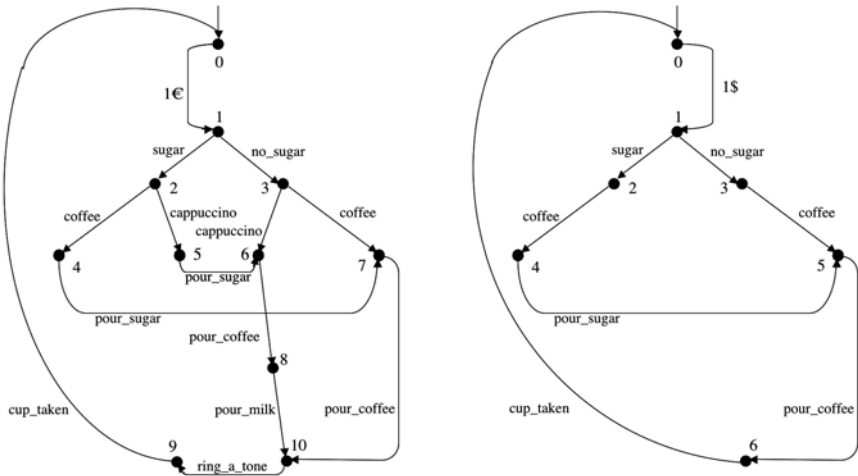
The MTS of Fig. 2(b), in which dashed arcs are used for may transitions and solid ones for must transitions, is another representation of the family of coffee machines described in Section 2. Note that an MTS is able to model the requirements concerning **optional** and **mandatory** characteristics through the use of may and must transitions. However, an MTS is not able to model that the actions 1€ and 1\$ are exclusive (i.e. **alternative** features) nor that the action cappuccino cannot be executed in a European product (as results from the **excludes** relation between the features Cappuccino and 1\$). This will be more clear later, after we define how to generate correct products from an MTS.

Definition 5. Given an MTS $F = (Q, A, \bar{q}, \rightarrow_{\square}, \rightarrow_{\diamond})$ specifying a family, a set of products specified as a set of LTSs $\{P_i = (Q_i, A_i, \bar{q}_i, \rightarrow_i) \mid i > 0\}$ may be consistently derived by considering the transition relation \rightarrow_i to be $\rightarrow_{\square} \cup R$, with $R \subseteq \rightarrow_{\diamond}$, and by pruning all states that are not reachable from \bar{q} .

More precisely, we say that P_i is a product of F , denoted by $P_i \vdash F$, if and only if $\bar{q}_i \vdash \bar{q}$, where $q_i \vdash q$ holds, for some $q_i \in Q_i$ and $q \in Q$, if and only if:

- whenever $q \xrightarrow{a}_{\square} q'$, for some $q' \in Q$, then $\exists q'_i \in Q_i : q_i \xrightarrow{a}_i q'_i$ and $q'_i \vdash q'$, and
- whenever $q_i \xrightarrow{a}_i q'_i$, for some $q'_i \in Q_i$, then $\exists q' \in Q : q \xrightarrow{a}_{\diamond} q'$ and $q'_i \vdash q'$.

Following Def. 5, starting from the MTS of Fig. 2(b), two consistent products can be derived: the coffee machines for the European and Canadian markets shown in Fig. 3. Note, however, that also the coffee machine described by the LTS of Fig. 2(a) can be consistently derived from this MTS. This is the demonstration of the fact that MTSs cannot model constraints in feature models regarding **alternative** features and the **excludes** relation. In fact, the product described by the LTS of Fig. 2(a) violates requirements 1 and 3 (cf. Section 2) by allowing the insertion of both 1€ and 1\$ and at the same time offering cappuccino.



(a) LTS of a European coffee machine. (b) LTS of a Canadian coffee machine.

Fig. 3. (a)-(b) Modelling coffee machines for the European and Canadian markets

6 A Logical Framework for Modelling Variability

In [3], we showed how certain MTSs can be completely characterised by deontic logic formulae and in [4] we presented a first attempt at a logical framework able to address both static and behavioural conformance of products of a family.

In this paper, we further develop that work and define a single logical framework in which to express both the evolution in time and the variability notions considered for product families. To this aim, we define the action-based and branching-time temporal logic MHML based on the “Hennessy–Milner logic with Until” defined in [14,24], but we interpret it over MTSs rather than LTSs. This leads to a deontic interpretation of the classical modal and temporal operators.

With respect to [4], we thus consider an action-based logic rather than an action- and state-based logic, and in Section 7 we will moreover provide model-checking algorithms to verify MHML formulae over MTSs.

6.1 Syntax of MHML

MHML extends the classical Hennessy–Milner logic with Until by taking into account the different type of transitions of an MTS and by incorporating the existential and universal state operators (quantifying over paths) from CTL [10]. As such, MHML is derived from the logics defined in [14,23,24] and it is an action-based variant of the logic proposed in [4].

MHML is a logic of state formulae (denoted by ϕ) and path formulae (denoted by π) defined over a set of atomic actions $A = \{a, b, \dots\}$.

Definition 6. *The syntax of MHML is:*

$$\begin{aligned} \phi &::= \text{true} \mid \neg\phi \mid \phi \wedge \phi' \mid \langle a \rangle\phi \mid [a]\phi \mid E\pi \mid A\pi \\ \pi &::= \phi U \phi' \mid \phi U^\square \phi' \end{aligned}$$

The semantics over MTSs makes MHML incorporate deontic interpretations of the classical modalities. In fact, the informal meaning of the nonstandard operators of MHML is as follows:

- $\langle a \rangle\phi$: a next state exists, reachable by a *must* transition executing action a , in which ϕ holds
- $[a]\phi$: in all next states, reachable by whatever transition executing action a , ϕ holds
- $E\pi$: there exists a full path on which π holds
- $A\pi$: on all possible full paths, π holds
- $\phi U \phi'$: in the current state, or in a future state of a path, ϕ' holds, while ϕ holds in all preceding states of the path (but not necessarily in that state)
- $\phi U^\square \phi'$: in the current state, or in a future state of a path, ϕ' holds, while ϕ holds in all preceding states of the path (but not necessarily in that state), and the path leading to that state is a *must* path

6.2 Semantics of MHML

The formal semantics of MHML is given through an interpretation over the MTSs defined in Section 5.1.

Definition 7. Let $(Q, A, \bar{q}, \rightarrow_{\square}, \rightarrow_{\diamond})$ be an MTS, let $q \in Q$ and let σ be a full path. Then the satisfaction relation \models of MHML over MTSs is defined as follows:

- $q \models \text{true}$ always holds
- $q \models \neg \phi$ iff not $q \models \phi$
- $q \models \phi \wedge \phi'$ iff $q \models \phi$ and $q \models \phi'$
- $q \models \langle a \rangle \phi$ iff $\exists q' \in Q : q \xrightarrow{a}_{\square} q'$ and $q' \models \phi$
- $q \models [a] \phi$ iff $\forall q' \in Q : q \xrightarrow{a}_{\diamond} q'$ and $q' \models \phi$
- $q \models E \pi$ iff $\exists \sigma' \in \text{path}(q) : \sigma' \models \pi$
- $q \models A \pi$ iff $\forall \sigma' \in \text{path}(q) : \sigma' \models \pi$
- $\sigma \models [\phi U \phi']$ iff $\exists j \geq 1 : \sigma(j) \models \phi'$ and $\forall 1 \leq i < j : \sigma(i) \models \phi$
- $\sigma \models [\phi U^{\square} \phi']$ iff $\exists j \geq 1 : \sigma^{\square}(j) \models \phi'$ and $\forall 1 \leq i < j : \sigma^{\square}(i) \models \phi$

The classical duality rule of Hennessy–Milner logic, which states that $\langle a \rangle \phi$ abbreviates $\neg[a]\neg\phi$, does not hold for MHML. In fact, $\neg[a]\neg\phi$ corresponds to a weaker version of the classical diamond operator which we denote as $P(a)\phi$: a next state *may* exist, reachable by executing action a , in which ϕ holds.

A number of further operators can now be derived in the usual way: *false* abbreviates $\neg \text{true}$, $\phi \vee \phi'$ abbreviates $\neg(\neg\phi \wedge \neg\phi')$, $\phi \implies \phi'$ abbreviates $\neg\phi \vee \phi'$. Moreover, $F\phi$ abbreviates $(\text{true} U \phi)$: there exists a future state in which ϕ holds. Likewise, $F^{\square}\phi$ abbreviates $(\text{true} U^{\square} \phi)$: there exists a future state of a must path in which ϕ holds. Finally, $AG\phi$ abbreviates $\neg EF\neg\phi$: in every state on every path, ϕ holds; and $AG^{\square}\phi$ abbreviates $\neg EF^{\square}\neg\phi$: in every state on every must path, ϕ holds.

An illustrative example of a well-formed formula in MHML is thus

$$[a](P(b) \text{true} \wedge (\phi \implies \langle c \rangle \text{true})),$$

which states that after the execution of action a , the system is in a state in which executing action b is *permitted* (in the sense of a *may* transition) and, moreover, whenever formula ϕ holds, then executing action c is *obligatory* (in the sense of a *must* transition). Note that by defining the semantics of MHML over MTSs, we have indeed given a deontic interpretation to the classical box and diamond modalities of Hennessy–Milner logic. In fact, MHML can express both *permitted* and *obligatory* actions (features).

We could of course extend the semantics of MHML formulae to an interpretation over LTSs rather than over MTSs. In that case, since LTSs consist of only *must* transitions, all modalities would need to be interpreted as in the classical Hennessy–Milner logic; this would mean that the weaker version of the diamond operator $P(a)\phi$ in MHML would collapse onto the classical diamond operator $\langle a \rangle \phi$ of Hennessy–Milner logic.

6.3 Expressing Static and Behavioural Requirements

MHML is able to complement the behavioural description of an MTS by expressing constraints over possible products of a family, modelling in this way the static requirements that cannot be expressed in an MTS.

To begin with we consider the following formalisations of the static requirements 1 and 3 (cf. Section 2).

Property A. The actions of inserting 1€ or 1\$ are **alternative** features:

$$(EF \langle 1\$ \rangle true \vee EF \langle 1€ \rangle true) \wedge \neg(EF P(1\$) true \wedge EF P(1€) true)$$

Property B. The action cappuccino cannot be executed in Canadian coffee machines (**excludes** relation between features):

$$\begin{aligned} ((EF \langle cappuccino \rangle true) \implies (AG \neg P(1\$) true)) \wedge \\ ((EF \langle 1\$ \rangle true) \implies (AG \neg P(\langle cappuccino \rangle true))) \end{aligned}$$

These formulae combine static requirements represented by the pure deontic formulae of Section 4.1, through their deontic interpretation in MHML, with behavioural relations among actions expressible by the temporal part of MHML.

Recall that the deontic obligation operator is mapped onto MHML's diamond modality and the deontic permission operator is represented by MHML's weaker version of the classical diamond modality. It is important to note, however, that the classical duality property among the O and P operators of deontic logic is not preserved by this mapping.

To continue, we consider the following formalisation of the static part of requirement 4 (cf. Section 2).

Property C. A ringtone must be rung whenever a cappuccino is delivered:

$$(EF \langle cappuccino \rangle true) \implies (AF \langle ring_a_tone \rangle true)$$

This is an example of a **requires** relation between features. Note that such a static relation between features does not imply any ordering among the related features; e.g., a coffee machine that performs a ringtone before producing a cappuccino cannot be excluded as a product of the family of coffee machines on the basis of this relation. It is the duty of the behavioural description of a product (family) as provided by an LTS (MTS) to impose orderings.

Subsequently, we consider the following formalisation of a further requirement that is particularly interesting for the user of a coffee machine:

Property D. Once the user has selected a coffee, a coffee is eventually delivered:

$$AG [coffee] AF^\square \langle pour_coffee \rangle true$$

7 Model-Checking Algorithms for MHML

The problem model checking aims to solve can be stated as: Given a desired property, expressed as a formula ψ in a certain logic, and a model M , in the form of a transition system, one wants to decide whether $M \models \psi$ holds, where

\models is the logic's satisfaction relation. If $M \not\models \psi$, then it is usually easy to generate a counterexample. If M is finite, model checking reduces to a graph search.

Based on the model-checking parameters M and ψ , different strategies can be pursued when designing a model-checking algorithm. The following global model-checking algorithm extends classical algorithms for the Hennessy–Milner logic and for CTL to MHML [10,11,31]. Actually, the only variation is the distinction of the transition relation (\rightarrow_\diamond or \rightarrow_\square) used in the different cases.

Algorithm 1. A global model-checking algorithm for MHML.

<pre> for all $q \in Q$ do $L(q) := \{true\}$ for $i = 1$ to $length(\psi)$ do for all subformulae ϕ of ψ such that $length(\phi) = i$ do if $\phi = true$ then {nothing to do} else if $\phi = \neg\phi_1$ then for all $q \in Q$ do if $\phi_1 \notin L(q)$ then $L(q) := L(q) \cup \{\phi\}$ else if $\phi = \phi_1 \wedge \phi_2$ then for all $q \in Q$ do if $\phi_1 \in L(q)$ and $\phi_2 \in L(q)$ then $L(q) := L(q) \cup \{\phi\}$ else if $\phi = [a]\phi_1$ then for all $q \in Q$ do if $\forall q': q \xrightarrow{a}_\diamond q', \phi_1 \in L(q')$ then $L(q) := L(q) \cup \{\phi\}$ else if $\phi = \langle a \rangle \phi_1$ then </pre>	<pre> for all $q \in Q$ do if $\exists q': q \xrightarrow{a}_\square q', \phi_1 \in L(q')$ then $L(q) := L(q) \cup \{\phi\}$ else if $\phi = P(a)\phi_1$ then for all $q \in Q$ do if $\exists q': q \xrightarrow{a}_\diamond q', \phi_1 \in L(q')$ then $L(q) := L(q) \cup \{\phi\}$ else if $\phi = E(\phi_1 U^\square \phi_2)$ then $T := \{q \mid \phi_2 \in L(q)\}$ for all $q \in T$ do $L(q) := L(q) \cup \{E(\phi_1 U^\square \phi_2)\}$ while $T \neq \emptyset$ do choose $q \in T$ $T := T \setminus \{q\}$ for all p such that $p \rightarrow_\square q$ do if $E(\phi_1 U^\square \phi_2) \notin L(p)$ and $\phi_1 \in L(p)$ then $L(p) := L(p) \cup \{E(\phi_1 U^\square \phi_2)\}$ $T := T \cup \{p\}$ </pre>
---	---

Algorithm 1 stores in $L(q)$ all subformulae of ψ that are true in q , initially associating *true* to every state and then evaluating subformulae of increasing size on the MTS. Evaluating Until formulae requires another visit of the MTS. The algorithm's complexity is $O(|\psi| \times |Q| \times (|Q| + |\rightarrow_\diamond|))$. A more efficient version uses for Until a depth-first search: its complexity is linear w.r.t. the state space size.

Note that we consider only one of the existential Until operators; other combinations of existential/universal quantification and Until operators can be dealt with similarly. In particular, the procedure for the classical Until operator U can be obtained from that for U^\square by allowing a may transition in its inner for-loop.

Verifying properties A–D on Figs. 2–3 with Algorithm 1 leads to Table 1.

Finally, note the potential for inconsistency: An MTS of a family might not allow any products that satisfy all constraints on features expressed by MHML formulae, i.e. all MHML formulae complementing the behavioural description of an MTS would be false in that MTS. This clearly advocates the usefulness of our model-checking algorithm on MTSs.

Table 1. Results of verifying properties A–D on Figs. 2–3 with Algorithm 1

Property	Fig. 2(a)	Fig. 2(b)	Fig. 3(a)	Fig. 3(b)
A	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
B	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>
C	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>
D	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

8 Towards the Derivation of Correct Products from a Family Description

In Section 5.1, we sketched an algorithm for deriving LTS descriptions of *correct* products from the MTS description of a product family. These products are correct in the sense that they respect the family’s requirements as modelled by the MTS, such as, e.g., the presence of features that are **optional** or **mandatory** but also their behavioural ordering in time. We subsequently presented a relation between LTSs and MTSs to formalise when an LTS *is a product of* a family (specified as an MTS).

In [19], the authors present an algorithm for checking conformance of LTS models against MTS ones according to a given branching relation, i.e. for checking conformance of the behaviour of a product against its product family’s behaviour. It is a fixed-point algorithm that starts with the Cartesian product of the states and iteratively eliminates those pairs that are not valid according to the given relation. They have implemented their algorithm in a tool that allows one to check whether a given LTS conforms to a given MTS according to a number of different branching relations.

Both algorithms allow the derivation of products (specified as LTSs) that are *correct with respect to* the MTS model of a family of products. As we have seen in the previous section, this means that these products might be *incorrect with respect to* the static constraints that cannot be expressed in MTSs, such as, e.g., the presence of features that are **alternative** or part of an **excludes** relation. Since these constraints can be formulated in MHML, we envision an algorithm for deriving correct LTS descriptions of products from an MTS description of a product family *and* an associated set of MHML formulae expressing further constraints for this family. The idea is to prune optional (may) transitions in the MTS in a counterexample-guided way, i.e. based on model-checking techniques.

We informally present our idea by considering as example Property A of Section 6.3, i.e. 1€ and 1\$ are **alternative** features:

$$(EF \langle 1\$ \rangle true \vee EF \langle 1€ \rangle true) \wedge \neg (EF P \langle 1\$ \rangle true \wedge EF P \langle 1€ \rangle true)$$

Model checking this formula over the MTS of Fig. 2(b) returns as counterexample two paths through this MTS, one starting with the 1\$ action and one starting with the 1€ action. Both these actions are optional, which means that two correct products (cf. Fig. 3) can be derived from this MTS: one by pruning the 1\$ action

and one by pruning the $1\in$ action instead. At this point, model checking should be repeated in order to see whether other counterexamples remain (which is not the case for our example).

Based on the principle illustrated by means of this example, an algorithm can be devised in which the conjunction of the constraints is repeatedly model checked, first over the MTS description of the product family and consequently over the resulting (set of) pruned MTSs. These intermediate MTSs are obtained by pruning may transitions in a counterexample-guided way until the formula (conjunction of constraints) is found to be true.

The precise definition of such an algorithm is left for future work, and requires a careful study of the different possible types of constraints and of the effectiveness of the counterexample-guided pruning. After an initial analysis, it seems that local model checking is a more effective strategy to base such an algorithm on, due to its ability to generate counterexample paths early on, without the need of an extensive exploration of the state space. The resulting algorithm would thus allow one to generate all LTSs that satisfy both the family's requirements as modelled by the MTS and its associated set of MHML formulae, i.e. all products that are *correct with respect to* the MTS model of a family of products.

9 Conclusions and Future Work

In this paper we have continued the line of research we initiated in [34] with the following contributions:

1. We have introduced the action-based branching-time temporal logic MHML, which allows the expression of both constraints over the products of a family and constraints over their behaviour in a single logical framework.
2. We have set a first step towards a modelling and verification framework based on model-checking techniques for MHML, by providing a model-checking algorithm to verify MHML formulae over MTSs and by sketching a way to derive correct products from a family description. Both an analysis of the complexity of the model-checking algorithm for MHML and the actual implementation of a model-checking environment for MHML are left as future work.

Such an actual implementation of a model-checking environment for MHML could be an extension of existing model-checking tools, like UMC [6,7,28] or MTSA [15]. UMC is an on-the-fly model checker for UCTL (UML-oriented CTL) formulae over a set of communicating UML state machines. MTSA, built on top of the LTS Analyser LTSA, is a tool that supports the construction, analysis and elaboration of MTSs. To this aim, we can make use of the fact that model checking MTSs is not more complex than model checking LTSs, as checking an MTS can be reduced to two times checking an LTS [19].

The added value of the logical framework we have introduced in this paper can thus be summarised as allowing the possibility to reason, in a single framework, on static *and* dynamic aspects of products of a family. Moreover, from a theoretical point of view, we provide a novel *deontic* interpretation of the classical modal and temporal operators.

Finally, there are a number of aspects of our line of research that require a deeper understanding:

- how to identify classes of properties that, once proved over family descriptions, are preserved by all products of the family;
- how to evaluate quantitative properties, like the number of different possible products of a family;
- how the approach scales to real-world situations in PLE;
- how to hide the complexity of the proposed modelling and verification framework from end users.

References

1. Antonik, A., Huth, M., Larsen, K.G., Nyman, U., Wąsowski, A.: 20 Years of Modal and Mixed Specifications. B. EATCS 95, 94–129 (2008)
2. Åqvist, L.: Deontic Logic. In: Gabbay, D., Guenther, F. (eds.) *Handbook of Philosophical Logic*, 2nd edn., vol. 8, pp. 147–264. Kluwer, Dordrecht (2002)
3. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Deontic Logics for modelling Behavioural Variability. In: Benavides, D., Metzger, A., Eisenecker, U. (eds.) *Proceedings Variability Modelling of Software-intensive Systems (VaMoS 2009)*. ICB Research Report, vol. 29, pp. 71–76. Universität Duisburg, Essen (2009)
4. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A deontic logical framework for modelling product families. In: Benavides, D., Batory, D., Grünbacher, P. (eds.) *Proceedings Variability Modelling of Software-intensive Systems (VaMoS 2010)*. ICB Research Report, vol. 37, pp. 37–44. Universität Duisburg, Essen (2010)
5. Batory, D.: Feature Models, Grammars and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) *SPLC 2005*. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
6. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: An action/state-based model-checking approach for the analysis of communication protocols for service-oriented applications. In: Leue, S., Merino, P. (eds.) *FMICS 2007*. LNCS, vol. 4916, pp. 133–148. Springer, Heidelberg (2008)
7. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. *Sci. Comput. Program.* (to appear 2010)
8. Castro, P.F., Maibaum, T.S.E.: A Complete and Compact Propositional Deontic Logic. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) *ICTAC 2007*. LNCS, vol. 4711, pp. 109–123. Springer, Heidelberg (2007)
9. Castro, P.F., Maibaum, T.S.E.: A Tableaux System for Deontic Action Logic. In: van der Meyden, R., van der Torre, L. (eds.) *DEON 2008*. LNCS (LNAI), vol. 5076, pp. 34–48. Springer, Heidelberg (2008)
10. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8(2), 244–263 (1986)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT, Cambridge (1999)
12. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: *Proceedings 32nd ACM/IEEE International Conference on Software Engineering*, vol. 1, pp. 335–344. ACM, New York (2010)
13. Czarnecki, K., Eisenecker, U.W.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston (2000)
14. De Nicola, R., Vaandrager, F.W.: Three Logics for Branching Bisimulation. *J. ACM* 42(2), 458–487 (1995)

15. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: The Modal Transition System Analyser. In: Proceedings 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 475–476. IEEE, Washington (2008)
16. Fantechi, A., Gnesi, S.: A Behavioural Model for Product Families. In: Crnkovic, I., Bertolino, A. (eds.) Proceedings 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on Foundations of Software Engineering, pp. 521–524. ACM, New York (2007)
17. Fantechi, A., Gnesi, S.: Formal modelling for Product Families Engineering. In: Proceedings 12th International Software Product Line Conference, pp. 193–202. IEEE, Washington (2008)
18. Fantechi, A., Gnesi, S., Lami, G., Nesti, E.: A Methodology for the Derivation and Verification of Use Cases for Product Lines. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 255–265. Springer, Heidelberg (2004)
19. Fischbein, D., Uchitel, S., Braberman, V.A.: A Foundation for Behavioural Conformance in Software Product Line Architectures. In: Hierons, R.M., Muccini, H. (eds.) Proceedings ISSSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, pp. 39–48. ACM, New York (2006)
20. Gruler, A., Leucker, M., Scheidemann, K.D.: Modelling and Model Checking Software Product Lines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 113–131. Springer, Heidelberg (2008)
21. Gruler, A., Leucker, M., Scheidemann, K.D.: Calculating and Modelling Common Parts of Software Product Lines. In: Proceedings 12th International Software Product Line Conference, pp. 203–212. IEEE, Washington (2008)
22. Kang, K., Choen, S., Hess, J., Novak, W., Peterson, S.: Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report SEI-90-TR-21, Carnegie Mellon University (1990)
23. Larsen, K.G.: Modal Specifications. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1989)
24. Larsen, K.G.: Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *Theor. Comput. Sci.* 72(2-3), 265–288 (1990)
25. Larsen, K.G., Nyman, U., Wařowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
26. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: Proceedings 3rd Annual Symposium on Logic in Computer Science, pp. 203–210. IEEE, Washington (1988)
27. Mannion, M., Camara, J.: Theorem Proving for Product Line Model Verification. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 211–224. Springer, Heidelberg (2004)
28. Mazzanti, F.: UMC model checker v3.6 (2009), <http://fmt.isti.cnr.it/umc>
29. Meyer, J.-J.C., Wieringa, R.J. (eds.): *Deontic Logic in Computer Science: Normative System Specification*. John Wiley & Sons, Chichester (1994)
30. Meyer, M.H., Lehnerd, A.P.: *The Power of Product Platforms: Building Value and Cost Leadership*. The Free Press, New York (1997)
31. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model-Checking: A Tutorial Introduction. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 330–354. Springer, Heidelberg (1999)
32. Pohl, K., Böckle, G., van der Linden, F.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Heidelberg (2005)
33. Schmidt, H., Fecher, H.: Comparing disjunctive modal transition systems with an one-selecting variant. *J. Logic Algebraic Program.* 77(1-2), 20–39 (2008)

Adding Change Impact Analysis to the Formal Verification of C Programs

Serge Autexier and Christoph Lüth*

Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI),
Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
{serge.autexier,christoph.lueth}@dfki.de

Abstract. Handling changes to programs and specifications efficiently is a particular challenge in formal software verification. Change impact analysis is an approach to this challenge where the effects of changes made to a document (such as a program or specification) are described in terms of rules on a semantic representation of the document. This allows to describe and delimit the effects of syntactic changes semantically. This paper presents an application of generic change impact analysis to formal software verification, using the GMoC and SAMS tools. We adapt the GMoC tool for generic change impact analysis to the SAMS verification framework for the formal verification of C programs, and show how a few simple rules are sufficient to capture the essence of change management.

1 Introduction

Software verification has come of age, and a lot of viable approaches to verifying the correctness of an implementation with respect to a given specification exist. However, the real challenge in software verification is to cope with changes — real software is never finished, the requirements may change, the implementation may change, or the underlying hardware may change (particularly in the embedded domain, where hardware is a major cost factor). Here, many existing approaches show weaknesses; it is not untypical to handle changes by rerunning the verification and see where it fails (though there exist more sophisticated approaches [15], cf. Sec. 5).

To handle changes efficiently, a verification methodology which supports a notion of modularity is needed, together with the appropriate tool support which allows to efficiently delimit the impact of changes made to source code or specifications, thus leveraging the benefits of modularity. This is known as *change impact analysis*, and this paper describes how to adapt it to formal software verification. We make use of a change impact analysis tool for collections of documents developed by the first author, which supports document-type specific, rule-based semantic annotation and change propagation, and apply it to a framework for formal verification of C programs developed by the second author

* Research supported by BMBF under research grant 01 IW 07002 *FormalSafe*, and DFG under research grant Hu737/3-1 *OMoC*.

and others, which is based on annotating C programs with specifications. Our contribution here is to demonstrate that a generic change impact analysis tool can be adapted quickly to obtain a working solution for management of change in formal verification. We posit that separating change impact analysis from the actual software verification is useful, as it is a sensible separation of concerns and allows the impact analysis to be adapted and reused with other tools. This is a typical situation in verification, where often different tools are used for different aspects of the verification (e.g. abstract interpretation to check safety properties, and a theorem prover for functional correctness).

This paper is structured as follows: we first describe the two frameworks, the generic change impact analysis tool GMoC in Sec. 2 and the SAMS verification framework in Sec. 3, and show how to apply the former to the latter in Sec. 4.

2 Generic Semantic Change Impact Analysis

The motivation to develop a framework for generic semantic change impact is that an overwhelming amount of documents of different types are produced every day, which are rarely isolated artifacts but rather related to other kinds of documents. Examples of documents are filled and signed forms, research reports, or artifacts of the development process, such as requirements, documentation, and in particular specifications and program source code as in our application. These documents evolve over time and there is a need to automate the impact of changes on other parts of documents as well as on related documents. In [4], we proposed the GMoC framework for semantic change impact analysis that embraces existing document types and allows for the declarative specification of semantic annotation and propagation rules inside and across documents of different types. We give a brief and necessarily incomplete overview over the framework here, and refer the reader to [4] for the technical details.

Document Graph Model. The framework assumes documents to be XML files, represented and enriched by semantic information in a single typed graph called the *document graph*. The framework is parametrised over the specific XML format and types for the nodes and vertices of the document graph. The document graph is divided into syntactic and semantic subgraphs; the latter is linked to those syntactic parts which induced them, i.e. their *origins*. As a result the graphs have interesting properties, which can be methodologically exploited for the semantic impact analysis: it may contain parts in the document subgraph for which there exists no semantic counter-part, which can be exploited during the analysis to distinguish added from old parts. Conversely, the semantic subgraph may contain parts which have no syntactic origin, that is they are dangling semantics. This can be exploited during the analysis to distinguish deleted parts of the semantics from preserved parts of the semantics. The information about added and deleted parts in the semantics subgraph is the basis for propagating the effect of changes throughout the semantic subgraph. In order to be able to distinguish semantically meaningful changes, we further define an *equivalence*

```

<guests>
  <person confirmed="true">
    <firstName>Serge</firstName>
    <lastName>Autexier</lastName>
    <email>serge.autexier@dfki.de
  </email>
</person>
  <person confirmed="true">
    <firstName>Normen</firstName>
    <lastName>Müller</lastName>
    <email type="prv">
      n.mueller@gmail.com
    </email>
  </person></guests>

```

```

<guests>
  <person confirmed="true">
    <firstName>Normen</firstName>
    <lastName>Müller</lastName>
    <email type="prv">n.mueller@gmail.com</email>
  </person>
  <person confirmed="false">
    <firstName>Serge</firstName>
    <lastName>Autexier</lastName>
    <birthday>1/23/45</birthday>
    <email type="bus">serge.autexier@dfki.de</email>
  </person></guests>

```

Fig. 1. Semantically Equivalent Guest Lists

relation indicating when two syntactically different XML elements are to be considered equivalent.

Abstraction and Projection Rules. To relate the two subgraphs, we define abstraction rules, which construct the semantic graph representing the semantics of a document, and projection rules, which project semantic properties computed in the semantic graph into impact annotations for the syntactic documents. The rules are defined as graph rewriting rules operating on the document graph.

Change Propagation Rules. The actual change impact is described by propagation rules, which are defined on the semantic entities, taking into account which parts of the semantics have been added and which have been deleted.

Example 1. Consider for instance a wedding planning scenario, where two types of documents occur: The first document is the guest list and the second one the seating arrangement. Both are semi-structured documents in an XML format as shown in Fig. 1; the latter depends on the former with the condition of male and female guests being paired. The change impact we consider is if one guest cancels the invitation the respective change in the guest list ripples through to the seating arrangement breaking the consistency condition of guests being rotationally arranged by gender. The idea of the semantic entities is that the semantic entity for a specific guest remains the same, even if its syntactic structure changes. For instance, the semantic entity of guest *Serge* remains the same even though the subtree is changed by updating the *confirmed* status.

Change Impact Analysis. Change impact analysis starts with taking all documents under consideration, building the syntactic document graph from the XML input, and applying in order the abstraction, propagation, and projection rule sets exhaustively. This initial step semantically annotates a document collection, and can be used to define semantic checks on document collections.

For change impact analysis between two versions of document collections, we then analyse the differences between the source and target documents. In order to ignore syntactic changes which are semantically irrelevant, we use a semantic difference analysis algorithm [13] for XML documents which is parametric in the equivalence relation of the documents. The result of the difference analysis

is an edit script, which is a set of edit operations; the effect of the equivalence models is that in general the edit scripts are less intrusive, thus preserving those parts of the documents from which the semantic parts have been computed. The edit scripts are applied on the syntactic parts of the document graph. As a result now there exists in the document graph new syntactic objects, for which no semantic counterparts exist yet as well as there are semantic parts, which syntactic origins have been deleted. On the graph we execute again the semantic annotation step, which exploits the information on added and deleted parts to compute the impact of the syntactic changes, represented by impact annotations to the documents.

Example 7 (continued). To sharpen our intuition on equivalent XML fragments, let us assume the two `guests` records shown in Fig. 4. Our focus is on the identification of guests. A guest is represented by an unordered `person` with the addition of whether this is committed or cancelled (`confirmed` attribute). Furthermore, information such as first name (`firstName`), surname (`lastName`), and e-mail address is stored. For the latter the distinction is between private address (`prv`) and business address (`bus`) encoded within a (`type`) attribute whereas the type is defaulted to be private. Optionally the date of birth is represented in a `birthday` element. Over time, the entries in the guest list change. Thus, for example, in the bottom half of Fig. 4 the order of guests is permuted and for one `person` element the status of commitment and the e-mail address type changed. In addition, the date of birth has been registered. To identify guests, we are not interested in the status regarding commitment or cancellation and we do not care if we send the invitation to the business address or home address. Therefore, we define the primary key of a guest as a combination of first name, last name and e-mail address, i.e. when comparing two `person` elements, changes in confirmed status or e-mail address type are negligible as well as the existence of birthday information. Existing XML differencing tools, however, would even with an adequate normalisation consider the two guest records to be different. The change of the confirmation status makes the two XML fragments unequal although regarding our primary key definition for `person` elements those two are equal.

Realisation. The semantics-based analysis framework has been realised in the prototype tool GMoC [2] that annotates and analyses the change impacts on collections of XML-documents. The abstraction, propagation and projection rules are graph transformation rules of the graph rewriting tool GrGen [9], which has a declarative syntax to specify typed graphs in so-called *GrGen graph models* as well as GrGen graph rewriting rules and strategies. The GMoC-tool is thus parametrised over a *document model* which contains (i) the GrGen graph model specifying the types of nodes and edges for the syntactic and semantic subgraphs for these documents, as well as the equivalence model for these documents; (ii) the abstraction and projection rules for these documents in GrGen's rule syntax; and (iii) the propagation rules for these documents, written in the GrGen rule syntax as well.

3 Formal Verification of C Programs

In this section, we give a brief exposition of the SAMS verification framework in order to make the paper self-contained and show the basics on which the change management is built in Sec. 4. For technical details, we may refer the interested reader to [10]. The verification framework is based on *annotations*: C functions are annotated with specifications of their behaviour, and the theorem prover Isabelle is used to show that an implementation satisfies the specification annotated to it by breaking down annotations to proof obligations, which are proven interactively. Thus, in a typical verification workflow, we have the role of the implementer, who writes a function, and the verifier, who writes the proofs. Specifications are typically written jointly by implementer and verifier.

Language. We support a subset of the C language given by the MISRA programming guidelines [11]; supported features include a limited form of address arithmetic, arbitrary nesting of structures and arrays, function calls in expressions, and unlimited use of pointers and the address operator; unsupported are function pointers, unstructured control flow elements such as goto, break, and switch, and arbitrary side effects (in particular those where the order would be significant). Programs are deeply embedded into Isabelle and modelled by their abstract syntax, thus there are datatypes representing, *inter alia*, expressions, statements and declaration. (A front-end translates concrete into abstract syntax, see Fig. 3.) The abstract syntax is very close to the phrase structure grammar given in [7 Annex A].

Semantics. The foundation of the verification is a denotational semantics as found e.g. in [15], suitably extended to handle a real programming language, and formalised in Isabelle. It is deterministic and identifies all kinds of faults like invalid memory access, non-termination, or division by zero as complete failure. Specifications are semantically considered to be state predicates, as in a classical total Hoare calculus. A specification of a program p consists of a precondition P and a postcondition Q , both of which are state predicates, and we define that p satisfies this specification if its semantics maps each state satisfying P to one satisfying Q :

$$\Gamma \vdash_s [P] p [Q] \stackrel{\text{def}}{=} \forall S. P S \longrightarrow \text{def}(\llbracket p \rrbracket S) \wedge Q(\llbracket p \rrbracket S) \quad (1)$$

where Γ is the global environment containing variables and the specifications of all the other functions. We can then prove rules for each of the constructors of the abstract syntax as derived theorems. Special care has been taken to keep verification modular, such that each function can be verified separately.

Specification language. Programs are specified through annotations embedded in the source code in specially marked comments (beginning with `/*@`, as in JML or ACSL). This way, annotated programs can be processed by any compiler without modifications. Annotations can occur before function declarations, where they take the form of function specifications, and inside functions to denote loop invariants. A function specification consists of a precondition (`@requires`), a postcondition (`@ensures`), and a modification set (`@modifies`). Fig. 2 gives an

```

/*@
  @requires 0 <= w
            && w < sams_config.brkdist.measurements[0].v
            && brkconfig_OK(sams_config)
  @modifies \nothing
  @ensures 0 < \result
            && \result < sams_config.brkdist.length
            && sams_config.brkdist.measurements[\result-1].v > w
            && w >= sams_config.brkdist.measurements[\result].v
  @*/
Int32 bin_search_idx_v( Float32 w);

```

Fig. 2. Example specification: Given a velocity w , find the largest index i into the global array `sams_config.brkdist.measurements` such that `measurements[i].v` is larger than w . The assumption is that at least the first entry in `measurements` is larger than w , and that the array is ordered (this is specified in the predicate `brkconfig_OK`). The function has no side effects, as specified by the `@modifies \nothing` clause.

example. The state predicates are written in a first-order language into which Isabelle expressions can be embedded seamlessly using a quotation/antiquotation mechanism; the exact syntax can be found in [10]. In contrast to programs, specifications are embedded shallowly as Isabelle functions; there is no abstract syntax modelling specifications.

Tool chain and data flow. The upper part of Fig. 3 shows a graphical representation of the current data flow in the SAMS environment. We start with the annotated source code; it can either be compiled and run, or verified. For each source file, the frontend checks type correctness and compliance with the MISRA programming guidelines, and then generates a representation of the abstract syntax in Isabelle, and for each function in that source file, a stub of the correctness proof (if the file does not exist). The user then has to fill in the correctness proofs; users never look at or edit the program representation. The resulting proof scripts can be run in Isabelle; if Isabelle finishes successfully, we know the program satisfies its specification. Crucially, the verification is modular — the proofs can be completed independently and in any order.

Proving correctness. This is because correctness is proven in a modular fashion, for each function separately. The correctness proof starts with the goal *correct* Θ f , which is unfolded to a statement of the form $\Theta \vdash_s [pre] \textit{body} [post]$, where *body* is the body of the function, and *pre* and *post* the pre- and postconditions, respectively. This goal is reduced by a tactic which repeatedly applies proof rules matching the constructors of the abstract syntax of the *body*, together with tactics to handle state simplification and framing. After all rules have been applied, we are left with *proof obligations* relating to the program domain, which have to be proven interactively in Isabelle. These proofs make up the individual proofs scripts in Fig. 3.

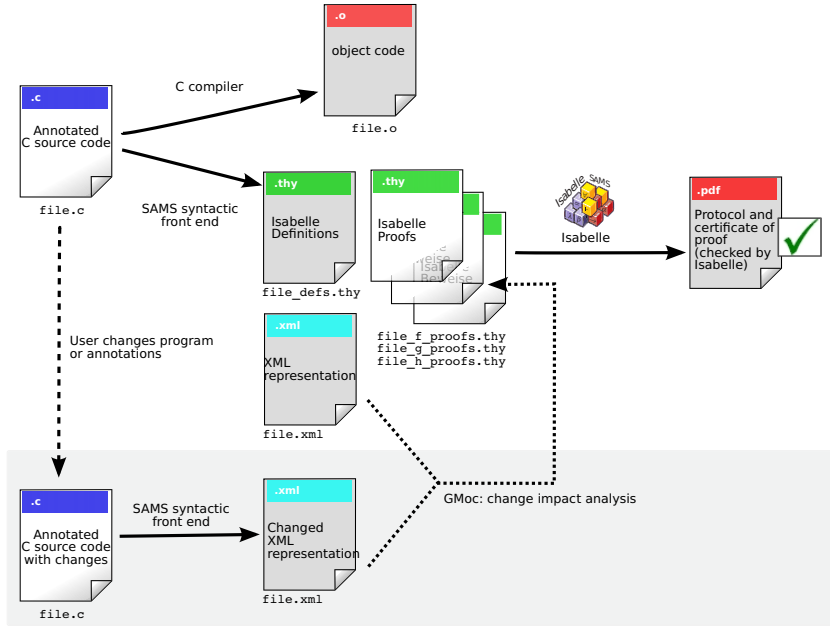


Fig. 3. Dataflow of the verification environment. Grey files are generated, white files are created and edited by the user. The lower part (highlighted in grey) is the added change impact analysis; given a change made by the user, the GMoC tool compares the two XML representations of original and changed copy, and calculates which proof scripts are affected by the change.

The proof scripts can be large, and the whole framework including proof rules and tactics implementing the reduction of correctness assertions to proof obligations is even larger. However, the correctness of the whole verification process reduces to correctness of a very small trusted core, consisting of the denotational semantics of the programming language and the notion of satisfaction from equation (11); the rest is developed conservatively from that. One consequence of this is that when we implement change management as meta-rules, we can never produce wrong results (i.e. a result which erroneously states a program satisfies a specification), as Isabelle is always the final arbiter of correctness.

Change Management. The framework has been developed and used in the SAMS project to verify the correctness of an implementation of a safety function for autonomous vehicles and robots, which dynamically computes safety zones depending on the vehicle’s current speed. The implementation consists of about 6 kloc of annotated C source code, with around fifty functions. What happens now if we change one of these functions or its specification? In the current framework, any changes to the annotated source code require the front-end to be run again, producing a new representation of the program. (The front-end will never overwrite proof scripts.) One then has to rerun the proof scripts, and see where they fail. For large programs, this is not very practical, as proof scripts can run

from several minutes up to an hour. One might break down a source file into smaller parts containing only one function each, but besides not being desirable from a software engineering point of view, this does not help when we change the specification or declaration of a function, which lives in its header file, and is always read by other functions.

In practise, the change impact analysis formalised here is done manually — when a change is made, the verifier reruns those proof scripts which he suspects may break. But this is error prone, and hence we want to make use of change impact analysis techniques, and in particular the GMoC tool, to help us delimiting the effects of changes, pointing us to the proofs which we need to look at. This will extend the current tool chain as depicted in the lower part of Fig. 3.

4 Change Impact Analysis for the SAMS Environment

In order to adapt the generic change impact analysis to the specific SAMS setting, we need to instantiate the generic *document model* of Sec. 2 in three easy steps: (i) define the graph model representing the syntax and semantics of annotated C programs, together with the equivalence relation on the semantics, then (ii) define abstraction and projection rules relating syntax and semantics, and finally (iii) define the rules how changes in annotated programs propagate semantically. We also need to tool the SAMS front-end to generate output of the abstract syntax in the specified XML format, so it can be input to GMoC. All of this has only to be done once and for all, to set up the generic environment. This is because of the basic assumption of the GMoC tool that the underlying document model does not change during the document lifetime (which is the case here, as the syntax and semantics of annotated C programs is fixed).

We now describe the three steps in turn. As a running example we consider the safety function software for autonomous vehicles and robots mentioned in the previous section. In this software one source file contained the functions to compute the configuration of the safety function based on an approximation of the braking distance. It consists of three functions `bin_search_idx_v` called by the function `brkdist_straight` itself called by `compute_brkconfig`. The specification of `bin_search_idx_v` was given in Fig. 2, and the change we consider is that there had been an error in the postcondition saying `@ensures w > sams_config.brkdist.measurements[\result].v` (wrongly using `>` instead of `>=`), and this is now changed in the source code to result in the (correct) specification in Fig. 2. This changes the specification of `bin_search_idx_v`, and as a consequence the proofs of `bin_search_idx_v` and of the calling function `brkdist_straight` must be inspected again, but not the proof of any other function, such as `compute_brkconfig`. Detecting this automatically is the goal of the change impact analysis with GMoC.

4.1 Graph Model of the Documents

The *syntactic* document model is the abstract syntax of the annotated C programs, with nodes corresponding to the types of the non-terminals of the abstract

```

enum CIAStatus {added,deleted,preserved}
node class CIANode extends GmocNode {status:CIAStatus = CIAStatus::added;}
abstract node class Symbol extends CIANode { name : string;}
node class Function extends Symbol {}
node class SemSpec extends CIANode { function : string;}
node class SemBody extends CIANode { function : string;}
edge class Origin extends GmocLink {}
edge class CIALink extends GmocLink { status : CIAStatus = CIAStatus::added;}
edge class Uses extends CIALink {}
edge class IsSpecOf extends CIALink {}
edge class IsBodyOf extends CIALink {}

```

Fig. 4. Semantic GrGen Graph Model for Annotated C Programs

```

equivspec annotatedC {
  element invariant {}
  unordered declaration {constituents = {InitDecl, StorageClassSpec}}
  element InitDecl {constituents = {IdtDecl, FunDecl}}
  element FunDecl {constituents = {IdtDecl}}
  element IdtDecl {constituents = { Identifier }}
  element Identifier {annotations = {id?}} ... }

```

Fig. 5. Part of the equivalence model for the XML syntax of annotated C files

syntax, and edges to the constructors. We use an XML representation of that abstract syntax, and let the SAMS front-end generate that XML representation. The *semantic* entities relevant for change impact analysis of the annotated C programs as sketched above are:

- functions of a specific name and for which we use the node type `Function`;
- the relationship which function calls which other functions directly and for which we have the `Uses` edges among `Function` nodes;
- specifications of functions, for which we use the nodes of type `SemSpec` which are linked to the respective `Function`-nodes by `IsSpecOf` edges; and
- bodies of functions, for which we use nodes of type `SemBody`, are linked to the respective `Function`-nodes by `IsBodyOf` edges.

Every semantic node has at most one link from a syntactic part, its `Origin`. Finally, all semantic nodes and edges have a status attribute indicating whether they are `added`, `deleted`, or `preserved`. Fig. 4 shows the GrGen graph model declaration to encode the intentional semantics of annotated C files.

Next we use the declarative syntax to specify the *equivalence model* for these files, an excerpt of which being shown in Fig. 5 first, the equivalence model specifies to ignore the filename and the file-positions contained as attributes. Thus, for most XML elements, the `annotations`-slot in the entry on the equivalence model does not contain these attribute names. Furthermore, it indicates that declarations are unordered elements and that two declarations are to be

<pre> rule detectNewFunDecls { attr : Attribute - : lsAttribute -> id : Identifier - : isin -> idt:IdtDecl - : isin -> fd:FunDecl - : isin -> d:FunDef; if { attr.name == "id";} negative { id - : Origin -> f:Function; if { f.name == attr.value; }} modify { id - : Origin -> f:Function; exec (findSpec(fd, f)); exec (findFctBody(d, f)); }} </pre>	<pre> rule detectExistingFunDecls { attr : Attribute - : lsAttribute -> id : Identifier - : isin -> idt:IdtDecl - : isin -> fd:FunDecl - : isin -> d:FunDef; if { attr.name == "id";} id - : Origin -> f:Function; if { f.status == CIAStatus::deleted;} if { f.name == attr.value;} modify { eval {f.status=CIAStatus::preserved;} exec (findSpec(fd, f)); exec (findFctBody(d, f)); }} </pre>
--	---

Fig. 6. Rules to detect new and existing functions

considered equal, if the children `Initdecl` and `StorageClassSpec` are equivalent. These equivalence relations specifications are also given and for `Identifiers` the recursion is over the value of the attribute `id`.

For the specifications `spec` and the bodies `FctBody` of functions no entries are specified. As a consequence, they are compared by deep tree equality in the semantic difference analysis and thus every change in these is detected as a replacement of the whole specification (respectively body) by a new version.

4.2 Abstraction and Projection Rules

The *abstraction rules* are sequences of GrGen graph rewriting rules that take the abstract syntax of annotated C programs and compute the semantic representation. In our case these are rules matching function declarations, bodies of functions and specifications of functions and — depending on whether a corresponding semantic entity exists or not — either adjust its status to `preserved` or add a new semantic entity which gets status `added`. Corresponding GrGen rules for these two cases are shown in Fig. 6. Both rules modify the graph structure as specified in the `modify` part and can invoke the call to other rules now concerned with matching the specification and body parts of found functions, which are defined analogously, except that they also add or update the `lsSpecOf` and `lsBodyOf` edges to the given `Function`-node `f` passed as an argument. Finally, there is one rule that establishes the function call relationship between functions by checking the occurrence of functions in function bodies.

Prior to the application of the abstraction rules, the status of all nodes and edges in the semantic graph are set to `deleted`. The adjustment of that information during the abstraction phase for `preserved` parts and setting the status of new semantic objects to `added` results in a semantic graph, where the status of the nodes and edges indicates what happened to these objects. For instance, if the specification of a function `f` has been changed, then the function node `f` has

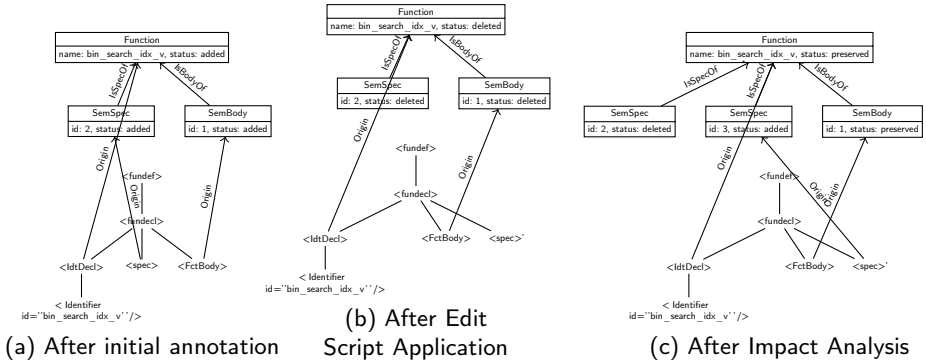


Fig. 7. Phases of the document graph during impact analysis

one `IsSpecOf` edge to an old `SemSpec` node, which has status `deleted`, and one `IsSpecOf` edge to a new `SemSpec` node, which has status `added`.

Example 2. In our running example the abstraction rules detect the functions `bin_search_idx_v`, `brkdlist_straight` and `compute_brkconfig` as well as their corresponding bodies and specifications (see Fig. 7(a) for the part for the function `bin_search_idx_v` only). Finally, they add the `Uses` edges indicating that `bin_search_idx_v` is called by `brkdlist_straight`, which in turn is called by the function `compute_brkconfig` (all not shown in Fig. 7(a)).

The *projection rules* simply project the computed affect information into the syntactic documents by linking the origins of the semantic entities with respective `Impact` nodes. Since we have essentially two kinds of changes causing the re-inspection of proofs, we have two corresponding rules, of which we only present the specification change rule:

```

pattern markSpecCauseProofRequiresInspection (f:Function,cause:Function) {
  id : Identifier ->:Origin-> f;
  negative { if { f.status == CIAStatus::deleted; }
            :fctSpecChangeMarked(f,cause); }
  modify { i:Impact->:affects-> id;
            eval { i.name = "ProofAffected"; i.value = "SpecChange "+cause.name;}}

```

The `Impact` nodes will be automatically serialised into an XML description of the impacts of the changes, reusing the name and value specified in the `impact` node and referencing the XML subtree corresponding to the linked node (e.g., `id:Identifier`) by its XPath in the document.

The final part of the projection phase is to actually delete those nodes and edges in the semantic graph still marked as `deleted`, which is some kind of garbage collection after completing the analysis — and before the next analysis phase starts after application of the syntactic changes on the documents.

4.3 Change Propagation Rules

In the SAMS verification environment, the verification is modular and each C function is verified separately. In this context, changes in the annotated source code affect proofs as follows:

(CIABodyChange). If a function is modified but its specification not changed, then only the proof for this function needs to be checked again.

(CIASpecChange). If the specification of a function f is changed, then the correctness proof of that function needs to be checked as well as the proofs of all functions that directly call f , because in these proofs the specification may have been used.

These are two simple rules that can be checked on the C files directly. In the following we describe how these checks have been automated using the GMoC-tool, presenting the formalisation of the latter in detail. First, we specify a pattern to recognise if the specification of a function has changed based on the fact that we have a function which has not status deleted and which has an `IsSpecOf` edge from an added `SemSpec`:

```

pattern fctSpecChanged(f:Function) {
  negative { if { f.status == CIAStatus::deleted; }}
  newss:SemSpec -:IsSpecOf-> f;
  if { newss.status == CIAStatus::added; }}

```

This pattern is used to detect functions which specifications have changed and mark these and all functions calling them as being affected by the change. This is done by the sub-rules `markSpecCauseProofRequiresInspection` and `markCallingFunctionsProofInspection` in the pattern below, where the keyword **iterated** indicates that the enclosed pattern must be applied exhaustively:

```

pattern propagateChangedSpec {
  iterated { f:Function;
    :fctSpecChanged(f);
    markself:markSpecCauseProofRequiresInspection(f,f);
    markothers:markCallingFunctionsProofInspection (f);
    modify { markself(); markothers(); }}
  modify {} }

```

A similar but simpler propagation rule exists to detect changed function bodies which marks only the function itself, but not the calling functions.

4.4 Change Impact Analysis

For the change impact analysis, the original annotated C programs are semantically annotated using the abstraction, propagation and projection rules from the previous section. Before analysing the changes caused by a new version of the annotated C programs, all impact nodes are deleted, because they encode the impacts of going from empty annotated C programs to the current versions. This is the equivalent to an adjustment of the baseline in systems like DOORS.

Next the differences to the new versions of the annotated C programs are analysed using the semantic difference analysis instantiated with the equivalence model for annotated C programs. This model is designed such that all type, constant and function declarations are determined by the name of the defined objects and any change in a specification or function body causes its entire replacement. Applying the resulting edit script thus deletes the node which was marked as the origin of a specification (resp. body, function, type or constant), and thus the corresponding node in the semantic graph becomes an orphan.

Example 2 (continued). For our example, the change in the specification of `bin_search_idx_v` from `>` to `>=` is detected by the semantic difference analysis and due to the way the equivalence model is defined, the edit script contains the edit operation to remove the entire specification and replace it with a new one. Applying that on the SDI graph deletes the old specification node in the document graph and adds a new specification to the document graph. Thus, in the semantic graph the `SemSpec` node for the function `bin_search_idx_v` now lacks an `Origin`-node. The shape of the graph now including deletion information on the semantic parts and the removed/added syntactic parts from the edit script is shown in Fig. 7(b).

On that graph the same abstraction, propagation and projection rules are applied again. All rules are change-aware in the sense that they take into account the status (**added**, **deleted**, or **preserved**) of existing nodes and edges in the semantic graph and adapt it to the new syntactic situation. As a result we obtain **Impact**-nodes marking those functions, which proofs must be inspected again according to the rules (**CIABodyChange**) and (**CIASpecChange**).

More specifically, for our example C files, the abstraction-phase introduces a new `SemSpec`, and hence the `Function`-node for `bin_search_idx_v` in the semantic graph now has one `IsSpecOf`-edge to the old `SemSpec`-node (i.e. with status **deleted**) and one to the `SemSpec`-node which has just been added (see Fig. 7(c)). This is exploited by the propagation rules to mark `bin_search_idx_v` and `brkdist_straight` as those functions which proofs need inspection.

5 Conclusions, Related and Future Work

This paper has shown how change impact analysis can be used to handle changes in formal software verification. Our case study was to adapt the generic GMoC tool to the change impact analysis for annotated C programs in the SAMS verification framework, and demonstrate its usage with source code developed in the SAMS project. Currently, prototypes of both tools are available at their respective websites [2,14].

Results. The case study has shown that the principles of the explicit semantic method underlying the GMoC framework indeed allow to add change impact analysis to an existing verification environment. Note that we have not shown all rules above, but only those of semantic importance. As far as the C language is concerned, further impact analysis is unlikely to be much of an improvement, as functions are the appropriate level of abstraction. One could consider analysing

the impact of changing type definitions, but because type checking takes place before the change impact analysis, this is covered by the rules described above: if we change the definition of a type, then in order to have the resulting code type-check, one will typically need to make changes in the function definitions and declarations using the type, which will in turn trigger the existing rules.

On a more technical note, the case study also showed that the current prototype implementation of GMoC does not scale well for large XML files, because interaction between GMoC, implemented in Java, and GrGen, based on .NET, is currently based on exchanging files. Thus, future work will consist of moving to an API based communication between GrGen and GMoC.

Related work. Other formal verification tools typically do not have much management of change. For example, two frameworks which are close to the SAMS framework considered here are Frama-C [8] and VCC [6], both of which use C source code annotated with specifications and correctness assertions, and both of which handle changes by rerunning the proofs. Theorem provers like Isabelle and Coq (which is used with Frama-C) have very course-grained change management at the level of files (i.e. if a source file changes, all definitions and proofs in that file and all other files using this, are invalidated and need to be rerun). Some formal methods tools, such as Rodin [11] and KIV [5], have sophisticated change management, which for these tools is more powerful than what a generic solution might achieve, but the separation advocated here has three distinct advantages which we believe outweigh the drawbacks: (i) it makes change impact analysis (CIA) reusable with other systems (e.g. the SAMS instantiation could be reused nearly as is with Frama-C); (ii) it allows experimentation with different CIA algorithms (amounting to changing the rules of the document model); and (iii) it allows development of the verification system to be separated from development of the change management, and in particular allows the use of third-party tools (such as Isabelle in our case) for verification. In previous work the first author co-developed the MAYA system [3] to maintain structured specifications based on development graphs and where change management support was integrated from the beginning. These experiences went into the development of GMoC which is currently also used to provide change impact analysis for the Hets tool [12] where change management was not included right from the beginning.

Outlook. The logical next step in this development would be change impact analysis for Isabelle theories, to allow a finer grained change management of the resulting theories. As opposed to the situation in C, fine-grained analysis in Isabelle makes sense, because in general proof scripts are much more interlinked than program source code, and because they take far longer time to process. However, for precisely this reason it requires a richer semantic model than C.

The change impact analysis could then be used for the Isabelle proof scripts occurring in the SAMS framework, for standalone Isabelle theories, or for Isabelle proof scripts used in other tools. This demonstrates that it is useful to keep change impact analysis separate from the actual tools, as it allows its reuse, both across different versions of the same tool (this is particularly relevant when tools are still under active development, which is often the case in academic

environment), or when combining different tools (a situation occurring quite frequently in software verification). Thus, we could have implemented the rules above straight into the frontend with not much effort, but that would still leave the necessity to handle changes for the Isabelle theories.

On a more speculative note, we would like to extend the change impact analysis to handle typical re-factoring operations (both for C, and even more speculative, for Isabelle), such as renaming a function or parameter (which is straightforward), or e.g. adding a field to a structure type t ; the latter should not impact any correctness proofs using t except those calling `sizeof` for t .

References

1. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Metha, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)* (2010)
2. Autexier, S.: The GMoC Tool for Generic Management of Change (2010), <http://www.informatik.uni-bremen.de/dfki-sks/omoc/gmoc.html>
3. Autexier, S., Hutter, D.: Formal software development in Maya. In: Hutter, D., Stephan, W. (eds.) *Mechanizing Mathematical Reasoning*. LNCS (LNAI), vol. 2605, pp. 407–432. Springer, Heidelberg (2005)
4. Autexier, S., Müller, N.: Semantics-based change impact analysis for heterogeneous collections of documents. In: *Proc. 10th ACM Symposium on Document Engineering (DocEng 2010)*, UK (2010)
5. Balsler, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) *FASE 2000*. LNCS, vol. 1783, pp. 363–366. Springer, Heidelberg (2000)
6. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *Theorem Proving in Higher Order Logics*. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
7. *Programming languages — C*. ISO/IEC Standard 9899:1999(E), 2nd edn. (1999)
8. Frama-C, <http://frama-c.cea.fr/> (2008)
9. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A fast SPO-based graph rewriting tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) *ICGT 2006*. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
10. Lüth, C., Walter, D.: Certifiable specification and verification of C programs. In: Cavalcanti, A., Dams, D.R. (eds.) *FM 2009*. LNCS, vol. 5850, pp. 419–434. Springer, Heidelberg (2009)
11. MISRA-C: 2004. Guidelines for the use of the C language in critical systems. MISRA Ltd. (2004)
12. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
13. Müller, N.: Change Management on Semi-Structured Documents. PhD thesis, School of Engineering & Science, Jacobs University Bremen (2010)
14. Safety Component for Autonomous Mobile Service Robots, SAMS (2010), <http://www.sams-project.org/>
15. Winskel, G.: *The Formal Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press, Cambridge (1993)

Creating Sequential Programs from Event-B Models

Pontus Boström

Åbo Akademi University, Department of Information Technology
Joukahaisenkatu 3-5, 20520 Turku, Finland
`pontus.bostrom@abo.fi`

Abstract. Event-B is an emerging formal method with good tool support for various kinds of system modelling. However, the control flow in Event-B consists only of non-deterministic choice of enabled events. In many applications, notably in sequential program construction, more elaborate control flow mechanisms would be convenient. This paper explores a method, based on a scheduling language, for describing the flow of control. The aim is to be able to express schedules of events; to reason about their correctness; to create and verify patterns for introducing correct control flow. The conclusion is that using patterns, it is feasible to derive efficient sequential programs from event-based specifications in many cases.

1 Introduction

Event-B [1] has emerged as a well known method for high-level specification of a wide variety of different types of systems. It has good tool support in the form of theorem provers [1], animators [2] and model checkers [2] to analyse specifications. The specifications can also be developed and analysed in a stepwise manner using the notion of refinement. However, the control flow in Event-B is very simplistic, with only non-deterministic choice of enabled events. This can be inconvenient especially when developing software.

Conceptually an Event-B model consists of a set of events inside a loop, where one enabled event is non-deterministically chosen for execution in each iteration of the loop. If there are no enabled events, the loop terminates. However, many problems could be expressed more conveniently if more expressive control flow constructs were allowed. Two different approaches to do this are already provided in [3,4]. Here a similar approach is developed, but based on the well-known theory for set-transformers. The benefit of using set-transformers is that they provide a mature theory that supports a refinement notion compatible with the one in Event-B. Furthermore, powerful algebraic rules for high-level manipulation of set-transformers have been developed [5]. For implementation of Event-B models, Abrial has developed patterns to translate them to an (imperative) programming language. In [6,7], Abrial gives a method for development of sequential programs in Event-B and patterns for introducing control flow constructs, such as while-loops and if-statements, to obtain the final imperative program. Those patterns

work well in many cases, but they are fairly limited in the type of sequential behaviour that can be introduced. An alternative approach to development of sequential programs in Event-B can also be found in [8].

This paper proposes a method to introduce control flow to Event-B models. The main goal is creation of sequential programs. By creation of a sequential program, we mean that control flow constructs such as while-loops, if-statements and sequential composition are introduced to schedule the events from the Event-B model. The methods are based on using a scheduling language to describe the flow of control. The main goal is to show how to verify scheduling of events, as well as to develop and reason about reusable patterns for introducing control flow constructs in practice. To illustrate the approach, patterns for scheduling events are presented and applied on an example. A more thorough presentation of sequential program development using an earlier version of this method can be found in [9].

The paper starts with an overview of Event-B. Event-B has no fixed semantics and we first present a semantics based on set transformers similar to the one in [10] and action systems. This semantics is compatible with the proof obligations generated by the Event-B tools. To create sequential programs from the Event-B models, a scheduling language is then developed. We then show how scheduling events according to the schedule will lead to a refinement of the original model. Patterns for verified schedules, as well as creation of executable program statements from the scheduled events are also presented.

2 Event-B

An Event-B model is referred to as a *machine* [1]. A machine consists of a set of *variables* constituting the state of the model, an *invariant* that describes the valid states and a collection of *events* that describes the evolution of the state of the model. The model can also use a *context machine*, with *constant* and *set* definitions. The properties of these constants and sets are given by a set of *axioms*.

2.1 Events as Set Transformers

The events in Event-B can be viewed as set transformers [11,10]. Our presentation of events as set transformers is similar to the presentation in [10].

The state space of an Event-B model is modelled as the Cartesian product of the types of the variables. Variables v_1, \dots, v_n having types $\Sigma_1, \dots, \Sigma_n$ give the state space $\Sigma = \Sigma_1 \times \dots \times \Sigma_n$. An event E has the form $E \hat{=} \mathbf{when} G(v, c) \mathbf{ then } v : |S(v', v, c) \mathbf{ end}$. Here v denotes the variables, c the constants, $G(v, c)$ denotes the guard of the event and $v : |S(v', v, c)$ denotes a non-deterministic assignment to the variables. Whenever $G(v, c)$ is true the substitution $v : |S(v', v, c)$ can be executed. The substitution assigns variables v any value v' such that S holds. The events of the machine are then considered to be inside a loop, where one event is executed in each iteration of the loop as long as there are enabled events.

To describe the valid subset of the state space, an invariant I is used. A model has a special initialisation event *initialisation*, which consists of a substitution of the form $v : |A(v', c)$. We have the following definitions:

$$\begin{aligned}
 \Sigma &= \{v | \top\} \\
 i &= \{v | I(v, c)\} \\
 g &= \{v | G(v, c)\} \\
 s &= \{v \mapsto v' | S(v, v', c)\} \\
 a &= \{v' | A(v', c)\}
 \end{aligned} \tag{1}$$

The sets i and g describe the subsets of the state space where the invariant I and the guard G holds, respectively. The relation s describes the possible before-after states that can be achieved by the assignment. Note that the initialisation results in a set a instead of a relation, since it does not depend on the previous values of the variables. In this paper, we do not consider properties of constants c separately, as it is not important on this level of reasoning. The axioms that describe the properties of the constants are here considered to be part of the invariant.

To simplify the definitions and increase readability, we introduce the following notation. The symbol `true` will be used to denote the complete state space of an Event-B model. In the above definition we have that $\text{true}_\Sigma = \Sigma$. We will use negation to denote the complement of a set $\neg q \hat{=} \text{true} \setminus q$ ¹. For completeness `false` is defined as $\text{false} \hat{=} \emptyset$. This is similar to the approach in [11].

We give the semantics of events as set transformers. A set transformer applied to a set q describes the set of states from where the set transformer will reach a state in q (cf. weakest precondition semantics of programs). We have the following set transformers:

$$[g](q) \hat{=} \neg g \cup q \quad (\text{Assumption}) \tag{2}$$

$$\{g\}(q) \hat{=} g \cap q \quad (\text{Assertion}) \tag{3}$$

$$[s](q) \hat{=} \{v | s[\{v\}] \subseteq q\} \quad (\text{Non - deterministic update}) \tag{4}$$

$$(S_1 \sqcap S_2)(q) \hat{=} S_1(q) \cap S_2(q) \quad (\text{Non - deterministic choice}) \tag{5}$$

$$S_1; S_2(q) \hat{=} S_1(S_2(q)) \quad (\text{Sequential composition}) \tag{6}$$

$$S^\omega(q) \hat{=} \mu X. (S; X \sqcap \text{skip})(q) \quad (\text{Strong iteration}) \tag{7}$$

$$S^*(q) \hat{=} \nu X. (S; X \sqcap \text{skip})(q) \quad (\text{Weak iteration}) \tag{8}$$

$$\text{skip}(q) \hat{=} q \quad (\text{Skip}) \tag{9}$$

$$\text{magic}(q) \hat{=} [\text{false}](q) \quad (\text{Magic}) \tag{10}$$

$$\text{abort}(q) \hat{=} \{\text{false}\}(q) \quad (\text{Abort}) \tag{11}$$

The sequential composition ; has higher precedence than \sqcap . The set transformers here are similar to the predicate transformers in e.g. [11]. The first four set transformers (2)-(5) also occur in [10]. The two set transformers (7) and (8) model iteration of a set transformer S using the least and greatest fixpoint

¹ \setminus denotes set subtraction.

[\[11.5\]](#), respectively. The three last set transformers are important special cases: `skip` is a set transformer that does nothing, `magic(q)` is always true and `abort(q)` is always false.

Let g and h denote two sets. We have the following rules that will be useful later:

$$\begin{array}{ll}
 \{g\}; \{h\} = \{g \cap h\} & [g]; [h] = [g \cap h] \\
 \{g\} \sqcap \{h\} = \{g \cap h\} & [g] \sqcap [h] = [g \cup h] \\
 \text{abort} \sqcap S = \text{abort} & \text{magic} \sqcap S = S \\
 \text{abort}; S = \text{abort} & \text{magic}; S = \text{magic}
 \end{array} \tag{12}$$

These rules are easy to prove directly from the definitions and proofs can also be found in [\[11\]](#).

The guard g ([\[13\]](#)) of a set transformer denotes the states where it will not behave miraculously (establish false) and the termination guard t ([\[14\]](#)) denotes the states where it will terminate properly (reach a state in the state space).

$$g(S) \hat{=} \neg S(\text{false}) \tag{13}$$

$$t(S) \hat{=} S(\text{true}) \tag{14}$$

These functions have the following useful properties:

$$\begin{array}{ll}
 g(\{g\}) = \text{true} & g([g]) = g \\
 t(\{g\}) = g & t([g]) = \text{true} \\
 t(S_1 \sqcap S_2) = t(S_1) \cap t(S_2) & g(S_1 \sqcap S_2) = g(S_1) \cup g(S_2)
 \end{array} \tag{15}$$

The proofs are again easy to do directly from the definitions.

Using the definitions in [\(1\)](#) and the set transformers in [\(2\)](#) and [\(4\)](#), an event $E \hat{=} \mathbf{when} \ G(v, c) \ \mathbf{then} \ v : |S(v', v, c) \ \mathbf{end}$ can be modelled as the set transformer $[E] \hat{=} [g]; [s]$ (see [\[10\]](#)). Note that the proof obligations for Event-B guarantee that $g([E]) = g$ and $t([E]) = \text{true}$ for all events [\[10\]](#). The initialisation event *initialisation* is modelled by the set transformer $[initialisation] \hat{=} [a]$.

An Event-B model can be modelled as an action system [\[2,13\]](#). The reason for introducing action systems is that we can use the algebraic transformation rules that have been developed for them [\[5\]](#). Like an Event-B model, an action system consists of a list of variables, an initialisation of the variables and a set of actions given as set transformers. The variables form the state of the system. The actions are iterated inside a **do od**-loop and thus they describe the evolution of the system. Consider an Event-B model \mathcal{M} with variables v , initialisation *initialisation* and events E_1, \dots, E_n . The corresponding action system \mathcal{M}_A is then:

$$\mathcal{M}_A \hat{=} [[\mathbf{var} \ v; \ \mathbf{init} \ [initialisation]; \ \mathbf{do} \ [E_1] \sqcap \dots \sqcap [E_n] \ \mathbf{od}]] \tag{16}$$

The action system \mathcal{M}_A has a list of variables v corresponding to the variables in the Event-B model. The initialisation event, *initialisation*, of the Event-B model becomes the initialisation of the action system. The **do od**-loop denotes non-deterministic execution of the events $[E_1] \sqcap \dots \sqcap [E_n]$. In each execution

of the loop body, one event is non-deterministically chosen. In terms of the set transformers presented earlier, the semantics of the loop is defined as [5]:

$$\mathbf{do} S \mathbf{od} \hat{=} S^\omega; [\neg \mathbf{g}(S)] \quad (17)$$

This means S is executed zero or more times. The assumption at the end ensures that the loop does not terminate before the guard of S is false.

2.2 Refinement

Data refinement is a key concept when developing models in Event-B [1]. This topic has been discussed elsewhere in detail [5,11,10]. Here we only need *algorithmic refinement*, which can be seen as a special case of data refinement and is defined as [11]:

$$S \sqsubseteq R \hat{=} \forall s. S(s) \subseteq R(s) \quad (18)$$

Here S and R are set transformers. Intuitively $S \sqsubseteq R$ means that if S will reach a state in a set s then so will R . Collections of refinement rules for statements can be found in e.g. [11,5]. All refinement rules used in this paper that have no explicit reference can be found in [11].

3 Introduction of Control Flow by Scheduling Events

We are interested in introducing more precise control flow constructs to Event-B models to e.g. create efficient sequential programs. This means that we create a schedule for the events which they should be executed according to. Here the control flow is introduced as a refinement of the underlying Event-B model. It could also be introduced as a basic modelling construct as in [3,4]. The approach in this paper has the advantage of not requiring changes to the basic modelling notation and tools, only new tool support for the final scheduling step is needed. In order for the schedule to be correct, execution of the events according to it should lead to a refinement of the original loop of events. First we describe the scheduling language. Based on this language, the events from the Event-B model are scheduled to obtain a sequential statement. We then show how to prove the correctness of this statement and present two reusable patterns for scheduling.

3.1 The Scheduling Language

First we introduce a scheduling language to describe the scheduling of events. It has the following grammar:

$$\begin{aligned} DoStmnt &::= \mathbf{do} Stmnt \mathbf{od} \\ ChoiceStmnt &::= E_1 \parallel \dots \parallel E_n \\ Stmnt &::= [\{g\} \rightarrow] (DoStmnt \mid ChoiceStmnt) [\rightarrow Stmnt] \end{aligned} \quad (19)$$

Here $[exp]$ optional occurrence of exp , g a predicate and E an event name. We can have sequential composition of events \rightarrow , choice of events \parallel and iteration of

events **do od** . We can also have assertions $\{g\}$ before a choice or a loop. This is often needed in the proofs of correctness. The language is not very flexible. However, the aim is that it should be possible to automatically generate proof obligations for schedule correctness directly from the events and the assertions given in the schedule. Furthermore, the statement obtained after scheduling should easily map to an imperative programming language.

The correctness criteria for the statement obtained by scheduling the events is that it refines the original loop of events (see (16)). The scheduled statement should also be non-miraculous, in order for the statement to be implementable. These properties can be difficult to verify in one step and the proof of correctness would probably not be reusable. The verification of the scheduling is therefore here done inductively in a top-down manner. Assume that we have a recursive decent parser `sched` of schedules that conform to the grammar in (19), which also acts as one-pass compiler from schedule to set-transformer. We show that each recursive call to `sched` will create a refinement of the previous call. Due to monotonicity of all set transformers involved, this will ensure that at the end the final statement obtained from the schedule refines the original Event-B model. This type of stepwise verification of scheduling seems to be well suited for creation and verification of reusable patterns for introducing different types of control flow. The creation of sequential programs in (6.7) is done in a more bottom-up manner, where individual events are merged to form larger units. However, there are often side-conditions for the scheduling patterns (see pattern (25) later) concerning non-interference of the rest of the system. In our approach those conditions are explicitly taken into account, which might be more difficult to do in a bottom-up approach.

3.2 Verification of Scheduling

Let the schedule compiler function $\text{sched}(S)$ be a function from a schedule S to a set transformer. Below E denotes a set of events. E.g. the set E is assumed to consist of events E_1, \dots, E_n and thus $\|_i E_i$ denotes $E_1 \parallel \dots \parallel E_n$. Here we use the notation $[[\|_i E_i]]$ to mean the demonic choice of all events in E , $[[\|_i E_i]] \hat{=} [E_1] \sqcap \dots \sqcap [E_n]$. Furthermore, we need a function $e(S)$ that returns the set of events mentioned in the schedule S . The function `sched` is recursively defined as follows:

$$\begin{aligned}
 \text{sched}(\{g\} \rightarrow S) &\longrightarrow \{g\}; \text{sched}(S) \\
 \text{sched}(\|_i E_i \rightarrow \{g\} \rightarrow S) &\longrightarrow [[\|_i E_i]]; \{g\}; \text{sched}(S) \\
 \text{sched}(\mathbf{do} S_1 \mathbf{od} \rightarrow \{g\} \rightarrow S) &\longrightarrow \\
 &([\mathbf{g}([\|_i e(S_1)])]; \text{sched}(S_1))^\omega; [\neg \mathbf{g}([\|_i e(S_1)])]; \{g\}; \text{sched}(S) \\
 \text{sched}(\langle \rangle) &\longrightarrow \text{skip}
 \end{aligned} \tag{20}$$

To emphasize the direction of function application, we have used \longrightarrow instead of $=$ for definition of the function `sched`. We have also here omitted the case when the assertion $\{g\}$ is not present, since it can be handled using the identity $\{\text{true}\} = \text{skip}$. The empty schedule $\langle \rangle$ is given the meaning `skip`. To verify that the

events can be scheduled according to the desired schedule, each application of the scheduling function should lead to a refinement of the previous step. The function application $\text{sched}(S)$ above refers to events that have not been scheduled yet and its semantics is therefore $[[\!|_i e(S)]^\omega; [\neg g(\llbracket\!|_i e(S)\rrbracket)]]$. Hence, we prove that the left-hand side is always refined by the right-hand side. Furthermore, it might also be desirable to prove that the right-hand side is non-miraculous. For simplicity, from here on we directly denote the set transformer $[[\!|_i E]$ corresponding to the choice of events E with only E .

Note that on the right-hand side, scheduling statements of the form $\text{sched}(S_i \rightarrow S)$ are always preceded by an assertion. This means that we can usually do the desired refinement proofs in a *context* \square where a context assertion holds. In the proof obligation below we assume that all applications of sched in (20) are done in a context, where the assertion $\{c\}$ holds. We then have refinement conditions of the form $\{c\}; \text{sched}(S_i \rightarrow S) \sqsubseteq \dots$

The proof obligations for the scheduling using sched from (20) become:

$$\{c\}; e(S)^\omega; [\neg g(e(S))] \sqsubseteq \{g\}; e(S)^\omega; [\neg g(e(S))] \quad (21)$$

$$\{c\}; (e(S) \sqcap E_1)^\omega; [\neg g(e(S) \sqcap E_1)] \sqsubseteq E_1; \{g\}; e(S)^\omega; [\neg g(e(S))] \quad (22)$$

$$\{c\}; (e(S) \sqcap e(S_1))^\omega; [\neg g(e(S) \sqcap e(S_1))] \sqsubseteq \{g\}; (\llbracket g(e(S_1)) \rrbracket); e(S)^\omega; [\neg g(e(S))]^\omega; [\neg g(e(S_1))]; \{g\}; e(S)^\omega; [\neg g(e(S))] \quad (23)$$

The statement $\text{sched}(S)$ is interpreted as a still unscheduled loop and thus its semantics is given as $e(S)^\omega; [\neg g(e(S))]$. To handle the complexity of the statement $(\llbracket g(e(S_1)) \rrbracket); e(S_1)^\omega; [\neg g(e(S_1))]^\omega; [\neg g(e(S_1))]$, Lemma 1 can then be used.

Lemma 1. $T^\omega; [\neg g(T)] = (\llbracket g(T) \rrbracket); T^\omega; [\neg g(T)]^\omega; [\neg g(T)]$

Proof.

$$\begin{aligned} & (\llbracket g(T) \rrbracket); T^\omega; [\neg g(T)]^\omega; [\neg g(T)] \\ &= \{\text{Unfolding [5] : } T^\omega = T; T^\omega \sqcap \text{skip}\} \\ & \quad (\llbracket g(T) \rrbracket); T^\omega; [\neg g(T)]; (\llbracket g(T) \rrbracket); T^\omega; [\neg g(T)]^\omega \sqcap \text{skip}; [\neg g(T)] \\ &= \{\text{Leapfrog (Lemma 11 in [5]) : } S; (T; S)^\omega = (S; T)^\omega; S\} \\ & \quad (\llbracket g(T) \rrbracket); T^\omega; ([\neg g(T)]; \llbracket g(T) \rrbracket); T^\omega; [\neg g(T)] \sqcap \text{skip}; [\neg g(T)] \\ &= \{[\neg g(T)]; \llbracket g(T) \rrbracket = \text{magic and magic}; T = \text{magic and magic}^\omega = \text{skip}\} \\ & \quad (\llbracket g(T) \rrbracket); T^\omega; [\neg g(T)] \sqcap \text{skip}; [\neg g(T)] \\ &= \{\text{Unfolding [5] : } T^\omega = T; T^\omega \sqcap \text{skip}\} \\ & \quad (\llbracket g(T) \rrbracket); (T; T^\omega \sqcap \text{skip}); [\neg g(T)] \sqcap \text{skip}; [\neg g(T)] \\ &= \{\text{Distribution over } \sqcap \text{ and } \llbracket g(T) \rrbracket; [\neg g(T)] = \text{magic}\} \\ & \quad ((\llbracket g(T) \rrbracket); T; T^\omega; [\neg g(T)] \sqcap \text{magic}) \sqcap \text{skip}; [\neg g(T)] \\ &= \{\text{Definition of } g \text{ and } T \sqcap \text{magic} = T\} \\ & \quad (T; T^\omega; [\neg g(T)] \sqcap \text{skip}); [\neg g(T)] \\ &= \{\text{Distribution over } \sqcap \text{ and rule : } [g]; [g] = [g]\} \\ & \quad (T; T^\omega \sqcap \text{skip}); [\neg g(T)] \\ &= \{\text{Unfolding [5] : } T^\omega = T; T^\omega \sqcap \text{skip}\} \\ & \quad T^\omega; [\neg g(T)] \end{aligned}$$

□

3.3 Scheduling Patterns

The proof obligations in (21)-(23) cannot be proved for arbitrary events. Assumptions about the events have to be made. There are also many possibilities for the schedules to actually be correct. The best approach to scheduling is probably to develop a set of patterns with known correctness conditions. The key to the development of patterns is that we can prove incrementally that the applications of the scheduling function `sched` result in a non-miraculous refinement. This can be used to show that a certain sequence of scheduling steps (i.e. a pattern) results in a refinement. A pattern P consists of a *pre-condition* for the pattern, a *schedule* describing the pattern, a *list of assumptions* about the events in the schedule and the *statement* obtained by applying the pattern. The pre-condition states under which conditions the pattern can be applied and it can therefore be used as a *context assertion*. The schedule describes the scheduling statement the pattern concerns. The list of assumptions describes the assumptions about the events that have to hold before the pattern can be applied. The result of the pattern then gives the statement (set-transformer) obtained after the pattern has been applied.

Loop introduction. Here we will first prove the correctness of the pattern for loop introduction in [6,7]. The pattern states that a loop of events can be refined into a loop where the first event is iterated until it becomes disabled and the second event is then executed. The goal is to introduce an inner while-loop so that the Event-B model to the left below is refined by the one to the right.

$$\boxed{
 \begin{array}{l}
 E_1 \triangleq \text{when } G_1 \text{ then } T_1 \text{ end} \\
 E_2 \triangleq \text{when } G_2 \text{ then } T_2 \text{ end}
 \end{array}
 } \sqsubseteq \boxed{
 \begin{array}{l}
 E \triangleq \text{when } G_1 \vee G_2 \text{ then} \\
 \quad \text{while } G_1 \text{ then } T_1 \text{ end ;} \\
 \quad T_2 \\
 \text{end}
 \end{array}
 }$$

The pattern above can be described as the scheduling pattern P_1 in (24), which has the parameters E_2 denoting a set of events and S_1 denoting an arbitrary schedule of events. Here we do the generalisation of the pattern in [6,7] that the events in the inner loop are scheduled according to some unknown schedule S_1 .

$$\begin{array}{l}
 P_1(E_2, S_1) \triangleq \\
 \text{Precondition} \quad : \text{true} \\
 \text{Schedule} \quad \quad : \text{do do } S_1 \text{ od} \rightarrow E_2 \text{ od} \\
 \text{Assumption 1} \quad : \{i \cap \mathbf{g}(e(S_1) \sqcap E_2)\}; e(S_1) = \{i \cap \mathbf{g}(e(S_1) \sqcap E_2)\}; e(S_1); \{\mathbf{g}(e(S_1) \sqcap E_2)\} \\
 \text{Result} \quad \quad \quad : ([\mathbf{g}(e(S_1) \sqcap E_2)]; ([\mathbf{g}(e(S_1))]); \text{sched}(S_1)^\omega; [\neg \mathbf{g}(e(S_1))]); \\
 \quad \quad \quad \quad \quad E_2)^\omega; [\neg \mathbf{g}(e(S_1) \sqcap E_2)]
 \end{array} \tag{24}$$

The statement in the result is obtained by applying function `sched` three times. Note that we still have one un-scheduled part `sched(S1)`, which is unknown. As before, this statement is interpreted as $e(S_1)^\omega; [\neg \mathbf{g}(e(S_1))]$ at this level. Since we like to prove that the application of the pattern results in a refinement, we get the following condition to prove:

$$\begin{aligned}
& (\mathbf{e}(S_1) \sqcap E_2)^\omega; [\neg \mathbf{g}(\mathbf{e}(S_1) \sqcap E_2)] \\
& \sqsubseteq \\
& ([\mathbf{g}(\mathbf{e}(S_1) \sqcap E_2)]; ([\mathbf{g}(\mathbf{e}(S_1))]; \mathbf{e}(S_1)^\omega; [\neg \mathbf{g}(\mathbf{e}(S_1))])^\omega; [\neg \mathbf{g}(\mathbf{e}(S_1))]; E_2)^\omega; [\neg \mathbf{g}(\mathbf{e}(S_1) \sqcap E_2)]
\end{aligned}$$

To prove this refinement correct, Assumption 1 is needed. This assumption states that events $\mathbf{e}(S_1)$ do not disable both $\mathbf{e}(S_1)$ and E_2 . The condition holds if $\mathbf{e}(S_1)$ was introduced after E_2 in the refinement chain and the value of $\mathbf{g}(\mathbf{e}(S_1) \sqcap E_2)$ depends only on variables introduced before or simultaneously with E_2 (events $\mathbf{e}(S_1)$ cannot change the value of this condition then). This requirement is also discussed in [7]. Note that the invariant i from the Event-B model holds after the execution of each event. This means that invariant assertions can be added between events, but they are not written out here [9]. To make the proof more readable, events $\mathbf{e}(S_1)$ are denoted E_1 . The refinement can now be proved:

Proof.

$$\begin{aligned}
& (E_1 \sqcap E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Decomposition (Lemma 12 in [5])} : (S \sqcap T)^\omega = S^\omega; (T; S^\omega)^\omega\} \\
& E_1^\omega; (E_2; E_1^\omega)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Leapfrog (Lemma 11 in [5])} : S; (T; S)^\omega = (S; T)^\omega; S\} \\
& (E_1^\omega; E_2)^\omega; E_1^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Rule} : S^\omega \sqsubseteq \text{skip}\} \\
& (E_1^\omega; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Assumption introduction} : \text{skip} \sqsubseteq [g]\} \\
& (E_1^\omega; [\neg \mathbf{g}(E_1)]; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\mathbf{g}(E_1) \subseteq \mathbf{g}(E_1 \sqcap E_2) \text{ and } g \subseteq h \Rightarrow [g] = [h]; [g]\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg \mathbf{g}(E_1)]; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Rule} : [g] = [g]; \{g\}\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; \{\mathbf{g}(E_1 \sqcap E_2)\}; E_1^\omega; [\neg \mathbf{g}(E_1)]; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Assumption 1 and Lemma 14c in [5]} : S; T \sqsubseteq U; S \Rightarrow S; T^\omega \sqsubseteq U^\omega; S\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; \{\mathbf{g}(E_1 \sqcap E_2)\}; [\neg \mathbf{g}(E_1)]; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
\sqsubseteq & \{\text{Distribution of assertions over assumptions and rule} : [g] = [g]\{g\}\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg \mathbf{g}(E_1)]; [\neg \mathbf{g}(E_1)]; \{\mathbf{g}(E_1 \sqcap E_2)\}; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Assertion properties} : \{g\}; \{h\} = \{g \cap h\} \text{ and definition of } \mathbf{g}\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg \mathbf{g}(E_1)]; \{\neg \mathbf{g}(E_1) \cap (\mathbf{g}(E_1) \cup \mathbf{g}(E_2))\}; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Set theory}\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg \mathbf{g}(E_1)]; \{\neg \mathbf{g}(E_1) \cap \mathbf{g}(E_2)\}; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Rule} : \{g \cap h\} = \{g\}; \{h\} \text{ and } [g] = [g]; \{g\}\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg \mathbf{g}(E_1)]; \{\mathbf{g}(E_2)\}; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)] \\
= & \{\text{Lemma 1 and } \{g\} \sqsubseteq \text{skip}\} \\
& ([\mathbf{g}(E_1 \sqcap E_2)]; ([\mathbf{g}(E_1)]; E_1^\omega; [\neg \mathbf{g}(E_1)])^\omega; [\neg \mathbf{g}(E_1)]; E_2)^\omega; [\neg \mathbf{g}(E_1 \sqcap E_2)]
\end{aligned}$$

In order to ensure that the refined statement is non-miraculous, we prove that $\mathbf{g}([\mathbf{g}(E_1 \sqcap E_2)]; ([\mathbf{g}(E_1)]; E_1)^\omega; [\neg \mathbf{g}(E_1)]; E_2) = \mathbf{g}(E_1 \sqcap E_2)$ and then use Lemma 17b in [5], $S^\omega; [\neg \mathbf{g}(S)](\text{false}) = \text{false}$.

$$\begin{aligned}
& \mathbf{g}([\mathbf{g}(E_1 \sqcap E_2)]; ([\mathbf{g}(E_1)]; E_1^\omega; [\neg\mathbf{g}(E_1)]^\omega; [\neg\mathbf{g}(E_1)]; E_2) \\
= & \{ \text{Lemma 1} \} \\
& \mathbf{g}([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg\mathbf{g}(E_1)]; E_2) \\
= & \{ \text{Rule : } [g] = [g]; \{g\}, \text{ Assumption 1, , Set - theory} \} \\
& \mathbf{g}([\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg\mathbf{g}(E_1)]; \{g(E_2)\}; E_2) \\
= & \{ \text{Definition of } \mathbf{g} \} \\
& \neg[\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg\mathbf{g}(E_1)]; \{g(E_2)\}; E_2 \text{ (false)} \\
= & \{ \{g(E_2)\}; E_2 \text{ (false)} = \text{false} \} \\
& \neg[\mathbf{g}(E_1 \sqcap E_2)]; E_1^\omega; [\neg\mathbf{g}(E_1)] \text{ (false)} \\
= & \{ \text{Rule (Lemma 17b in [5]) : } S^\omega; [\neg\mathbf{g}(S)](\text{false}) = \text{false} \} \\
& \neg[\mathbf{g}(E_1 \sqcap E_2)] \text{ (false)} \\
= & \{ \text{Definitions} \} \\
& \neg(\neg\mathbf{g}(E_1 \sqcap E_2) \cup \text{false}) \\
= & \{ \text{Set theory} \} \\
& \mathbf{g}(E_1 \sqcap E_2)
\end{aligned}$$

□

This demonstrates one possible proof of one simple schedule. Note that the user of this scheduling pattern will only have to prove Assumption 1.

Sequential composition. One of the most important type of patterns concerns the introduction of sequential composition of events. This pattern is already given as the equation two in the definition of sched (20). It is thus a pattern that concerns only one application of sched:

$$\begin{aligned}
P_{seq}(E, g, S) & \hat{=} \\
\text{Precondition} & : \mathbf{g}(E) \\
\text{Schedule} & : E \rightarrow (\{g\} \rightarrow S) \\
\text{Assumption 1} & : \{i \cap \neg\mathbf{g}(E)\}; \mathbf{e}(S) = \\
& \{i \cap \neg\mathbf{g}(E)\}; \mathbf{e}(S); \{\neg\mathbf{g}(E)\} \\
\text{Assumption 2} & : \{i\}; E = \{i\}; E; \{\neg\mathbf{g}(E) \cap g\} \\
\text{Result} & : E; \{g\}; \text{sched}(S)
\end{aligned} \tag{25}$$

To verify this pattern, we have to prove condition (22). There are no unique assumptions that would enable the proof, but there are several possibilities. In this pattern, we use the assumption that once E has become disabled it remains disabled. This property is stated in Assumption 1. To prove the schedule correct, Assumption 2 is also needed. The proof is carried out in the same manner as the proof of pattern P_1 .

This way of proving sequencing of events is not without problems. Note that the assumptions about the events for this pattern do not depend on only E , but also on the all the rest of the events $\mathbf{e}(S)$. Here we prove that each event does not become re-enabled. Assume we have n events that should be executed after each other. We need to show for each step that the event just scheduled is not re-enabled by any event that comes after. Furthermore, each condition of the form $g \subseteq (E_1 \sqcap E_2)(h)$ is divided into two separate proof obligations $g \subseteq E_1(h)$ and $g \subseteq E_2(h)$. Taking these properties into account, there will be $O(n^2)$ proof obligations (more exactly $n(n-1)/2$) from the schedule due to Assumption 1. For long sequences of events this means there are a huge number of proof

obligations to prove. This is not the only assumption that enables a proof, but the correctness will always in the end depend also on the events $e(S)$.

Discussion. We have here given two patterns to introduce nested loops and sequential composition to Event-B specifications. Wherever in a schedule, a schedule fragment matching the pattern occurs, we can use the result of the pattern and be sure that it leads to a correct refinement of the original event loop. This requires that the assumptions the pattern rely on are fulfilled. The choice of patterns that were presented here is rather ad-hoc. They were chosen to demonstrate how patterns are developed and verified. To use the scheduling method efficiently, a library of scheduling patterns with associated proof obligations would be needed.

One might ask why the scheduling language and the patterns are needed at all. It would also be possible to directly use the algebraic rules to reason about an Event-B model as a whole. The problem with that approach is that the derivations and proofs have to be done for each developed program, which might be demanding especially for people who are not formal methods experts. The patterns expressed in the scheduling language encode reusable structures that have known proof obligations, which could be generated automatically by a tool. The idea is that the scheduling method and patterns partition the scheduling problem into smaller parts. New patterns can then be applied separately on the parts themselves. This will hopefully make the scheduling problem more manageable.

3.4 Creation of Sequential Programs

The scheduling of events presented so far does not yet provide deterministic programs, only statements that are known to be implementable. To develop sequential programs with the familiar constructs, such as if-statements and while-loops, the scheduling language need to be extended slightly. First recall that an event of the form $E = \mathbf{when } G \mathbf{ then } S \mathbf{ end}$ corresponds to a set transformer $[E] = [g]; [s]$ as discussed in Section 2. Let $s(E)$ denote the assignment part of event E , i.e., here $s(E) = S$. We have the extended grammar for the scheduling language:

$$\begin{aligned}
 DoStmnt &::= \mathbf{while } Stmnt \mathbf{end} \mid \mathbf{do } Stmnt \mathbf{od} \\
 ChoiceStmnt &::= \mathbf{if } g(E_1) \mathbf{then } s(E_1) \mathbf{elseif } \dots \mathbf{else } s(E_n) \mathbf{end} \mid \\
 &\quad s(E) \mid E_1 \parallel \dots \parallel E_n \\
 Stmnt &::= [\{g\} \rightarrow] (DoStmnt \mid ChoiceStmnt) [\rightarrow Stmnt]
 \end{aligned} \tag{26}$$

No new concepts is needed for verification of schedules that conforms to this scheduling language. The assignment $s(E)$ does not introduce any new proof obligations, as it can be verified by replacing it by $\{g(E)\} \rightarrow [E]$, since

$$\{g(E)\}; [E] \sqsubseteq [s(E)] \tag{27}$$

We can also introduce if-statements $\mathbf{if } g(E_1) \mathbf{then } s(E_1) \mathbf{elseif } \dots \mathbf{else } s(E_n) \mathbf{end}$ as a refinement of the original choice statement. This does not require any new

proof obligations, but can be verified by replacing it by $\{g(\parallel_i E)\} \rightarrow [\parallel_i E]$, since

$$\{g(\parallel_i E)\}; [\parallel_i E] \sqsubseteq \mathbf{if} \ g(E_1) \ \mathbf{then} \ [s(E_1)] \ \mathbf{elseif} \ \dots \ \mathbf{else} \ [s(E_n)] \ \mathbf{end} \quad (28)$$

[11]. This is similar to the implementation pattern for choice between events given in [7]. While loops can also be easily introduced. The semantics of the while-loop $\mathbf{while} \ g \ \mathbf{then} \ S \ \mathbf{end}$ is given as $e(S)^\omega; [\neg g]$ [11]. Due to how a loop of the form $\mathbf{do} \ S \ \mathbf{od}$ is verified in the schedule, it will always be translated as $e(S)^\omega; [\neg g(\parallel_i e(S))]$. We thus have that:

$$\mathbf{while} \ g(\parallel_i e(S)) \ \mathbf{then} \ S \ \mathbf{end} = \mathbf{do} \ S \ \mathbf{od} \quad (29)$$

3.5 Example of Development of a Sequential Program

To give feel for how the control flow constructs can be used for development of a sequential program in Event-B, a small example is given. The example consists of the development of an algorithm for computing the variance of an array of integers. We use the method outlined in [6,7] for the algorithm development. The variance V of an array $f : 1..n \rightarrow \mathbb{Z}$ is defined as $V = \frac{1}{n} \sum_{i=1}^n (f(i) - \mu)^2$, where μ is the arithmetic mean $\mu = \frac{1}{n} \sum_{i=1}^n f(i)$. The variance is used in many statistical methods, e.g. the standard deviation is the square root of the variance. To make the algorithm efficient, the mean is first pre-computed and then used in the computation of the variance. In the abstract machine *Variance* in Fig. [1] the variance is computed in one step. In the refinement machine *Variance1* the pre-computation of the mean value *mean* is introduced. In the final refinement machine *Variance2* shown in Fig. [2], the sums are iteratively computed. The final model is implementable as all constructs in the events are deterministic. Note that the merging rules in [6,7] do not work well for creating a sequential program here, since the event *comp_mean* should be executed only once in between two loops. Using our scheduling approach we can derive and verify a schedule:

$$\begin{aligned} \text{Sched} &\hat{=} \\ &\mathbf{while} \ g(c_m_p) \ \mathbf{then} \ s(c_m_p) \ \mathbf{end} \ \rightarrow s(c_m) \ \rightarrow \\ &\mathbf{while} \ g(c_v_p) \ \mathbf{then} \ s(c_v_p) \ \mathbf{end} \end{aligned} \quad (30)$$

Note that we are only interested in scheduling the new events, since due to the development method in [6,7], the original event is known to be executed last. This is discussed in more detail in [9]. All constructs in this schedule did not occur in the language given in (19) and in the corresponding verification rules (21)-(23). However, verification of also these constructs was presented in Subsection 3.4. Using the mappings for unguarded assignments and while-loops we get the schedule:

$$\text{Sched2} \hat{=} \mathbf{do} \ c_m_p \ \mathbf{od} \ \rightarrow \{g(c_m)\} \ \rightarrow c_m \ \rightarrow \mathbf{do} \ c_v_p \ \mathbf{od} \quad (31)$$

To verify this schedule, we like to apply patterns. However, no pattern developed in the paper so far match the beginning of this schedule. An additional pattern is needed for verification. Such a pattern is, e.g., P_{loop} :

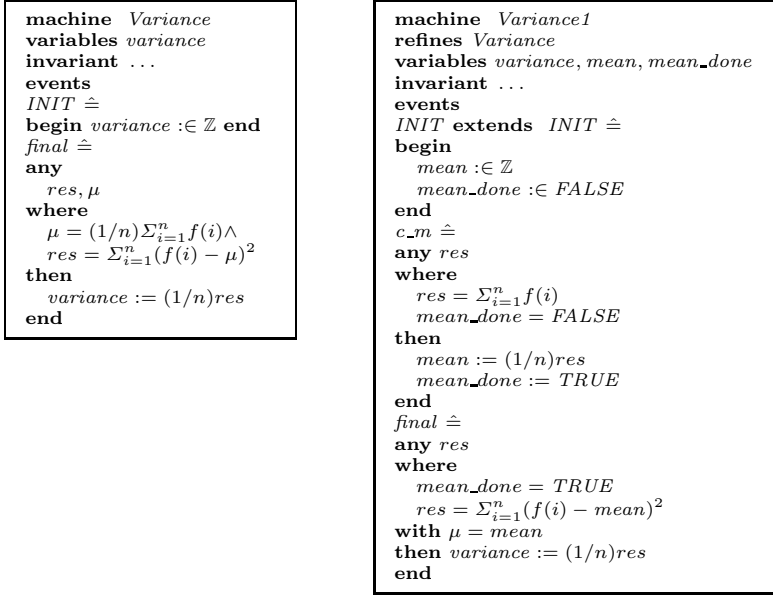


Fig. 1. Abstract model and its refinement for computing the variance of array f

$$\begin{aligned}
& P_{loop}(E_1, g, S) \hat{=} \\
& \text{Precondition} : \text{true} \\
& \text{Schedule} : \mathbf{do} E_1 \mathbf{od} \rightarrow \{g\} \rightarrow S \\
& \text{Assumption 1} : \{i \cap \neg g(E_1)\}; e(S) = \\
& \quad \{i \cap \neg g(E_1)\}; e(S); \{\neg g(E_1)\} \\
& \text{Assumption 2} : \{i \cap \neg g(E_1)\} = \{i \cap \neg g(E_1) \cap g\} \\
& \text{Result} : E_1^\omega; [\neg g(E_1)]; \{g\}; \text{sched}(S)
\end{aligned} \tag{32}$$

Using the pattern P_{seq} and P_{loop} the schedule can now be verified. First we instantiate P_{loop} as $P_{loop}(c_m-p, g(c_m), c_m \rightarrow \mathbf{do} c_v-p \mathbf{od})$. On the remaining schedule in the result, we then instantiate P_{seq} as $P_{seq}(c_m, \text{true}, \mathbf{do} c_v-p \mathbf{od})$. Note that the pre-condition for the pattern is given as the assertion $\{g(c_m)\}$. To then obtain the final program, we instantiate P_{loop} again, but now as $P_{loop}(c_v-p, \text{true}, \langle \rangle)$. After these pattern instantiations, we have obtained a set-transformer that is a non-miraculous refinement of the original loop of events. The user only has to prove that the assumptions about the events in the patterns hold. In this case, the assumptions lead to four proof obligations.

Note that we here chose to do the scheduling on the final Event-B model. The scheduling could be introduced earlier as in [34] and the scheduled Event-B model could then be refined. This has not been explored in the paper. However, the refinement proof obligations in Event-B are compatible with the set-transformers in the paper and refinement of schedules would therefore be straightforward to incorporate in this scheduling framework.

<pre> machine <i>Variance2</i> refines <i>Variance1</i> variables <i>variance, mean, m_comp,</i> <i>v_comp, i, j</i> invariant ... events <i>INIT extends INIT</i> $\hat{=}$ begin ... <i>m_comp, v_comp := 0, 0</i> <i>i, j := 0, 0</i> end <i>c_m_p</i> $\hat{=}$ when $i < n$ then <i>m_comp := m_comp + f(i + 1)</i> <i>i := i + 1</i> end </pre>	<pre> <i>c_m</i> $\hat{=}$ when $i = n$ with $res = m_comp$ then <i>mean := (1/n)m_comp</i> <i>i := i + 1</i> end <i>c_v_p</i> $\hat{=}$ when $i > n \wedge j < n$ then <i>v_comp := v_comp + (f(i + 1) - mean)²</i> <i>j := j + 1</i> end <i>final</i> $\hat{=}$ when $j = n$ with $res = v_comp$ then <i>variance := (1/n)v_comp</i> end </pre>
---	---

Fig. 2. The final refinement of the models in Fig. 1

4 Conclusions

This paper describes a method for introducing control flow constructs to Event-B models and for deriving sequential programs from event-based specifications. We first presented a suitable semantics for Event-B models for this purpose, which was based on set transformers. A scheduling language for describing the flow of control was then presented. The algebraic approach from [5] was used to analyse the models and the scheduling. Using this approach, we developed and verified scheduling patterns and applied them on an example.

The most similar approaches are [3] and [4]. However, here we do the analysis based on set-transformers which is a very well-developed theory with powerful algebraic methods for high-level program analysis. Scheduling of events in Event-B or actions in action system has also been done in CSP before [14][15]. A related approach has also been used for hardware synthesis utilising a special scheduling language [16]. In those approaches, program counters are introduced in the (Event-)B models to verify the scheduling. Here we use a more direct approach where we directly give the needed proof obligations for a schedule. Our approach is not as flexible as scheduling using CSP, but it might be easier to apply since the correctness conditions for the schedules are explicit. The benefits of our approach is that it is easy to derive and analyse reusable scheduling patterns, which help reasoning about control flow in Event-B models in practice.

There are limitations to creating sequential programs from event-based specifications using the approach proposed in the paper. The proof obligations needed by the scheduling can create a significant extra work (see pattern presented for sequential composition). On the other hand, the example showed that it can work rather well in some circumstances. However, it might be beneficial to introduce at least some degree of control flow already in the more abstract models as described in [4]. This could be easily done here also. The expressiveness of the scheduling language is also a limitation. E.g., the choice operator \parallel operates

on events and not on statements. This was a design decision to simplify the methods in this paper. However, this limitation will be removed in the future.

This paper gives one approach how to schedule events and prove the correctness of schedules. It shows how scheduling patterns can be developed and proved. The algebraic approach used in the paper seems to be useful for reasoning about Event-B models on a higher level of abstraction than the traditional proof obligations.

References

1. Abrial, J.R.: *Modelling in Event B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
2. Leuschel, M., Butler, M.: ProB: An Automated Analysis Toolset for the B Method. *Journal Software Tools for Technology Transfer* 10(2), 185–203 (2008)
3. Iliasov, A.: *On Event-B and control flow*. Technical Report CS-TR No 1159, School of Computing Science, Newcastle University (2009)
4. Hallerstede, S.: Structured Event-B models and proofs. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) *ABZ 2010*. LNCS, vol. 5977, pp. 273–286. Springer, Heidelberg (2010)
5. Back, R.J.R., von Wright, J.: Reasoning algebraically about loops. *Acta Informatica* 36, 295–334 (1999)
6. Abrial, J.R.: Event driven sequential program construction. Clearsty (2001), <http://www.atelierb.eu/php/documents-en.php>
7. Abrial, J.R.: Event based sequential program development: Application to constructing a pointer program. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) *FME 2003*. LNCS, vol. 2805, pp. 51–74. Springer, Heidelberg (2003)
8. Méry, D.: Refinement-based guidelines for algorithmic systems. *Int. J. Software and Informatics* 3(2-3), 197–239 (2009)
9. Boström, P.: *Creating sequential programs from Event-B models*. Technical Report 955, TUCS, Turku, Finland (2009)
10. Hallerstede, S.: On the purpose of Event-B proof obligations. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) *ABZ 2008*. LNCS, vol. 5238, pp. 125–138. Springer, Heidelberg (2008)
11. Back, R.J.R., von Wright, J.: *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, Heidelberg (1998)
12. Back, R.J.R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing*, pp. 131–142 (1983)
13. Back, R.J.R., Sere, K.: Stepwise refinement of action systems. *Structured Programming* 12, 17–30 (1991)
14. Butler, M.: Stepwise refinement of communicating systems. *Science of Computer Programming* 27, 139–173 (1996)
15. Butler, M.: csp2b: A practical approach to combining CSP and B. *Formal Aspects of Computing* 12(3), 182–198 (2000)
16. Plosila, J., Sere, K., Waldén, M.: Asynchronous system synthesis. *Science of Computer Programming* 55, 259–288 (2005)

Symbolic Model-Checking of Optimistic Replication Algorithms

Hanifa Boucheneb¹, Abdessamad Imine², and Manal Najem¹

¹ Laboratoire VeriForm, Department of Computer Engineering,
École Polytechnique de Montréal, P.O. Box 6079, Station Centre-ville, Montréal,
Québec, Canada, H3C 3A7

hanifa.boucheneb@polymtl.ca

² INRIA Grand-Est & Nancy-Université, France
imine@loria.fr

Abstract. The Operational Transformation (OT) approach, used in many collaborative editors, allows a group of users to concurrently update replicas of a shared object and exchange their updates in any order. The basic idea of this approach is to transform any received update operation before its execution on a replica of the object. This transformation aims to ensure the convergence of the different replicas of the object. However, designing transformation algorithms for achieving convergence is a critical and challenging issue. In this paper, we address the verification of OT algorithms with a symbolic model-checking technique. We show how to use the difference bound matrices to explore symbolically infinite state-spaces of such systems and provide symbolic counterexamples for the convergence property.

Keywords: collaborative editors; operational transformation; difference bound matrices; symbolic model checking; convergence property.

1 Introduction

Motivations. Collaborative editing systems constitute a class of distributed systems where dispersed users interact by manipulating simultaneously some shared objects like texts, images, graphics, etc. One of the main challenges is the data consistency. To improve data availability, optimistic consistency control techniques are commonly used. The shared data is replicated so that the users update their local data replicas and exchange their updates between them. So, the updates are applied in different orders at different replicas of the object. This potentially leads to divergent (or different) replicas, an undesirable situation for collaborative editing systems. *Operational Transformation* (OT) is an optimistic technique which has been proposed to overcome the divergence problem [4]. This technique consists of an algorithm which transforms an update (previously executed by some other user) according to local concurrent updates in order to achieve convergence. It is used in many collaborative editors including Joint Emacs [8] (an Emacs collaborative editor), CoWord [13] (a collaborative version of

Microsoft Word), CoPowerPoint [13] (a collaborative version of Microsoft PowerPoint) and, more recently, the Google Wave (a new google platform¹).

It should be noted that the data consistency relies crucially on the correctness of an OT algorithm. According to [8], the consistency is ensured iff the transformation function satisfies two properties $TP1$ and $TP2$ (explained in Section 2). Finding such a function and proving that it satisfies $TP1$ and $TP2$ is not an easy task. In addition, the proof by hand of these properties is often unmanageably complicated due to the fact that an OT algorithm has infinitely many states. Consequently, proving the correctness of OT algorithms should be assisted by automatic tools.

Related Work. Very little research has been done on automatically verifying the correctness of OT algorithms. To the best of our knowledge, [7] is the first work that addresses this problem. In this work, the authors have proposed a formal framework for modelling and verifying transformation functions with algebraic specifications. For checking the properties $TP1$ and $TP2$, they used an automatic theorem prover. However, this theorem proving approach has some shortcomings: (i) the model of the system is sound but not complete w.r.t. $TP1$ and $TP2$ (i.e., it does not guarantee that the violation of property $TP1$ or $TP2$ is really feasible); (ii) there is no guidance to understand the counterexamples (when the properties are not verified); (iii) it requires some interaction (by injecting new lemmas) to complete the verification. In [3], the authors have used a model-checking technique to verify OT algorithms. This approach is not based on the verification of properties $TP1$ and $TP2$ but is instead based on the generation of the effective traces of the system. So, it allows to get a complete and informative scenario when a bug (a divergence of two copies of the shared object) is detected. Indeed, the output contains all necessary operations and the step-by-step execution that lead to the divergence situation. This approach guaranties that the detected divergence situations are really feasible. However, it needs to fix the shared object, the number of sites, the number of operations, the domains of parameters of operations and to execute explicitly the updates.

Contributions. We propose here a symbolic model-checking technique, based on difference bound matrices (DBMs) [1], to verify whether an OT algorithm satisfies properties $TP1$ and $TP2$. We show how to use DBMs, to handle symbolically the update operations of the collaborative editing systems and to verify symbolically the properties $TP1$ and $TP2$. The verification of these properties is performed automatically without carrying out different copies of the shared object and executing explicitly the updates. So, there is no need to fix the alphabet and the maximal length of the shared object. Thus, unlike [3], the symbolic model-checking proposed here enables us to get more abstraction and to build symbolic counterexamples. Moreover, for fixed numbers of sites and operations, it allows to prove whether or not an OT algorithm satisfies properties $TP1$ and $TP2$.

The paper starts with a presentation of the OT approach (Section 2). Section 3 is devoted to our symbolic model-checking. Conclusions are presented in Section 4.

¹ <http://www.waveprotocol.org/whitepapers/operational-transform>

² A model M of a system S is said to be sound w.r.t. a given property ϕ if M satisfies ϕ implies S satisfies ϕ . It is complete w.r.t. ϕ if S satisfies ϕ implies M satisfies ϕ .

2 Operational Transformation Approach

2.1 Background

OT is an optimistic replication technique which allows many sites to concurrently update the shared data and next to synchronize their divergent replicas in order to obtain the same data. The updates of each site are executed on the local replica immediately without being blocked or delayed, and then are propagated to other sites to be executed again. The shared object is a finite sequence of elements from a data type \mathcal{E} (alphabet). This data type is only a template and can be instantiated by many other types. For instance, an element may be regarded as a character, a paragraph, a page, a slide, an XML node, etc. It is assumed that the shared object can only be modified by the following primitive operations:

$$\mathcal{O} = \{Ins(p, e) | e \in \mathcal{E} \text{ and } p \in \mathbb{N}\} \cup \{Del(p) | p \in \mathbb{N}\} \cup \{Nop\}$$

where $Ins(p, e)$ inserts the element e at position p ; $Del(p)$ deletes the element at position p , and Nop is the idle operation that has null effect on the object. Since the shared object is replicated, each site will own a local state l that is altered only by operations executed locally. The initial state of the shared object, denoted by l_0 , is the same for all sites. Let \mathcal{L} be the set of states. The function $Do : \mathcal{O} \times \mathcal{L} \rightarrow \mathcal{L}$, computes the state $Do(o, l)$ resulting from applying operation o to state l . We denote by $[o_1; o_2; \dots; o_n]$ an operation sequence. Applying an operation sequence to a state l is defined as follows: (i) $Do([], l) = l$, where $[]$ is the empty sequence and; (ii) $Do([o_1; o_2; \dots; o_n], l) = Do(o_n, Do(\dots, Do(o_2, Do(o_1, l))))$. Two operation sequences seq_1 and seq_2 are *equivalent*, denoted by $seq_1 \equiv seq_2$, iff $Do(seq_1, l) = Do(seq_2, l)$ for all states l .

The OT approach is based on two notions: concurrency and dependency of operations. Let o_1 and o_2 be operations generated at sites i and j , respectively. We say that o_2 *causally depends* on o_1 , denoted $o_1 \rightarrow o_2$, iff: (i) $i = j$ and o_1 was generated before o_2 ; or, (ii) $i \neq j$ and the execution of o_1 at site j has happened before the generation of o_2 . Operations o_1 and o_2 are said to be *concurrent*, denoted by $o_1 \parallel o_2$, iff neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$. As a long established convention in OT-based collaborative editors [4,11], the *timestamp vectors* are used to determine the causality and concurrency relations between operations. A timestamp vector is associated with each site and each generated operation. Every timestamp is a vector of integers with a number of entries equal to the number of sites. For a site j , each entry $V_j[i]$ returns the number of operations generated at site i that have been already executed on site j . When an operation o is generated at site i , a copy V_o of V_i is associated with o before its broadcast to other sites. $V_i[i]$ is then incremented by 1. Once o is received at site j , if the local vector V_j “dominates”³ V_o , then o is ready to be executed on site j . In this case, $V_j[i]$ will be incremented by 1 after the execution of o . Otherwise, the o ’s execution is delayed.

Let V_{o_1} and V_{o_2} be timestamp vectors of o_1 and o_2 , respectively. Using these timestamp vectors, the causality and concurrency relations are defined as follows:

(i) $o_1 \rightarrow o_2$ iff $V_{o_1}[i] < V_{o_2}[j]$; (ii) $o_1 \parallel o_2$ iff $V_{o_1}[i] \geq V_{o_2}[j]$ and $V_{o_2}[j] \geq V_{o_1}[i]$.

³ We say that V_1 dominates V_2 iff $\forall i, V_1[i] \geq V_2[i]$.

2.2 Operational Transformation Approach

A crucial issue when designing shared objects with a replicated architecture and arbitrary messages communication between sites is the *consistency maintenance* (or *convergence*) of all replicas. To illustrate this problem, consider the group text editor scenario shown in Fig 1. There are two users (on two sites) working on a shared document represented by a sequence of characters. Initially, both copies hold the string “*efecte*”. Site 1 executes operation $o_1 = Ins(1, f)$ to insert the character *f* at position 1. Concurrently, site 2 performs $o_2 = Del(5)$ to delete the character *e* at position 5. When o_1 is received and executed on site 2, it produces the expected string “*effect*”. But, when o_2 is received on site 1, it does not take into account that o_1 has been executed before it and it produces the string “*effece*”. The result at site 1 is different from the result of site 2 and it apparently violates the intention of o_2 since the last character *e*, which was intended to be deleted, is still present in the final string. Consequently, we obtain a *divergence* between sites 1 and 2. It should be pointed out that even if a serialization protocol [4] was used to require that all sites execute o_1 and o_2 in the same order (*i.e.* a global order on concurrent operations) to obtain an identical result *effece*, this identical result is still inconsistent with the original intention of o_2 .

To maintain convergence, the *Operational Transformation* (OT) approach has been proposed by [4]. When a site i gets an operation o that was previously executed by a site j on his replica of the shared object, the site i does not necessarily integrate o by executing it “as is” on his replica. It will rather execute a variant of o , denoted by o' (called a *transformation* of o) that *intuitively intends to achieve the same effect* as o . This transformation is based on an *Inclusive Transformation* (IT) function.

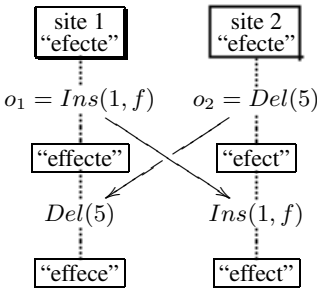


Fig. 1. Incorrect integration

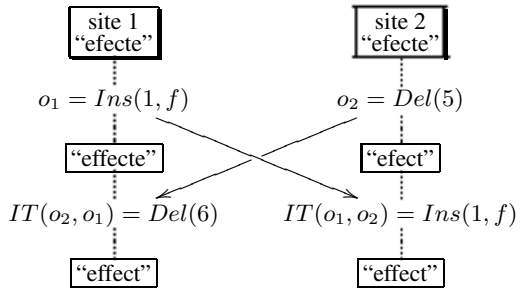


Fig. 2. Integration with transformation

As an example, Fig 2 illustrates the effect of an *IT* function on the previous example. When o_2 is received on site 1, o_2 needs to be transformed according to o_1 as follows: $IT(Del(5), Ins(1, f)) = Del(6)$. The deletion position of o_2 is incremented because o_1 has inserted a character at position 1, which is before the character deleted by o_2 . Next, o'_2 is executed on site 1. In the same way, when o_1 is received on site 2, it is transformed as follows: $IT(Ins(1, f), Del(5)) = Ins(1, f)$; o_1 remains the same because f is inserted before the deletion position of o_2 .

2.3 Inclusive Transformation Functions

We can find, in the literature, several IT functions: *Ellis's* algorithm [4], *Ressel's* algorithm [8], *Sun's* algorithm [12], *Suleiman's* algorithm [9] and *Imine's* algorithm [6]. Due to the lack of space, we report, here, only the IT function proposed by Ellis and Gibbs [4]. In Ellis's IT function, the insert operation is extended with another parameter pr ⁴. The priority pr is used to solve a conflict occurring when two concurrent insert operations were originally intended to insert different characters at the same position. Note that concurrent editing operations have always different priorities. Fig.3 gives the four transformation cases for *Ins* and *Del* proposed by Ellis and Gibbs.

$$\begin{array}{l}
 IT(Ins(p_1, c_1, pr_1), Ins(p_2, c_2, pr_2)) = \\
 \left\{ \begin{array}{ll} Ins(p_1, c_1, pr_1) & \text{if } (p_1 < p_2) \vee (p_1 = p_2 \wedge c_1 \neq c_2 \wedge pr_1 < pr_2) \\ Ins(p_1 + 1, c_1, pr_1) & \text{if } (p_1 > p_2) \vee (p_1 = p_2 \wedge c_1 \neq c_2) \wedge pr_1 > pr_2 \\ Nop() & \text{if } p_1 = p_2 \wedge c_1 = c_2 \end{array} \right. \\
 IT(Ins(p_1, c_1, pr_1), Del(p_2)) = \left\{ \begin{array}{ll} Ins(p_1, c_1, pr_1) & \text{if } p_1 < p_2 \\ Ins(p_1 - 1, c_1, pr_1) & \text{otherwise} \end{array} \right. \\
 IT(Del(p_1), Ins(p_2, c_2, pr_2)) = \left\{ \begin{array}{ll} Del(p_1) & \text{if } p_1 < p_2 \\ Del(p_1 + 1) & \text{otherwise} \end{array} \right. \\
 IT(Del(p_1), Del(p_2)) = \left\{ \begin{array}{ll} Del(p_1) & \text{if } p_1 < p_2 \\ Del(p_1 - 1) & \text{if } p_1 > p_2 \\ Nop() & \text{otherwise} \end{array} \right.
 \end{array}$$

Fig. 3. IT function of Ellis et al

Let $seq = [o_1; o_2; \dots; o_n]$ be a sequence of operations. Transforming any editing operation o according to seq is denoted by $IT^*(o, seq)$ and is recursively defined by: $IT^*(o, []) = o$, where $[]$ is the empty sequence, and $IT^*(o, [o_1; o_2; \dots; o_n]) = IT^*(IT(o, o_1), [o_2; \dots; o_n])$.

2.4 Integration Procedures

Several integration procedures have been proposed in the groupware research area, such as dOPT [4], adOPTed [8], SOCT2,4 [10, 14] and GOTO [11]. Every site generates operations sequentially and stores these operations in a stack also called a *history* (or *execution trace*). When a site receives a remote operation o , the integration procedure executes the following steps:

1. From the local history seq , it determines the equivalent sequence seq' that is the concatenation of two sequences seq_h and seq_c where (i) seq_h contains all operations happened before o (according to the causality relation defined in Subsection 2.1), and (ii) seq_c consists of operations that are concurrent to o .
2. It calls the transformation component in order to get operation o' that is the transformation of o according to seq_c (i.e. $o' = IT^*(o, seq_c)$).
3. It executes o' on the current state and then adds o' to local history seq .

⁴ This priority is calculated at the originating site. Two operations generated from different sites have always different priorities. Usually, the priority i is assigned to all operations generated at site i .

The integration procedure allows history of executed operations to be built on every site, provided that the causality relation is preserved. At stable state⁵, history sites are not necessarily identical because the concurrent operations may be executed in different orders. Nevertheless, these histories must be equivalent in the sense that they must lead to the same final state.

2.5 Consistency Criteria

An OT-based collaborative editor is *consistent* iff it satisfies the following properties:

1. *Causality preservation*: if $o_1 \rightarrow o_2$ then o_1 is executed before o_2 at all sites.
2. *Convergence*: when all sites have performed the same set of updates, the copies of the shared document are identical.

To preserve the causal dependency between updates, timestamp vectors are used. In [8], the authors have established two properties *TP1* and *TP2* that are necessary and sufficient to ensure data convergence for *any number* of operations executed in *arbitrary order* on copies of the same object: For all o_0, o_1 and o_2 pairwise concurrent operations:

- *TP1*: $[o_0; IT(o_1, o_0)] \equiv [o_1; IT(o_0, o_1)]$.
- *TP2*: $IT^*(o_2, [o_0; IT(o_1, o_0)]) = IT^*(o_2, [o_1; IT(o_0, o_1)])$.

Property *TP1* defines a *state identity* and ensures that if o_0 and o_1 are concurrent, the effect of executing o_0 before o_1 is the same as executing o_1 before o_0 . Property *TP2* ensures that transforming o_2 along equivalent and different operation sequences will give the same operation. Accordingly, by these properties, it is not necessary to enforce a global total order between concurrent operations because data divergence can always be repaired by operational transformation. However, finding an IT function that satisfies *TP1* and *TP2* is considered as a hard task, because this proof is often unmanageably complicated.

3 Modeling Execution Environment of the OT Algorithms

We propose a symbolic model, based on the DBM data structure and communicating extended automata, to verify the consistency of OT algorithms. The DBM data structure is usually used to handle dense time domains in model-checkers of timed systems. It is used here to handle symbolically parameters (positions and symbols) of update operations (discrete domains) and verify properties *TP1* and *TP2* on traces (sequences of operations). Using DBM enables us to 1) abstract the shared object, 2) manipulate symbolically parameters of operations without fixing their sizes and 3) provide symbolic counterexamples for *TP1* and *TP2*. First, we present the DBM data structure. Afterwards, our symbolic model of the OT execution environment is described. We show, at this level, how to use the DBM data structure to handle symbolically operations and verify properties *TP1* and *TP2*.

⁵ A stable state is a state where all sites have executed the same set of operations but possibly in different orders.

3.1 Difference Bound Matrices

Let $X = \{x_1, \dots, x_n\}$ be a finite and nonempty set of variables. An atomic constraint over X is a constraint of the form $x_i - x_j \prec c$, $x_i \prec c$, or $-x_j \prec c$, where $x_i, x_j \in X$, $\prec \in \{<, \leq\}$ and $c \in \mathbb{Z}$, \mathbb{Z} being the set of integers. Constraints of the form $x_i - x_j \prec c$ are triangular constraints while the others are simple constraints. Constraints $x_i \prec x_j + c$, $x_i = x_j + c$, $x_i \geq x_j + c$, $x_i > x_j + c$, $x_i > c$, $-x_i > c$, $x_i \geq c$ and $-x_i \geq c$ are considered as abbreviations of atomic constraints.

In the context of this paper, X is a set of nonnegative integer variables (discrete variables), representing operation parameters (positions and lexical values of symbols). Therefore, atomic constraints $x_i - x_j < c$, $x_i < c$ and $-x_i < c$ are equivalent to $x_i - x_j \leq c - 1$, $x_i \leq c - 1$ and $-x_i \leq c - 1$, respectively. Moreover, we are interested in triangular constraints (i.e., all atomic constraints are supposed to be of the form $x_i - x_j \leq c$). In the rest of the paper, for simplicity, we will invariantly use atomic constraints or their abbreviations.

A difference bound matrix is used to represent a set of atomic constraints. Given a set of atomic constraints A over the set of variables X . The DBM of A is the square matrix M of order $|X|$, where m_{ij} is the upper bound of the difference $x_i - x_j$ in A . By convention $m_{ii} = 0$, for every $x_i \in X$. In case, there is no constraint in A on $x_i - x_j$ ($i \neq j$), m_{ij} is set to ∞ . For example, we report, in Table 1 the DBM M of the following set of atomic constraints:

$$A = \{x_2 - x_1 \leq 5, x_1 - x_2 \leq -1, x_3 - x_1 \leq 3, x_1 - x_3 \leq 0\}.$$

Though the same nonempty domain may be expressed by different sets of atomic constraints, their DBMs have a unique form called *canonical form*. The canonical form of a DBM is the representation with tightest bounds on all differences between variables. It can be computed, in $O(n^3)$, n being the number of variables in the DBM, using a shortest-path algorithm, like Floyd-Warshall's all-pairs shortest-path algorithm [1]. As an example, Table 2 shows the canonical form M' of the DBM M . Canonical forms make easier some operations over DBMs like the test of equivalence. Two sets of atomic constraints are equivalent iff the canonical forms of their DBMs are identical.

A set of atomic constraints may be inconsistent (i.e., its domain is empty). To verify the consistency of a set of atomic constraints, it suffices to apply a shortest-path algorithm and to stop the algorithm as soon as a negative cycle is detected. The presence of negative cycles means that the set of atomic constraints is inconsistent.

In the context of our work, we use, in addition to the test of equivalence, three other basic operations on DBMs: adding a constraint to a set of constraints, incrementing/decrementing a variable in a set of constraints. We establish, in the following, computation procedures for these operations which do not need any operation of canonicalization (computing canonical forms).

Let $X = \{x_1, \dots, x_n\}$ be a finite and nonempty set of nonnegative integer variables, A a consistent set of triangular constraints over X , M the DBM, in canonical form, of A , x_i and x_j two distinct variables of X .

Incrementing by 1 a variable x_i in A is realized by replacing x_i with $x_i - 1$ (*old* $x_i = \textit{new } x_i - 1$). Using the DBM M , in canonical form, of A , this incrementation consists of adding 1 to each element of the line x_i and subtracting 1 from each element of the column x_i . Intuitively, this corresponds to replacing each constraint

$x_i - x_j \leq m_{ij}$ with $x_i - x_j \leq m_{ij} + 1$ and each constraint $x_j - x_i \leq m_{ji}$ with $x_j - x_i \leq m_{ji} - 1$. The resulting set of constraints and its DBM are denoted $A_{[x_i++]}$ and $M_{[i++]}$, respectively. The complexity of this operation is $O(n)$.

Similarly, to subtract 1 from a variable x_i in A , it suffices to replace x_i with $x_i + 1$ (*old* $x_i = \text{new } x_i + 1$). Using the DBM M , in canonical form, of A , this operation consists of subtracting 1 from each element of the line x_i and adding 1 to each element of the column x_i . The resulting set of constraints and its DBM are denoted $A_{[x_i--]}$ and $M_{[i--]}$, respectively. This operation is also of complexity $O(n)$. The following theorem establishes that $M_{[i++]}$ and $M_{[i--]}$ are in canonical form too. There is not need to compute their canonical forms.

Theorem 1. (i) $A \cup \{x_i - x_j \leq c\}$ is consistent iff $m_{ji} + c \geq 0$. If $A \cup \{x_i - x_j \leq c\}$ is consistent, its DBM M' , in canonical form, can be computed from M as follows: $M' = M$ if $m_{ij} \leq c$, and $(\forall k, l \in [1, n], m'_{kl} = \text{Min}(m_{kl}, m_{ki} + c + m_{jl}))$ otherwise. (ii) $M_{[i++]}$ and $M_{[i--]}$ are in canonical form.

Proof. (i) A can be represented by a weighted and oriented graph where each constraint $x_l - x_k \leq d$ of A is represented by the edge (x_l, x_k, d) . Since A is consistent, its graph does not contain any negative cycle. Therefore, $A \cup \{x_i - x_j \leq c\}$ is consistent iff, in its graph, the shortest cycle going through edge (x_i, x_j, c) is nonnegative (i.e., $m_{ji} + c \geq 0$). If $A \cup \{x_i - x_j \leq c\}$ is consistent, then $\forall k, l \in [1, n], m'_{kl}$ is the weight of the shortest path connecting x_k to x_l , i.e., $m'_{kl} = \text{Min}(m_{kl}, m_{ki} + c + m_{jl})$. By assumption M is in canonical form. It follows that $m_{kl} \leq m_{ki} + m_{ij} + m_{jl}$ and then: $m_{ij} \leq c$ implies that $m'_{kl} = m_{kl}$.

(ii) M is in canonical form iff $\forall j, k, l \in [1, n], m_{jk} \leq m_{jl} + m_{lk}$.

We give the proof for $M_{[i++]}$. The proof for $M_{[i--]}$ is similar. By definition,

$$\forall j, k \in [1, n], m_{[i++]_{jk}} = \begin{cases} m_{jk} & \text{if } j \neq i \wedge k \neq i \\ m_{jk} + 1 & \text{if } j = i \wedge k \neq i \\ m_{jk} - 1 & \text{if } j \neq i \wedge k = i \\ 0 & \text{if } j = i \wedge k = i \end{cases}$$

It follows that:

- If $j \neq i, k \neq i$, and $l \neq i$ then: $m_{[i++]_{jk}} = m_{jk}$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} + m_{lk}$.
- If $j \neq i, k \neq i$, and $l = i$ then: $m_{[i++]_{jk}} = m_{jk}$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} - 1 + m_{lk} + 1$.
- If $j = i, k \neq i$, and $l \neq i$ then: $m_{[i++]_{jk}} = m_{jk} + 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} + 1 + m_{lk}$.
- If $j = i, k \neq i$, and $l = i$ then: $m_{[i++]_{jk}} = m_{jk} + 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = 0 + m_{jk} + 1$.
- If $j \neq i, k = i$, and $l \neq i$ then: $m_{[i++]_{jk}} = m_{jk} - 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} + m_{lk} - 1$.
- If $j \neq i, k = i$, and $l = i$ then: $m_{[i++]_{jk}} = m_{jk} - 1$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = m_{jl} - 1 + 0$.
- If $j = i, k = i$, and $l = i$ then: $m_{[i++]_{jk}} = 0$ and $m_{[i++]_{jl}} + m_{[i++]_{lk}} = 0$.

By assumption $m_{jk} \leq m_{jl} + m_{lk}$. Then $m_{[i++]_{jk}} \leq m_{[i++]_{jl}} + m_{[i++]_{lk}}$. \square

The complexity of the consistency test of $A \cup \{x_i - x_j \leq c\}$ is $O(1)$. The computation complexity of the canonical form of its DBM is reduced to $O(n^2)$.

For instance, consider the set of constraints A of the previous example and its DBM, in canonical form, M' . According to Theorem 1, $A \cup \{x_2 - x_3 \leq 0\}$ is consistent iff $m'_{32} + 0 \geq 0$ (i.e., $2 \geq 0$). We give in Table 1 DBMs M'' of $A \cup \{x_2 - x_3 \leq 0\}$ and M''_{2++} .

Table 1. Some examples of DBMs

M	x_1	x_2	x_3	M'	x_1	x_2	x_3	M''	x_1	x_2	x_3	$M'_{[2++]}$	x_1	x_2	x_3
x_1	0	-1	0	x_1	0	-1	0	x_1	0	-1	-1	x_1	0	-2	0
x_2	5	0	∞	x_2	5	0	5	x_2	3	0	0	x_2	4	0	1
x_3	3	∞	0	x_3	3	2	0	x_3	3	2	0	x_3	3	1	0

3.2 Our Symbolic Model

Our model of OT-based collaborative editor is a network of communicating extended automata. A communicating extended automaton is an automaton extended with finite sets of variables, binary channels of communication, guards and actions. In such automata, edges are annotated with selections, guards, synchronization signals and blocks of actions. Selections bind non-deterministically a given identifier to a value in a given range (type). The other three labels of an edge are within the scope of this binding. A state is defined by the current location and current values of all variables. An edge is enabled in a state if and only if the guard evaluates to true. The block of actions of an edge is executed atomically when the edge is fired. The side effect of this block changes the state of the system. Edges labelled with complementary synchronization signals over a common channel must synchronize. Two automata synchronize through channels with a sender/receiver syntax [2]. For a binary channel, a sender can emit a signal through a given channel Syn ($Syn!$), if there is another automaton ready to receive the signal ($Syn?$). Both sender and receiver synchronize on execution of complementary actions $Syn!$ and $Syn?$. The update of the sender is executed before the update of the receiver.

An OT-based collaborative editor is composed of two or more sites (users) which communicate via a network and use the principle of multiple copies, to share some object (a text). Initially, each user has a copy of the shared object. It can afterwards modify its copy by executing operations generated locally and those received from other users. When a site executes a local operation, it is broadcast to all other users. The execution of a non local operation consists of the integration and the transformation steps as explained in Sub-section 2.4. To avoid managing queues of messages, the network is abstracted by allowing access to all operations and all timestamp vectors (declared as global variables). We propose also to abstract away the shared object and managing symbolically, using DBMs, the update operations.

Our OT-based collaborative editor model consists of one automaton per site, named *Site*, and an automaton named *Integration* devoted to the integration procedure and the verification of properties $TP1$ and $TP2$. Each automaton has only one parameter which is also an implicit parameter of all functions defined in the automaton. The parameter of automaton *Site* is the site identifier named *pid*. The parameter of automaton *Integration* is the property $TP1$ or $TP2$ to be verified. These automata communicate via shared variables and a binary channel named *Syn*.

3.3 Automaton *Site*

This automaton, depicted in Fig 4 (left), is devoted to generate, using timestamp vectors of different sites ($V[NbSites][NbSites]$), all possible execution orders of operations

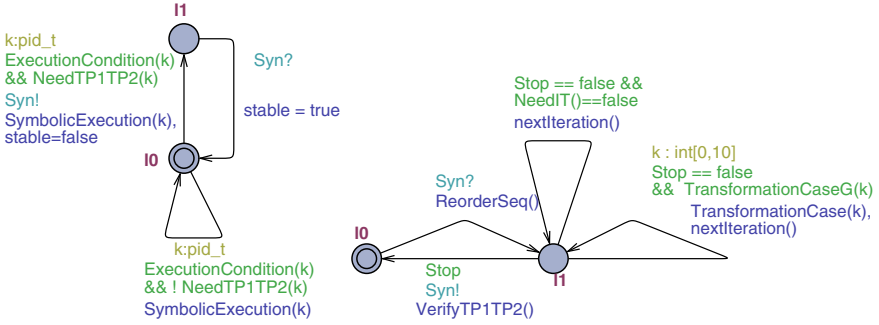


Fig. 4. Automata Site (*const pid_t pid*) and Integration (*const Property prop*)

(traces), respecting the causality principle. The loop on the initial location l_0 specifies the symbolic execution of an operation leading to a situation where there is no need to verify properties $TP1$ and $TP2$. The transition from l_0 to l_1 corresponds to the symbolic execution of an operation that needs the verification of $TP1$ or $TP2$. The boolean function $ExecutionCondition(k)$ verifies whether a site pid can execute, according to the causality principle, an operation o of site k (k in $pid.t$). The function $SymbolicExecution(k)$ adds o to the symbolic trace of site pid , sets the timestamp vector of o to $V[pid]$, and then updates $V[pid]$ (i.e., $V[pid][pid]++$).

The integration of operations is performed by automaton *Integration* when there is a need to verify $TP1$ or $TP2$ (i.e., function $NeedTP1TP2(k)$ returns *true*). For $TP1$, $NeedTP1TP2(k)$ returns *true*, if the execution, by site pid , of an operation o_1 of site k leads to a state where there is another site j s.t. traces of pid and j are $[seq_1; o_0; o_1]$ and $[seq_2; o_1; o_0]$, respectively, seq_1 and seq_2 are two equivalent operation sequences, o_0 and o_1 are two concurrent operations. This function returns *true* for $TP2$, if the execution of an operation o_2 by site pid leads to a state where there is another site j s.t. traces of pid and j are $[seq_1; o_0; o_1; o_2]$ and $[seq_2; o_1; o_0; o_2]$, respectively, seq_1 and seq_2 are two equivalent sequences, o_0, o_1 and o_2 are pairwise concurrent operations.

3.4 Automaton *Integration*

This automaton, depicted in Fig. 4 (right), is devoted to the verification of properties $TP1$ and $TP2$ on two sequences of operations. The verification starts when it receives a signal Syn from any site. It consists of applying the integration procedure to each sequence and then verifying that the resulting sequences satisfy the property $TP1$ or $TP2$. As explained in Section 2.4, the integration procedure of a non local operation o , in a sequence seq , consists of two steps: 1) computing a sequence seq' equivalent to seq , where operations dependent of o precede the concurrent ones (this is the role of function $ReorderSeq()$), and 2) transforming o against seq' (i.e. $IT^*(o, seq')$), realized by loops on location l_1 and functions $TransformationCaseG(k)$, $TransformationCase(k)$ and $NextIteration()$. The loop containing $NeedIT() == false$ is executed if o does not need to be transformed against the current operation of seq' . Otherwise, the other loop is executed and o is

transformed against the current operation of seq' . The verification of properties $TP1$ and $TP2$ on two sequences is performed by $VerifyTP1TP2()$, when the transformation process is completed for both sequences. The transformation and the verification are symbolic in the sense that operations are manipulated symbolically using DBMs.

3.5 Symbolic Transformation

The transformation procedure is applied when there is a need to verify $TP1$ or $TP2$ on traces of two sites. To handle symbolically the update operations of these traces, we use a DBM over the positions of the original operations, their copies and also symbols of the original operations. Initially, there is no constraint on symbols of operations and the position of each original operation is identical to those of its copies. Note that, the transformation of an operation does not affect the symbols, but, in some IT functions, the transformation procedures depend on symbols. So, there is no need to represent, symbols of the copies of operations, since they are always equal to the original ones.

Let us explain, by means of an example, how to handle and transform symbolically an operation against another operation. Suppose that we need to verify $TP1$ on sequences $seq_0 = [o_0; o_1]$ of site 0 and $seq_1 = [o_1; o_0]$ of site 1, where operations o_0 and o_1 are concurrent and generated at sites 0 and 1, respectively. The automaton *Integration* starts by calling function $ReorderSeq()$ to reorder sequences seq_0 and seq_1 as explained in Section 2.4 and create the initial DBM of the set of constraints $A = \{p_0 = p'_0 = p''_0, p_1 = p'_1 = p''_1\}$, where $p_i, p'_i, p''_i, i \in \{0, 1\}$ represent positions of operations o_i and its copies o'_i and o''_i , respectively. Afterwards, two transformations are performed sequentially by the automaton (loops on location $l1$): $IT(o'_1, o'_0)$ for Seq_0 and $IT(o''_0, o''_1)$ for Seq_1 .

For $IT(o'_1, o'_0)$, the process offers different possibilities of transformation, through the selection block on k , which are explored exhaustively. Each value of k corresponds to a case of transformation. For instance, for Ellis's IT function, the eleven cases of transformation are shown in Fig 5. These cases are trivially derived from Ellis's IT algorithm given in Fig. 3. Note that initially, the kinds of operations are not fixed. They will be fixed when a case of transformation is selected. For example, $k = 10$ corresponds to the case where o'_0 is a delete operation, o'_1 is an insert operation and $p'_1 = p'_0$. The function $TransformationCaseG(10)$ returns true iff the set of constraints $A \cup \{p'_0 = p'_1\}$ is consistent, o'_0 is either a delete operation or not fixed yet⁶ (*Nfx*), and o'_1 is either an insert operation or not fixed yet. In our case $TransformationCaseG(10)$ returns true and the function $TransformationCase(10)$ adds the constraint $p'_0 = p'_1$ to A (i.e., $A = \{p_0 = p'_0 = p''_0 = p_1 = p'_1 = p''_1\}$), decrements p'_1 in the resulting A (i.e., $A = \{p_0 = p'_0 = p''_0 = p_1 = p''_1, p_1 - p'_1 = 1\}$), sets o_0, o'_0, o''_0 to delete operations, and o_1, o'_1, o''_1 to insert operations.

For $IT(o''_0, o''_1)$, the automaton *Integration* offers only one possibility of transformation corresponding to the case fixed by the previous transformation, i.e., o''_0 is a delete operation, o''_1 is an insert operation and $p''_0 = p''_1$. In this case, the function $TransformationCase$ adds the constraint $p''_0 = p''_1$ to A (i.e., $A = \{p_0 = p'_0 = p''_0 = p_1 = p'_1 = p''_1, p_1 - p'_1 = 1\}$) and increments p''_0 in the resulting A , (i.e., $A = \{p_0 = p'_0 = p_1 = p'_1 = p''_1, p_1 - p'_1 = 1, p_0 - p''_0 = -1\}$) (see Fig 6 and Fig 7).

⁶ *Nfx* means that the operation type is not fixed yet and then can be set to *Del* or *Ins*.

Functions TransformationCaseG(int k) and TransformationCase(int k) for Ellis's IT algorithm
 TransformationCaseG(k) tests whether a transformation numbered k can be applied;
 Function TransformationCase(k) applies the transformation k (IT(o1',o0'))

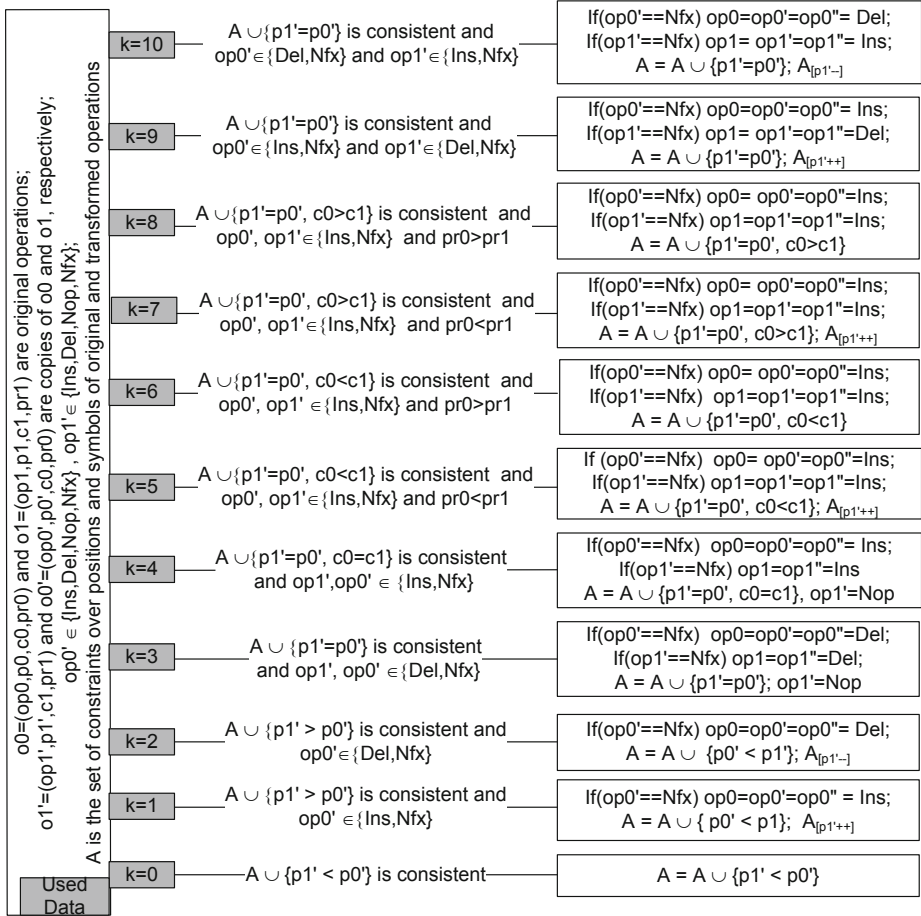


Fig. 5. Symbolic IT algorithm of Ellis

3.6 Verification of TP1 and TP2

Property TP1 ensures that the execution of two operations o_0 and o_1 in different orders, on two identical copies of a text, has the same effect. To compare the effects of sequences $[o'_0; o'_1]$ and $[o''_0; o''_1]$ on two identical copies of a text, it suffices to determine the final positions, in each copy, of all parts affected, when both operations are executed. The affected parts must be the same in both copies. Since the operations are handled symbolically (represented by a set of atomic constraints A (i.e., DBMs)), the effect of these symbolic operations must be also represented by a set of constraints. Property

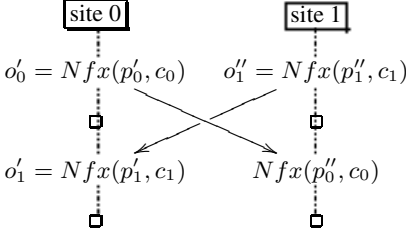


Fig. 6. Before integration: $A = \{p_0 = p'_0 = p''_0, p_1 = p'_1 = p''_1\}$

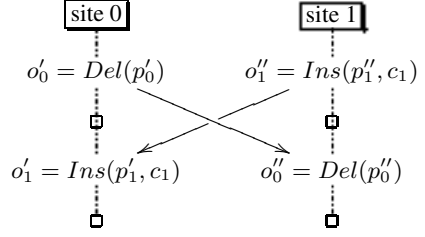


Fig. 7. After Integration for $k = 10$: $A = \{p_0 = p'_0 = p_1 = p''_1 = p'_1 + 1 = p''_0 - 1\}$

$TP1$ is not satisfied if the number of idle operations differs from one sequence to the other. It is also not satisfied if there is an idle operation in each sequence and the remaining ones are of different types, their positions are different or their symbols are different.

For the other cases, let us first explain how to verify $TP1$ on our previous example (see Fig. 7). Since, in $A = \{p_0 = p'_0 = p_1 = p''_1 = p'_1 + 1 = p''_0 - 1\}$, constraint $p'_1 < p'_0$ is always satisfied, it follows that after executing the sequence $[Del(p'_0); Ins(p'_1, c_1)]$, the positions of the deleted and the inserted elements are $p'_0 + 1$ and p'_1 , respectively. In A , constraint $p''_1 > p''_0$ is also always satisfied. Therefore, after executing the sequence $[Ins(p''_1, c_1); Del(p''_0)]$, the inserted and the deleted elements are at positions p''_1 and p''_0 , respectively. Property $TP1$ is satisfied iff each valuation of the domain of A , satisfies the both constraints: $p'_0 + 1 = p''_0$ and $p'_1 = p''_1$, i.e., $A = A \cup \{p'_0 + 1 = p''_0, p'_1 = p''_1\}$. Since, in A , we have $p'_1 \neq p''_1$, it follows that $TP1$ is not satisfied for Ellis's IT algorithm and then the previous example is a symbolic counterexample for $TP1$. Each nonnegative valuation of positions p_0 and p_1 that satisfies constraints of A but do not satisfy constraints of $A \cup \{p'_0 = p''_0, p'_1 = p''_1\}$ corresponds to a concrete counterexample. As an example, for $p_0 = p_1 = 2$, we obtain the counterexample where sequences executed by sites 0 and 1 are $[Del(2); Ins(1, c_1)]$ and $[Ins(2, c_1); Del(3)]$, respectively. These sequences lead to a divergence. For instance, if the initial text is $abcde$, the previous sequences lead to two different copies: ac_1bde and abc_1de .

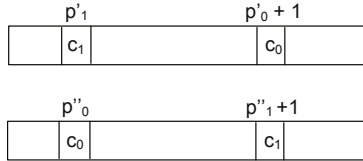
To verify $TP1$, function $VerifyTP1TP2$ partitions the domain of A in three or four partitions as shown in Table 2 and then determines, for each consistent partition, the positions of the inserted/deleted elements when both sequences are executed. We give, in Table 2, the different partitions of A and the associated conditions that guarantee to get the same effect on both copies of the shared text, for different sequences of two non idle operations. As an example, suppose two sequences $seq' = [Ins(p'_0, c_0); Ins(p'_1, c_1)]$ and $seq'' = [Ins(p''_1, c_1); Ins(p''_0, c_0)]$. Let us compare the effects of these sequences. The positions of the inserted elements, when seq' is executed, depend on the relationships between p'_0 and p'_1 : $p'_0 + 1$ and p'_1 , if $p'_0 \geq p'_1$; and p'_0 and p'_1 otherwise. Similarly, the positions of the inserted elements when seq'' is executed are: $p''_1 + 1$ and p''_0 , if $p''_1 \geq p''_0$; and p''_1 and p''_0 otherwise. Therefore, to compare the effects of both sequences, four cases are considered (see Fig. 8). Each case has its own condition of convergence. For example, for $p'_0 \geq p'_1 \wedge p''_1 \geq p''_0$, the condition of convergence is $A \cup \{p'_0 \geq p'_1, p''_1 \geq p''_0\} = A \cup \{p'_0 \geq p'_1, p''_1 \geq p''_0, p'_0 = p''_1, p'_1 = p''_0, c_0 = c_1\}$.

Table 2. Symbolic verification of TP1

$[Ins(p'_0, c_0); Ins(p'_1, c_1)] \parallel [Ins(p''_1, c_1); Ins(p''_0, c_0)]$	
Partitions of A	Convergence condition
$A_1 = A \cup \{p'_1 \leq p'_0, p''_1 < p''_0\}$	$A_1 = A_1 \cup \{p'_0 + 1 = p''_0, p'_1 = p''_1\}$
$A_2 = A \cup \{p'_1 \leq p'_0, p''_1 \geq p''_0\}$	$A_2 = A_2 \cup \{p'_0 = p''_1, p'_1 = p''_0, c_0 = c_1\}$
$A_3 = A \cup \{p'_1 > p'_0, p''_1 < p''_0\}$	$A_3 = A_3 \cup \{p'_0 = p''_1, p'_1 = p''_0, c_0 = c_1\}$
$A_4 = A \cup \{p'_1 > p'_0, p''_1 \geq p''_0\}$	$A_4 = A_4 \cup \{p'_0 = p''_0, p'_1 + 1 = p''_1\}$

$[Del(p'_0); Del(p'_1)] \parallel [Del(p''_1); Del(p''_0)]$	
Partitions of A	Convergence condition
$A_1 = A \cup \{p'_1 < p'_0, p''_1 \leq p''_0\}$	$A_1 = A_1 \cup \{p'_0 = p''_0 + 1, p'_1 = p''_1\}$
$A_2 = A \cup \{p'_1 < p'_0, p''_1 > p''_0\}$	$A_2 = A_2 \cup \{p'_0 = p''_1, p'_1 = p''_0\}$
$A_3 = A \cup \{p'_1 \geq p'_0, p''_1 \leq p''_0\}$	$A_3 = A_3 \cup \{p'_0 = p''_1, p'_1 = p''_0\}$
$A_4 = A \cup \{p'_1 \geq p'_0, p''_1 > p''_0\}$	$A_4 = A_4 \cup \{p'_0 = p''_0, p'_1 = p''_1 + 1\}$

$[Del(p'_0); Ins(p'_1, c_1)] \parallel [Ins(p''_1, c_1); Del(p''_0)]$	
Partitions of A	Convergence condition
$A_1 = A \cup \{p'_1 < p'_0\}$	$A_1 = A_1 \cup \{p'_0 + 1 = p''_0, p'_1 = p''_1\}$
$A_2 = A \cup \{p'_1 \geq p'_0, p''_1 \leq p''_0\}$	$A_2 = A_2 \cup \{p'_0 = p''_1, p'_1 + 1 = p''_0, p'_0 = p''_1\}$
$A_3 = A \cup \{p'_1 \geq p'_0, p''_1 > p''_0\}$	$A_3 = A_3 \cup \{p'_0 = p''_0, p'_1 = p''_1 - 1\}$

**Fig. 8.** Effect of seq' and seq'' in case $p'_0 \geq p'_1 \wedge p''_1 \geq p''_0$

The verification of property $TP2$ is much more simpler than $TP1$. Let $[seq_0; o_0; o_1; o_2]$ and $[seq_1; o_1; o_0; o_2]$ be a pair of equivalent sequences, such that o_0, o_1 and o_2 are pairwise concurrent operations. Verifying $TP2$ consists of testing that o_2 is transformed in the same manner against sequences $[seq_0; o_0; o_1]$ and $[seq_1; o_1; o_0]$.

For both properties, function $VerifyTP1TP2$ sets a boolean variable named *Detected* to *true* as soon as the violation of property $TP1$ or $TP2$ is detected. This variable is initially set to *false*.

Our approach is sound and complete w.r.t. to the convergence property as it generates only feasible traces (in respect with the causality principle) and the integration procedure is performed exactly as in OT-based collaborative editors. Properties $TP1$ and $TP2$ are verified on feasible traces in conformity with their definitions.

We have used the on-the-fly model-checker UPPAAL⁷ to test the symbolic model proposed here. The Computation tree logic (CTL) formula [5] *AG not Detected* allows us to verify whether or not property $TP1$ or $TP2$ is satisfied. We have considered several IT functions: *Ellis* [4], *Ressel* [8], *Sun* [12], *Suleiman* [9] and *Imine* [6]. To test different IT functions, it suffices to rewrite accordingly functions

⁷ www.uppaal.com

Table 3. Verification of $TP1$ and $TP2$ for five IT functions proposed in the literature

IT function	Verification of TP1 and TP2
Ellis	$TP1$ $false, o_0 \in \{Ins(p_0, c_0), Del(p_0)\}, o_1 = Ins(p_1, c_1), p_1 < p_0$
Sizes / Time	234 / 157 / 0.452
	$TP2$ $false, o_0 = Del(p_0), o_1 = Ins(p_0 - 1, c_1), o_2 = Ins(p_0, c_1)$
Sizes / Time	667 / 561 / 1.029
Ressel	$TP1$ $true$
Sizes / Time	788 / 788 / 0.811
	$TP2$ $false, o_0 = Del(p_0), o_1 = Ins(p_0 + 1, c_1), o_2 = Ins(p_0, c_2)$
Sizes / Time	477 / 413 / 0.686
Sun	$TP1$ $false, o_0 \in \{Ins(p_0, c_0), Del(p_0)\}, o_1 = Ins(p_1, c_1), p_1 < p_0$
Sizes / Time	225 / 156 / 0.405
	$TP2$ $false, o_0 = Ins(p_0, c_0), o_1 = Del(p_0 - 1), o_2 = Ins(p_0 - 1, c_2)$
Sizes / Time	477 / 413 / 0.717
Suleiman	$TP1$ $true$
Sizes / Time	1023 / 1023 / 1.060
	$TP2$ $false, o = Del(p_0 - 1), o_0 = Ins(p_0, c_0), o_1 = Ins(p_0 - 1, c_1),$ $o_2 = Ins(p_0 - 1, c_2), c_1 < c_2 < c_0$
Sizes / Time	10961 / 9913 / 2.124
Imine	$TP1$ $true$
Sizes / Time	963 / 963 / 1.130
	$TP2$ $false, o = Del(p), o_0 = Del(p_0), o_1 = Ins(p_0, c_1),$ $o_2 = Ins(p_0, c_2), p + 1 \leq p_0, c_2 < c_1$
Sizes / Time	9730 / 8808 / 2.130

$TransformationCaseG(k)$ and $TransformationCase(k)$. We give, in Table 3, the results obtained for both properties $TP1$ and $TP2$ in the case of four operations o_0, o_1, o_2, o and three sites. All operations are pairwise concurrent, except that o_2 is causally dependent of o . The property $TP2$ is not satisfied for all considered IT functions. A symbolic counterexample is provided for each unsatisfied property. We report also the number of explored/computed abstract states and the time, in second, of the verification, under UPPAAL, of CTL formula $AG \text{ not Detected}$.

4 Conclusion

We have proposed here a symbolic model-checking technique to verify that an OT algorithm used, in replication-based collaborative editors ensures convergence of replicas. In our technique, the shared objects are abstracted and their update operations are handled symbolically using difference bound matrices. Unlike in [3], there is no need to fix neither the shared object nor sizes of parameters of its update operations. Unlike in [7], our approach allows us to provide symbolic feasible counterexamples for the convergence property. Indeed, in [7], the verification of convergence is not based on only feasible traces. Consequently, it is sound but not complete. Our approach is sound and complete. However, its termination needs to fix the numbers of sites and operations. We plan to determine, if they exist, the smallest values for m and n s.t. an OT algorithm

ensures convergence for m operations and n sites implies that the OT algorithm ensures convergence for any arbitrary numbers of operations and sites.

References

1. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone-based abstractions of timed automata. *Theoretical Computer Science* 8(3) (2006)
2. Bérard, B., Bouyer, P., Petit, A.: Analysing the pgm protocol with UPPAAL. *International Journal of Production Research* 42(14), 2773–2791 (2004)
3. Boucheneb, H., Imine, A.: On model-checking optimistic replication algorithms. In: *FMOODS/FORTE*, pp. 73–89 (2009)
4. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: *SIGMOD Conference*, vol. 18, pp. 399–407 (1989)
5. Emerson, E.A.: Temporal and modal logic. In: *Handbook of Theoretical Computer Science*, ch. 16. Formal Methods and Semantics, vol. B (1990)
6. Imine, A., Molli, P., Oster, G., Rusinowitch, M.: Proving correctness of transformation functions in real-time groupware. In: *ECSCW 2003*, Helsinki, Finland (September 14–18, 2003)
7. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science* 351(2), 167–183 (2006)
8. Ressel, M., Nitsche-Ruhland, D., Gunzenhauser, R.: An integrating, transformation-oriented approach to concurrency control and undo in group editors. In: *ACM CSCW 1996*, Boston, USA, pp. 288–297 (November 1996)
9. Suleiman, M., Cart, M., Ferrié, J.: Serialization of concurrent operations in a distributed collaborative environment. In: *ACM GROUP 1997*, pp. 435–445 (November 1997)
10. Suleiman, M., Cart, M., Ferrié, J.: Concurrent operations in a distributed and mobile collaborative environment. In: *IEEE ICDE 1998*, pp. 36–45 (1998)
11. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms and achievements. In: *ACM CSCW 1998*, pp. 59–68 (1998)
12. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.* 5(1), 63–108 (1998)
13. Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., Cai, W.: Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.* 13(4), 531–582 (2006)
14. Vidot, N., Cart, M., Ferrié, J., Suleiman, M.: Copies convergence in a distributed real-time collaborative environment. In: *ACM CSCW 2000*, Philadelphia, USA (December 2000)

From Operating-System Correctness to Pervasively Verified Applications*

Matthias Daum¹, Norbert W. Schirmer², and Mareike Schmidt¹

¹ Computer Science Dept., Saarland University
66123 Saarbrücken, Germany

{md11, mareike}@wjpserver.cs.uni-saarland.de

² German Research Center for Artificial Intelligence (DFKI)
66041 Saarbrücken, Germany
Norbert.Schirmer@dfki.de

Abstract. Though program verification is known and has been used for decades, the verification of a complete computer system still remains a grand challenge. Part of this challenge is the interaction of application programs with the operating system, which is usually entrusted with retrieving input data from and transferring output data to peripheral devices. In this scenario, the correct operation of the applications inherently relies on operating-system correctness. Based on the formal correctness of our real-time operating system OLOS, this paper describes an approach to pervasively verify applications running on top of the operating system.

1 Introduction

Various electronic devices are embedded in the modern car, and some are even in charge of safety-critical tasks like accelerator control. In the past years, a failing accelerator control has caused several fatal accidents [1]. Though the manufacturer has attributed these failures to a blocked gas pedal, a software problem has recently been suspected for the sudden, unintended acceleration of a car from the same manufacturer while driven by cruise control [2]. The mere rumor of such a software flaw is economically troublesome, not to mention the tragedy of possibly resulting fatal accidents.

There are different approaches to increase the reliability of software. A rigorous way to prevent flaws is the exclusion of systematic errors by verification. If the proofs are checked by a computer, we speak of *formal verification*. Certainly, this method should not be limited to a single system layer but span as many layers as possible. *Pervasive verification* means that the system layers are coupled by formal soundness and simulation theorems, such that any verification result, obtained on a suitable layer, can ultimately be transferred down to a correctness theorem on the lowest level. While program verification has been known

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

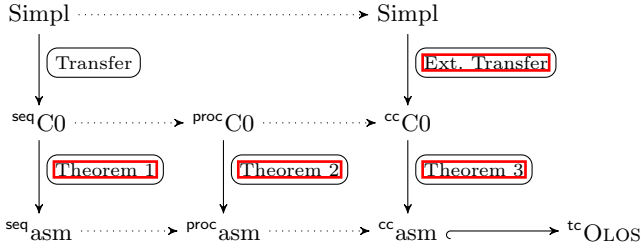


Fig. 1. Extending the Language Stack towards Concurrency

and used over decades, the pervasive formal verification of complete computer systems still remains a grand challenge [3].

Among others [4], the Verisoft project takes on this challenge. The goal of its automotive subproject is a pervasively verified distributed real-time system, consisting of hardware, a real-time operating system, and application programs. We have implemented the operating system OLOS on a verified processor [5] using a generic programming framework for operating systems [6]. Moreover, we have proven the correctness of OLOS [7] in the proof assistant Isabelle/HOL [8].

In this paper, we report on the formally proven foundation of a verification approach for applications running on top of OLOS. We present the necessary theorems to transfer verification results down to the operating-system level and thus, establish a formal link between the proofs on the application layer and those on the operating-system layer. More specifically, our verification approach is an extension of an existing language stack [9] based upon a verified C compiler [10] and a generic verification framework for sequential imperative programs [11].

Our overall proof architecture is depicted in Fig. 1: The original language stack (see Sect. 2.3) is shown on the left column. On the lower end, we have the sequential assembly language ($^{\text{seq}}\text{asm}$), above is the C variant $C0$. Leinenbach & Petrova’s [10] correct compiler translates a sequential $C0$ program ($^{\text{seq}}C0$) to assembly. Schirmer [11] has developed the generic Simpl language together with Hoare logics and a transfer theorem stating that properties proven for Simpl actually hold in $C0$. The sequential semantics, however, have no means for communication. Thus, we extend $^{\text{seq}}C0$ and $^{\text{seq}}\text{asm}$ in Sect. 3 to application processes (marked by $^{\text{proc}}$ in the 2nd column), where the communication with OLOS is modeled by explicit inputs and outputs. We do not extend Simpl because it is solely used for reasoning in Hoare logic, which does not support inputs and outputs. Instead, we further extend the language stack in Sect. 4 to cooperative concurrent applications (marked by $^{\text{cc}}$ in the 3rd column). By *cooperative concurrency*, we refer to the sequential execution of an application with calls to the operating system until a final call of a synchronization primitive. Finally, we embed the lowest layer of this stack into a true concurrent model ($^{\text{tc}}\text{OLOS}$) of our operating system with an interleaved execution of applications (last column, Sect. 5).

In Sect. 6, we provide an application example demonstrating how our approach may be used in practice. We conclude in Sect. 7.

Notation. The formalizations presented in this article are mechanized and checked within the interactive theorem prover Isabelle/HOL [8]. This paper is written using Isabelle’s document generation facilities, which guarantees that the presented theorems correspond to formally proven ones [9]. We distinguish formal entities typographically from other text. We use a sans-serif font for types and constants (including functions and predicates), e. g., `map`, a slanted serif font for free variables, e. g., x , and a slanted sans-serif font for bound variables, e. g., x . Small capitals are used for data-type constructors, e. g., `EXFINISH`. Type variables have a leading tick, e. g., $'a$. Keywords are typeset in bold font, e. g., **let**.

The logical and mathematical notation mostly follows standard conventions; we only present the more unconventional parts here. We prefer curried function application, e. g., $f\ a\ b$ instead of $f\ (a,\ b)$. We write f^n for the n -fold composition of function f .

Isabelle/HOL provides a library of standard types like Booleans, natural numbers, integers, total functions, pairs, lists, and sets as well as packages to define new data types and records. Isabelle allows polymorphic types, e. g., $'a\ list$ is the list type with type variable $'a$. In HOL all functions are total, e. g., `nat` \Rightarrow `nat` is a total function on natural numbers. There is, however, a type $'a\ option$ to formalize partial functions. It is a data type with two constructors, one to inject values of the base type, e. g., $[x]$, and the additional element \perp . A base value can be projected by $[x]$, which is defined by the sole equation $[[x]] = x$. As HOL is a total logic, the term $[\perp]$ is still a valid yet unspecified value. Partial functions can be represented by the type $'a \Rightarrow 'b\ option$.

2 Background

2.1 On A Simple Real-Time Operating System

The continually increasing number and variety of electronic components in cars result in an even faster growing demand for communication channels. Over time, adding more and more wires has led to space, complexity and maintenance problems. Alternatively, several components can share the same wire and use a communication protocol on this bus. For that purpose, Kopetz and Grünsteidl [12] have developed the *time-triggered protocol*, which schedules fixed transmission times for each component on the bus. Variations of this protocol are nowadays widely accepted in industry.

We adopt this idea and assume a distributed system comprising a number of components that are connected via a communication bus. The components are called *electronic control units* (ECUs). Each ECU consists of a general-purpose RISC processor and an *automotive bus controller* (ABC). The latter takes care of the timely transmission and reception of messages. This device is responsible for clock synchronization, decoupling the processor from the communication bus.

On each processor there runs an instance of the operating system OLOS, providing a virtual processor abstraction to the applications that share the same

¹ For the theory files, see <http://www.verisoft.de/VerisoftRepository.html>

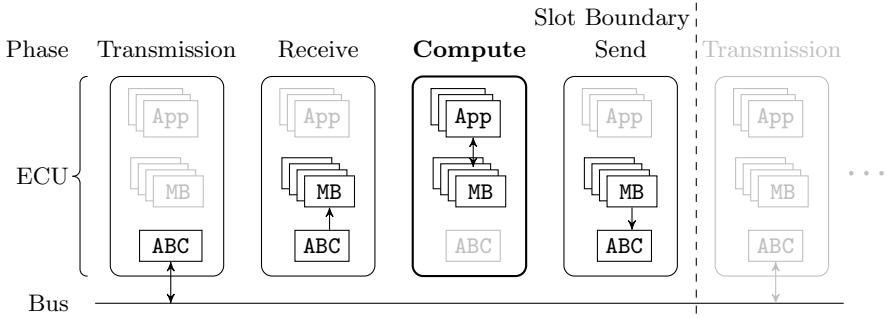


Fig. 2. The transition phases in each slot of our time-triggered computer system

physical processor. OLOS features its own message buffers (MB) for the communication between applications (on the same as well as on different physical processors). The schedule of the transmission times on the bus is statically fixed and repeated perpetually. A period, or *round*, is subdivided into equal time slices, the so-called *slots*.

Each slot is divided into four transition phases. Fig. 2 illustrates the data flow between the ECU components in the different phases:

Transmission. In this phase, the ABC device of one ECU transmits a message to the communication bus. According to a predefined schedule, exactly one ECU has the send permission in each slot. All ECUs listen on the bus to receive the transmitted message.

Receive. The operating system reads the receive buffer from the ABC device into one of its own message buffers.

Compute. A statically fixed table specifies, which application is executed during this phase of the current slot. The application may compute locally or exchange messages with OLOS. When the application has finished its computation for the current slot, it informs the operating system by a special system call EXFINISH.

Send. If the ECU is permitted to send, the operating system writes the corresponding message into the ABC’s send buffer.

2.2 Formally Specifying OLOS – The True Concurrent ECU Model

Correctness is usually defined as the compliance with a specification. In our case, this specification is an automaton. Note that OLOS relies on a specific protocol with the ABC. Hence, the abstract automaton describes the behavior of the whole ECU consisting of the processor with its running operating system and applications together with the ABC device. A state s of this ECU automaton comprises the application states $s.AM$, the message buffers $s.MB$, an ABC state ($s.abc_dev$), and an *idle flag* ($s.idle_flag$). The latter determines whether the application scheduled in the current slot has finished its computation.

The transitions of the abstract ECU automaton concisely specify our informal description of the ECU behavior as depicted in [Fig. 2](#). In Isabelle/HOL, we have formalized the transitions by the function $\delta_{\text{ECU}} t i$. The static schedule t determines for each slot, which message buffer should be sent, and which application should be scheduled. The input i distinguishes external device steps ($i = [e]$) from processor computation ($i = \perp$).

In this paper, we are primarily interested in the compute phase. The beginning of the compute phase is marked by the ECU turning into the *computing* state. The name is derived from the fact that all transitions starting in this state involve a computation of the application scheduled in this slot. In this state, the idle flag as well as the ABC's interrupt flag are unset. Several transitions are possible from this state:

- An external device input might raise the interrupt line of the ABC device. In this case, the currently scheduled application has exceeded its execution time. OLOS reacts exactly as if it had been waiting for the interrupt.
- If the application issues an EXFINISH call, the operating system acknowledges the call and waits for an interrupt. Formally, the ECU transition raises the idle flag. It thereby turns into an idle state waiting for an input $[e]$.
- Otherwise, the ECU simply remains in the computing state.

2.3 On a Correct Compiler – The Sequential Language Stack

ANSI C [\[13\]](#) has a complex and highly underspecified semantics. Low-level functionality like the communication with the operating system, however, inherently *relies* on properties of a particular compiler and a specific target hardware, e. g., register bindings or the internal representation of data types. They can therefore not be verified based only on the vague ANSI C semantics. Hence, Leinenbach and Petrova [\[10\]](#) specified the C-like imperative language *C0*, which has sufficient features to implement low-level software but is interpreted by a more specific semantics. Due to lack of space, we omit the details of the language and only glance at its formal semantics.

The C0 Small-Step Semantics. A *C0 program* is statically defined by a type-name table tt , a function table ft , and a symbol table gst of global variables.

In contrast to this static program definition, the program state evolves during the execution of a C0 program. A state s_{C0} comprises: (a) the statement $s_{\text{C0}}.\text{prog}$ of the program that remains to be executed, and (b) the current state $s_{\text{C0}}.\text{mem}$ of the program variables and the heap objects. The transition relation δ_{C0} of the C0 semantics is deterministic, i. e., a partial function.

The Target Language. Leinenbach & Petrova's verified C0 compiler translates C0 programs into the assembly language developed for the VAMP architecture [\[5\]](#). VAMP assembly abstracts from the paging mechanism of the processor and employs a linear memory model. An assembly state s_{asm} is a record comprising

two program counters² ($s_{\text{asm}}.\text{pcp}$ and $s_{\text{asm}}.\text{dpc}$), general-purpose as well as special-purpose register files ($s_{\text{asm}}.\text{gprs}$ and $s_{\text{asm}}.\text{sprs}$), and memory ($s_{\text{asm}}.\text{mm}$).

Assembly transitions are modeled by function δ_{asm} . Again, we omit the details of the semantics because of space restrictions. Note that the effects of hardware exceptions like accessing unavailable memory cannot be fully determined from an assembly-machine state. In this case, δ_{asm} gets stuck. With sufficient memory, however, there are no exceptions generated when a well-formed C0 program is compiled and executed.

On a Correct Compiler. Compiler correctness is formulated as a simulation theorem. The simulation relation `consistent` holds for corresponding C0 and assembly states. In essence, the compiler-simulation theorem states that every step i of the source program executed on the C0 semantics simulates a certain number s_i of steps of the VAMP assembly machine executing the compiled code.

The memory requirements can directly be checked on the C0 semantics. We assume that memory is available from 0 to an address $x \leq 2^{32}$. The predicate `sufficient_memory x tt ft sC0` holds iff x is large enough such that the C0 state s_{C0} of the program (tt, ft) can be encoded in assembly. Furthermore, we denote the successful (i. e., fault-free) execution from an assembly state s_{asm} in t steps to state s'_{asm} by $(\text{crange}, \text{arange}) \vdash_{\text{asm}} s_{\text{asm}} \rightarrow^t s'_{\text{asm}}$, where instructions are only read from range crange and only memory addresses in range arange are accessed. We can compute the ranges for a given C0 program by the functions `code_range` and `address_range`, respectively.

Theorem 1 (Stepwise Compiler Simulation). *We assume:*

- The C0 state s_{C0} is well-formed, i. e., `is_validC0 tt ft sC0`.
- The program counters of the well-formed assembly state s_{asm} do not start in a delay slot.³ i. e., `is_validasm sasm` and $s_{\text{asm}}.\text{pcp} = s_{\text{asm}}.\text{dpc} + 4$.
- The simulation relation holds for s_{C0} and s_{asm} under an allocation function `alloc`, i. e., `consistent tt ft sC0 alloc sasm`.
- The C0 transition from s_{C0} is defined to s'_{C0} , i. e., `δC0 tt ft sC0 = [s'C0]`.
- there is sufficient memory before and after the transition, i. e., $x \leq 2^{32}$, `sufficient_memory x tt ft sC0` and `sufficient_memory x tt ft s'_{C0}`.

Under these assumptions, there exists a step number n , an allocation function `alloc'`, and an assembly state s'_{asm} such that (a) the assembly machine successfully advances in n steps from s_{asm} to s'_{asm} , (b) the final C0 state s'_{C0} simulates s'_{asm} under the allocation function `alloc'`, and (c) no special-purpose registers have been changed. Formally:

$$\exists n \text{ alloc}' s'_{\text{asm}}. (\text{code_range } tt \text{ (gm_st } s_{\text{C0}}.\text{mem}) \text{ ft, address_range } x) \vdash_{\text{asm}} s_{\text{asm}} \rightarrow^n s'_{\text{asm}} \wedge \text{consistent } tt \text{ ft } s'_{\text{C0}} \text{ alloc}' s'_{\text{asm}} \wedge s'_{\text{asm}}.\text{sprs} = s_{\text{asm}}.\text{sprs}$$

² We need two program counters because branches are delayed by one instruction.

³ When a C0 statement has been completely executed, the assembly machine should certainly not be about to execute a previously seen branch.

Verifying C Programs – the Isabelle/Simpl Framework. The verification environment Isabelle/Simpl [11] is implemented as a conservative extension of the higher-order logic (HOL) instance of the theorem-proving environment Isabelle [8]. Though the verification environment was motivated by C0, it is by no means restricted to C0. In fact, it is a self-contained theory development for a quite generic model of a sequential imperative programming language called *Simpl*. Part of this extensive framework are big- and small-step semantics as well as Hoare logics for both, partial and total correctness. In order to facilitate the usage of the Hoare logics within Isabelle/HOL, the application of the rules is automated as a verification-condition generator. Furthermore, proofs have been developed that the logics are sound and complete with respect to the operational semantics. Soundness is crucial for pervasive verification in order to formally link the results from the Hoare logics to the operational Simpl semantics. Correctness theorems about the embedding of C0 into Simpl then allow us to map these results to the small-step semantics of C0 [11,14]. Completeness can be viewed as a sophisticated sanity check for the Hoare logics, ensuring that verification cannot get stuck because of missing Hoare rules.

3 Application Processes

As their most basic feature, operating systems provide a processor abstraction to applications with (a) an exclusive access to resources like registers and memory and (b) means for the communication with the operating system to request further services. This abstraction is commonly referred to as a *process*. As even most high-level programs are eventually compiled to run as a process, we consider any program semantics with primitives for the communication with an operating system as a process. In this section, we formally specify processes as automata with outputs and inputs, present two particular process semantics for C0 and VAMP assembly, and finally extend compiler correctness to processes.

3.1 Process Semantics

In general, we define:

Definition 1 (Process Semantics). *A process semantics is an automaton A_{proc} specified by a tuple $(S_{\text{proc}}, \text{is_valid}_{\text{proc}}, \text{is_init}_{\text{proc}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \delta_{\text{proc}}, \omega_{\text{proc}})$ with a state space S_{proc} , a validity predicate $\text{is_valid}_{\text{proc}}$, an initialization predicate $\text{is_init}_{\text{proc}}$, an input alphabet Σ_{proc} , an output alphabet Ω_{proc} , a transition function δ_{proc} , as well as an output function ω_{proc} .*

The state space S_{proc} depends on the underlying programming language – in our case, C0 or VAMP assembly. The communication interface, on the contrary, is determined by the operating system such that all OLOS processes possess the same in- and output alphabets. Table 1 presents both alphabets side by side. Note the strong correlation of outputs and inputs: When a process state features an output seen on the left, OLOS responds with one of the inputs shown in the

Table 1. Interface between OLOS and its application processes

Outputs Ω_{proc}	Inputs Σ_{proc}
ϵ_{Ω} (no call to a primitive)	ϵ_{Σ} (internal step)
SENDMSG $msgval\ msgnr$	SENDSUCCESS INVALIDMSGNR
RCVMSG $msgnr$	RCVSUCCESS $msgval$ INVALIDMSGNR
EXFINISH	FINISHSUCCESS
INVPTRERR	INVPTRRESPONSE
REPEATERR	—
CONTINUEERR	CONTINUE

same table row on the right. Formally, we collect the matching output-input pairs (ω, i) in the set `olos_responses`.

The predicate `is_initproc` mainly determines the set of initial states; it takes two parameters that constrain the initial memory of processes (effectively, specifying different subsets of initial states). We implicitly assume that the parameters fulfill basic validity constraints, which are formalized in the predicate `valid_params`. The predicate `is_validproc` formulates an invariant over the execution traces of processes.

Definition 2 (Validity of Process Semantics). *We call a process semantics valid, iff the invariant `is_validproc` holds for all initial states, i. e.,*

$$\llbracket \text{valid_params } img \text{ pages; is_init}_{\text{proc}} \text{ } img \text{ pages } s_{\text{proc}} \rrbracket \implies \text{is_valid}_{\text{proc}} s_{\text{proc}}$$

and furthermore, the invariant is preserved under transitions with valid inputs:

$$\llbracket \text{is_valid}_{\text{proc}} s_{\text{proc}}; (\omega_{\text{proc}} s_{\text{proc}}, i) \in \text{olos_responses} \rrbracket \implies \text{is_valid}_{\text{proc}} (\delta_{\text{proc}} i s_{\text{proc}})$$

3.2 Specifying the Semantics for C0 and Assembly Processes

In this section, we shortly glance at the specification of our two particular process semantics. There are several runtime errors, namely `INVPTRERR`, `REPEATERR`, and `CONTINUEERR`. As correct programs do not feature these errors, we omit further details, here. VAMP assembly provides a special instruction `TRAP n` for the communication of a process with an operating system. The process-output function $\omega_{\text{proc}} s_{\text{asm}}$ uses the number n to distinguish between the `SENDMSG`, the `RCVMSG`, and the `EXFINISH` primitive; a number not assigned to a primitive results in `CONTINUEERR`, i. e., the instruction will simply be skipped. The parameter $msgval$ is specified by a register pointing into the memory. Finally, the parameter $msgnr$ is directly taken from a register. If neither a runtime error occurred nor the next instruction is `TRAP n`, the process output is ϵ_{Ω} .

For a transition $\delta_{\text{proc}} s_{\text{asm}}$ with the empty input ϵ_{Σ} , the assembly process semantics employs the underlying, sequential assembly semantics δ_{asm} for its

transition. Otherwise, the response from OLOS is reflected by placing a corresponding value into a specified *response register*; and in case of RECVSUCCESS, the received message is additionally stored into the process memory.

For C0 processes, we have implemented functions with inline assembly that wrap the necessary assembly instructions for the communication with OLOS. For illustration, [Table 2](#) shows the implementation of the function `olosRecvMsg`, which wraps the system call RECVMSG.

The C0-process semantics treats a call to these functions as a primitive, i. e., the output function $\omega_{\text{proc}} s_{\text{C0}}$ simply determines whether the next statement is a call to such a wrapper function, and the transition function $\delta_{\text{proc}} i s_{\text{C0}}$ directly removes the function-call statement from the remaining program and updates the C0 state s_{C0} according to the input i .

3.3 Extending Compiler Correctness to Processes

Recall that Leinenbach & Petrova [\[10\]](#) have shown compiler correctness for the sequential C0 semantics with respect to the sequential part of VAMP assembly (cf. [Theorem 1](#)). Below, we extend their result to the two corresponding process semantics, which we have defined above.

First, we extend the existing simulation relation `consistent` for processes to `consisproc`. This is mainly a syntactic adaptation and therefore we omit the details. Second, we define predicates for the successful execution of processes analogous to the sequential counterpart for assembly machines. Intuitively, a successful execution is characterized by the absence of runtime errors (including the sufficiency of memory). Furthermore, process transitions take inputs from OLOS. We make the output-input sequence `ois` explicit and require that the outputs in the sequence equal the process output in the corresponding state as well as that the sequence only contains matching output-input pairs, i. e., all pairs in the sequence are contained in `olos_responses`. Successful execution, we denote as: $\vdash_{\text{C0}}^{\text{proc}} s_{\text{C0}} \xrightarrow{\text{ois}} s'_{\text{C0}}$ for C0 processes and $\text{crange} \vdash_{\text{asm}}^{\text{proc}} s_{\text{asm}} \xrightarrow{\text{ois}} s'_{\text{asm}}$ for assembly processes, respectively. We only need the code range `crange` to determine that the assembly code does not modify itself. The maximal address, in contrast, that we know from the sequential assembly semantics, is encoded in the process state.

Table 2. C0 implementation of the receive primitive

```
int olosRecvMsg (t_msg *msg_ptr, unsigned int msgnr) {
    int result;
    asm { lw(r11, r30, asm_offset(msg_ptr));
          lw(r12, r30, asm_offset(msgnr));
          trap(2);
          sw(r22, r30, asm_offset(result));
        };
    return result;
}
```

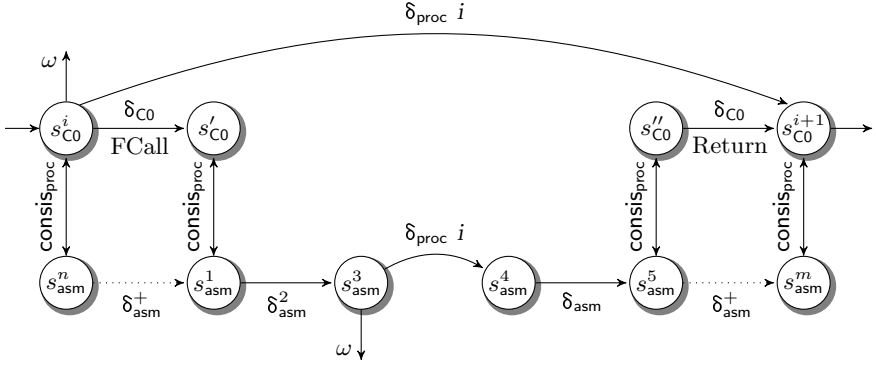


Fig. 3. Verification plan for the C0 implementation of the `olosRecvMsg` primitive

We overload `code_range` taking a C0 process state s_{C0} , which also contains the static C0 program.

Recall that multiple assembly transitions might simulate a single C0 transition. With our notions of successful process execution, we reflect this circumstance by output-input sequences of different length. Nevertheless, both process models should invoke the same primitives, i. e., all output-input pairs except for internal steps (ϵ_Ω , ϵ_Σ), remain equal. For this purpose, we define a normalization function $\gg ois \ll$ over the ois sequences that simply removes all internal steps.

Finally, we extend the compiler theorem to processes:

Theorem 2 (Process Simulation). *As in [Theorem 1](#), we assume well-formed C0 and assembly states s_{C0} and s_{asm} , where the latter is not in a delay slot. Moreover, we assume that the states are related by $\text{consis}_{\text{proc}}$ and there is a successful execution from s_{C0} to a state s'_{C0} .*

Then, it exists a sequence ois' , a function $alloc'$, and a state s'_{asm} such that the normalized sequences are equal, s_{asm} successfully advances under ois' to s'_{asm} , which is not in a delay slot, and s'_{C0} and s'_{asm} are related by $\text{consis}_{\text{proc}}$. Formally:

$$\begin{aligned} & \llbracket \text{is_valid}_{\text{proc}} s_{C0}; \text{is_valid}_{\text{proc}} s_{asm}; s_{asm}.\text{pcp} = s_{asm}.\text{dpc} + 4; \text{consis}_{\text{proc}} s_{C0} \text{ alloc } s_{asm}; \\ & \vdash_{C0}^{\text{proc}} s_{C0} \xrightarrow{ois} s'_{C0} \rrbracket \\ \implies & \exists ois' \text{ alloc}' s'_{asm}. \gg ois \ll = \gg ois' \ll \wedge \text{code_range } s_{C0} \vdash_{asm}^{\text{proc}} s_{asm} \xrightarrow{ois'} s'_{asm} \wedge \\ & s'_{asm}.\text{pcp} = s'_{asm}.\text{dpc} + 4 \wedge \text{consis}_{\text{proc}} s'_{C0} \text{ alloc}' s'_{asm} \end{aligned}$$

Proof. We prove the theorem by induction on the output-input sequence ois . The induction start is trivial. In the induction step, we distinguish the possible process inputs. For an empty input ϵ_Σ , we employ [Theorem 1](#).

For the other inputs, we examine the implementation of the corresponding primitives. [Fig. 3](#) shows the case that primitive `olosRecvMsg` is called in some C0-process state s_{C0}^i . From the induction hypothesis, we know that there exists a corresponding assembly state s_{asm}^n that satisfies the simulation relation $\text{consis}_{\text{proc}}$. Using the sequential C0 semantics, we execute the function-call statement. From

the sequential compiler theorem, we know that the execution (δ_{asm}^+) of the corresponding, compiled code yields an assembly state s_{asm}^1 satisfying the simulation relation $\text{consis}_{\text{proc}}$.

Starting in this state, we execute the inlined assembly code (cf. [Table 2](#)) of the function body⁴. After 2 steps, the code reaches the trap instruction in state s_{asm}^3 . If our implementation is correct, the assembly process signals the same output ω in this state as the C0 process does in s_{C0}^i . At this stage, the transition function δ_{proc} uses an input i from OLOS to proceed to s_{asm}^4 . After a further step, we arrive at the end of the inlined assembly code. From the final assembly state s_{asm}^5 and the C0 state s'_{C0} immediately before the execution of the inlined assembly statement, we construct the corresponding s''_{C0} immediately after the assembly statement⁵. For this to work, we have to show that the assembly code did not disrupt the C0 execution environment (code, stack pointer, etc.). If the assembly code preserves the integrity of the execution environment, we can again establish the $\text{consis}_{\text{proc}}$ relation.

In state s''_{C0} , we employ the sequential C0 semantics to execute the return statement and arrive in the state s_{C0}^{i+1} . From the compiler theorem, we know that there exists a corresponding assembly process such that the simulation relation holds. If the primitive implementation is correct, the state s_{C0}^{i+1} is equal to the state computed from state s_{C0}^i by δ_{proc} with the input i .

The proof for the other primitives proceeds very similarly. \square

4 The Cooperative Concurrent Application Model

The previous section presented a computational model for applications featuring inputs and outputs for the communication with OLOS. For the verification of applications, however, the communication primitives are distracting. Most notably, Isabelle/Simpl cannot deal with inputs and outputs. Hence, we prefer a cooperative concurrent execution model, where we can sequentially reason about a complete computation phase, i. e., between two calls to the EXFINISH primitive. This execution model forms the basis of the application semantics in Simpl.

Recall that during the compute phase, the sole task of OLOS is the exchange of messages with the application that is scheduled to compute in the current slot. Hence, we can perceive the computation of OLOS and the application as a single, sequential program with two separate states: The internal process state, on the one hand, and the OLOS message buffers, on the other hand. While the internal state can be accessed via normal C0 statements, the message buffers are only accessible through the OLOS-communication primitives.

The definition of this new computational model is straightforward: Each state is a pair (s_{proc}, mb) of a process state s_{proc} and a file of message buffers mb . The transition function emulates the behavior of OLOS during the computation

⁴ Recall that a transition δ_{asm} is equal to $\delta_{\text{proc}} \in \Sigma$.

⁵ For reasoning about inlined VAMP assembly, we have been able to reuse previous work [\[15\]](#). Note, however, that the semantics of the TRAP instruction is OLOS-specific.

phase (assuming the corresponding application is scheduled). It distinguishes the output of the process and computes the corresponding input. Formally:

$$\begin{aligned} \delta_{cc} (s_{proc}, mb) \equiv & \\ & \mathbf{case} \ \omega_{proc} \ s_{proc} \ \mathbf{of} \\ & \text{SENDMSG } msgval \ msgnr \Rightarrow \\ & \quad \mathbf{if} \ msgnr < \text{MSGCOUNT} \\ & \quad \mathbf{then} \ (\delta_{proc} \ \text{SENDSUCCESS } s_{proc}, mb[msgnr := msgval]) \\ & \quad \mathbf{else} \ (\delta_{proc} \ \text{INVALIDMSGNR } s_{proc}, mb) \\ & | \text{RCVMSG } msgnr \Rightarrow \\ & \quad \mathbf{if} \ msgnr < \text{MSGCOUNT} \ \mathbf{then} \ (\delta_{proc} \ (\text{RCVSUCCESS } mb[msgnr]) \ s_{proc}, mb) \\ & \quad \mathbf{else} \ (\delta_{proc} \ \text{INVALIDMSGNR } s_{proc}, mb) \\ & | \text{EXFINISH} \Rightarrow (\delta_{proc} \ \text{FINISHSUCCESS } s_{proc}, mb) \\ & | \text{INVPTRERR} \Rightarrow (\delta_{proc} \ \text{INVPTRRESPONSE } s_{proc}, mb) \\ & | \text{REPEATERR} \Rightarrow (s_{proc}, mb) \\ & | \text{CONTINUEERR} \Rightarrow (\delta_{proc} \ \text{CONTINUE } s_{proc}, mb) \\ & | \epsilon_{\Omega} \Rightarrow (\delta_{proc} \ \epsilon_{\Sigma} \ s_{proc}, mb) \end{aligned}$$

Note that this model uses the generic process interface, i. e., s_{proc} might equally refer to a C0 or assembly state. Thus, it is easy to lift process simulation (Theorem 2) to cooperative concurrently executing applications:

Theorem 3 (Cooperative Concurrent Simulation). *Assuming (a) well-formed process states s_{C0} and s_{asm} , where the latter is not in a delay slot, (b) the absence of runtime errors in s_{C0} and all its immediate successors (predicate `runtime_error` does not hold), and (c) a well-formed file of message buffers mb (predicate `is_valid_mb` holds), then a C0 transition in the cooperative concurrent model simulates a number of cooperative concurrent assembly steps. Formally:*

$$\begin{aligned} & \llbracket \text{is_valid}_{proc} \ s_{C0}; \text{is_valid}_{proc} \ s_{asm}; s_{asm}.pcp = s_{asm}.dpc + 4; \text{consis}_{proc} \ s_{C0} \ \text{alloc} \ s_{asm}; \\ & \quad \neg \text{runtime_error} \ s_{C0}; \text{is_valid_mb} \ mb \rrbracket \\ & \Rightarrow \exists n \ \text{alloc}' \ s'_{asm}. \\ & \quad \mathbf{let} \ (s'_{C0}, mb') = \delta_{cc} (s_{C0}, mb) \\ & \quad \mathbf{in} \ (s'_{asm}, mb') = (\delta_{cc}^n) (s_{asm}, mb) \wedge \text{consis}_{proc} \ s'_{C0} \ \text{alloc}' \ s'_{asm} \end{aligned}$$

Proof. At first, we show that the inputs i chosen by function δ_{cc} always match the process output. Hence, the transition from s_{C0} to $\delta_{proc} \ i \ s_{C0}$ is a successful execution. With Theorem 2, we know that there is a corresponding output-input sequence ois' such that

- the sequence ois' contains exactly one pair $(\omega_{proc} \ s_{C0}, i)$ as well as a number of pairs $(\epsilon_{\Omega}, \epsilon_{\Sigma})$, i. e., $\gg ois' \ll = [(\omega_{proc} \ s_{C0}, i)]$, and
- there is a successful assembly execution under ois' starting in s_{asm} and ending in a state s'_{asm} , where `consisproc s'_{C0} alloc' s'_{asm}` holds.

As δ_{cc} reacts with an empty input whenever the process outputs ϵ_{Ω} , function δ_{cc} yields the assembly state s'_{asm} if it is applied $|ois'|$ times to s_{asm} . Furthermore, the message buffers are equal in both cases. \square

Verifying Applications in Isabelle/Simpl. The chief attraction of our cooperative concurrent execution model is that it allows us to reuse Isabelle/Simpl

[14]. The necessary adaptation of the existing technology for application verification is straightforward: It amounts to specify the effects of the primitives for SENDMSG and RECVMMSG in the Simpl language and show that this specification corresponds with the definition of δ_{cc} for these cases. Using the Hoare logic of Isabelle/Simpl, we can conveniently establish the absence of runtime errors as well as efficiently reason about functional correctness of applications between two calls to the EXFINISH primitive.

5 Embedding Applications into the Overall ECU Model

In the previous section, we have claimed that the transition function δ_{cc} of the cooperative concurrent application model emulates the OLOS transitions during the computation phase iff the corresponding application is scheduled. In this section, we prove that claim. Formally, we express our claim with the help of a projection function Π_p , which extracts the state of the application p , and an injection $\text{inj}_p s_{cc} s_{ECU}$, which updates the state of application p in the ECU state s_{ECU} by s_{cc} . We state:

Theorem 4 (Application Embedding). *We assume that (a) the application p is computing in the ECU state s according to the static schedule t and (b) that p does not call for EXFINISH. Then, the projection of p followed by a transition of δ_{cc} and its injection into s is equal to a transition of δ_{ECU} . Formally:*

$$\llbracket \text{is_computing } p \ t \ s; \neg \text{calls_finish } p \ s \rrbracket \implies \text{inj}_p (\delta_{cc} (\Pi_p \ s)) \ s = \delta_{ECU} \ t \ \perp \ s$$

Proof. We have proven this fact in Isabelle/HOL. □

Note that OLOS leaves the computing state (cf. Sect. 2.2) when an application calls for EXFINISH, i. e., predicate $\text{calls_finish } pid \ s$ holds. In this case, the transition $\delta_{ECU} \ tables \ \perp \ s$ only raises the idle flag and sends FINISHSUCCESS to process pid . From the OLOS specification, we know that at all other ECU states, the process states remain constant.

6 Reasoning about Applications – A Practical Example

So far, we have elaborated on the foundation of our verification approach. Now, we take our arguments a step further and venture a practical example. For this purpose, we use a simple application program for cruise control. Note that all OLOS applications share a common control flow, which is depicted in Fig. 4: After an application-specific set-up, they implement an infinite loop. The loop body contains a function call to a function `compute`, which implements the actual functionality of the application, and a call to the EXFINISH primitive.

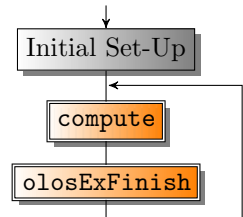


Fig. 4. General control-flow of applications

```

int compute() {
  unsigned int command; unsigned int current_speed;

  dummy = olosRecvMsg(buffer, 0u); command = buffer->Field; // read command
  dummy = olosRecvMsg(buffer, 1u); current_speed = buffer->Field; // read current speed

  // target_speed adjustment
  if (enabled) {
    if (command == CC_INCREASE) {
      if (target_speed < MAX_SPEED - 1u) { target_speed = target_speed + 2u; }
    }
    else if (command == CC_DECREASE) {
      if (target_speed > MIN_SPEED + 1u) { target_speed = target_speed - 2u; }
    }
    else if (command != CC_SET) { enabled = false; }
  }
  else if (current_speed >= MIN_SPEED && (command == CC_SET ||
    command == CC_INCREASE || command == CC_DECREASE)) {
    enabled = true;
    if (current_speed >= MAX_SPEED) { target_speed = MAX_SPEED; }
    else { target_speed = current_speed; }
  }

  // speed regulation
  *buffer = INIT_BUFFER;
  if (!enabled) { dummy = olosSendMsg(buffer, 2u); dummy = olosSendMsg(buffer, 3u); }
  else if (current_speed > target_speed) {
    dummy = olosSendMsg(buffer, 2u);
    buffer->Field = current_speed - target_speed; dummy = olosSendMsg(buffer, 3u);
  }
  else {
    dummy = olosSendMsg(buffer, 3u);
    buffer->Field = target_speed - current_speed; dummy = olosSendMsg(buffer, 2u);
  }

  return 0;
}

```

Fig. 5. Function `compute` of our simple cruise-control application

Our example program features a global variable `target_speed` storing the speed that the regulator is aiming for. Furthermore, there is a global Boolean variable `enabled` that is true iff the speed control is enabled.

The `compute` function (see [Fig. 5](#)) reads the message buffer 0 to receive one of the commands *ON*, *OFF*, *INCREASE*, and *DECREASE* as well as the message buffer 1 to receive the current speed. The function adjusts the global variables wrt. the received command and subtracts the current from the target speed. If the difference is positive, the function sends the difference to message buffer 2 (which we assume to be read by the accelerator unit) and value 0 to message buffer 3 (which we assume to be read by the brake unit). If the difference is negative, the function sends the absolute value to buffer 3 and value 0 to buffer 2. Afterwards, `compute` returns and the program calls `EXFINISH`.

For the verification of the `compute` function, we employ the existing technology for sequential reasoning: At first, we mechanically translate the C0 code into Simpl. Then, we formally specify the functionality in terms of Hoare triples and, conveniently relying on the Hoare logics, prove the correctness of our specification. Obviously, the containing loop alters neither the application's variables nor

the message buffers. Thus, our proven property holds for a complete computation phase (i. e., the loop body). Finally, we know from the property transfer theorem of Isabelle/Simpl that there is an equivalent property over the C0 semantics.

Using [Theorem 3](#), we can then infer that the property can be translated down to assembly level. Furthermore, [Theorem 4](#) allows us to infer properties about the whole ECU behavior by combining verification results from the applications running on the ECU. Recall that our example application relied on several assumptions: The message buffers 0 and 1 are assumed to stem from sensors, and the buffers 2 and 3 should be sent to other control units. Consequently, the static schedule should provide slots, where the messages received from the bus are stored into the buffers 0 and 1; moreover, the buffers 2 and 3 should be sent onto the bus. Furthermore, the applications sharing the same ECU as our example application, should not alter the buffers after receiving or before sending, respectively. In addition, we have assumed that the example application calls EXFINISH before the slot end. Within the Hoare logic, we can prove that our compute function terminates. Thus, the only remaining issue is to find an upper bound for the worst-case execution time, which is easily done by static analysis [\[16\]](#). Eventually, we can then argue that the values computed by our example application are indeed sent onto the bus several slots later.

7 Conclusion

Based on existing technology for ordinary program verification, we formally proved the foundation of a pervasive verification approach for applications communicating with the operating system OLOS. Additionally, we provided an application example illustrating that our approach can indeed be used in practice.

With our work, we respond to a long lasting grand challenge [\[3\]](#). Despite many recent achievements in operating-systems verification [\[4\]](#), we only know of a single project that attempted pervasive systems verification: Bevier *et. al* [\[17\]](#) verified the correctness of KIT, a small assembly program that provides task isolation, device I/O, and single word message passing. Moreover, they ventured into the verification of applications but could not fully integrate their results. We can only refer to their work as groundbreaking because of KIT's simplicity.

Though even our computer system is simple, it is practically usable and the developed verification technique as well as the overall proof architecture may be reused for real computer systems. We implemented OLOS as well as our example application in a C variant and employed a verified compiler for the mechanic translation into executable code. Thus, we are able to verify programs on the source code level—conveniently in a Hoare logic using the verification environment Isabelle/Simpl—and can then transfer the proven properties down to the assembly level (or even further [\[9\]](#)), e. g., to combine it with properties of the operating system or peripheral devices.

Integrating different layers of abstraction into a coherent theory is an important prerequisite for efficient reasoning. The verification engineer can then choose a convenient abstraction layer for reasoning although the results might

eventually be needed at a different abstraction layer. We see our contribution as an important milestone towards an evidence-based validation of safety-critical computer systems. Pervasive verification and software engineering should become two complementing disciplines aiming at the same target: perfectly reliable software.

Acknowledgments. We thank the anonymous peer reviewers for their detailed review reports with a very constructive criticism and many helpful suggestions.

References

1. Vartabedian, R., Bensinger, K.: Doubt cast on Toyota’s decision to blame sudden acceleration on gas pedal defect, Los Angeles Times (January 30, 2010)
2. Gwynn, J.: Apple co-founder Steve Wozniak says his Toyota Prius accelerates on its own, Los Angeles Times (February 3, 2010)
3. Moore, J.S.: A grand challenge proposal for formal methods: A verified stack. In: 10th Anniversary Colloquium of UNU/IIST, pp. 161–172. Springer, Heidelberg (2002)
4. Klein, G.: Operating system verification — an overview. *Sādhanā* 34(1), 27–69 (2009)
5. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. *STTT* 8(4-5), 411–430 (2006)
6. In der Rieden, T., Tsyban, A.: CVM – a verified framework for microkernel programmers. In: Huuck, R., Klein, G., Schlich, B. (eds.) *Systems Software Verification*. ENTCS, vol. 217, pp. 151–168. Elsevier Science B.V., Amsterdam (2008)
7. Daum, M., Schirmer, N.W., Schmidt, M.: Implementation correctness of a real-time operating system. In: van Hung, D., Krishnan, P. (eds.) *SEFM*, pp. 23–32. IEEE Computer Society, Los Alamitos (2009)
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. *LNCS*, vol. 2283. Springer, Heidelberg (2002)
9. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load – leveraging a semantics stack for systems verification. *J. Autom. Reasoning* 42(2-4), 389–454 (2009)
10. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Huuck, R., Klein, G., Schlich, B. (eds.) *Systems Software Verification*. ENTCS, vol. 217, pp. 23–40. Elsevier Science B.V., Amsterdam (2008)
11. Schirmer, N.W.: *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU Munich (2006)
12. Kopetz, H., Grünsteidl, G.: TTP – A protocol for fault-tolerant real-time systems. *IEEE Computer* 27(1), 14–23 (1994)
13. American National Standards Institute: ANSI ISO IEC 9899-1999: Programming Languages — C. American National Standards Institute, New York, USA (1999)
14. Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N., Starostin, A.: The Verisoft approach to systems verification. In: Shankar, N., Woodcock, J. (eds.) *VSTTE 2008*. *LNCS*, vol. 5295, pp. 209–224. Springer, Heidelberg (2008)
15. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Huuck, R., Klein, G., Schlich, B. (eds.) *Systems Software Verification*. ENTCS, vol. 217, pp. 169–185. Elsevier Science B.V., Amsterdam (2008)
16. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. White paper, AbsInt Angewandte Informatik GmbH (2004)
17. Bevier, W.R.: Kit and the short stack. *J. Autom. Reasoning* 5(4), 519–530 (1989)

A Compositional Method for Deciding Equivalence and Termination of Nondeterministic Programs

Aleksandar Dimovski

Faculty of Information-Communication Tech., FON University, Skopje, 1000, MKD

Abstract. In this paper we address the problem of deciding may- and must-equivalence and termination of nondeterministic finite programs from second-order recursion-free Erratic Idealized Algol. We use game semantics to compositionally extract finite models of programs, and the CSP process algebra as a concrete formalism for representation of models and their efficient verification. Observational may- and must-equivalence and liveness properties, such as divergence and termination, are decided by checking traces refinements and divergence-freedom of CSP processes using the FDR tool. The practicality of the approach is evaluated on several examples.

1 Introduction

Game semantics is a syntax-independent approach of modeling open programs by looking at the ways in which a program can observably interact with its environment (context). Types are modeled by games (or arenas) between a Player (i.e. program) and an Opponent (i.e. environment), and programs are modeled by strategies on games. It was shown that, for several interesting programming language fragments, their game semantics yield algorithms for model checking. The focus has been on Idealized Algol (IA) [11,12], which represents a metalanguage combining imperative with higher-order functional features. Game semantics is compositional, i.e. defined recursively on the syntax, which is essential for the modular analysis of larger programs. Previous work on model checking using game semantic models has been mainly concerned with verification of safety properties of sequential, concurrent, and probabilistic programs [9,10,13]. All these models say about what a program *may* do, but nothing about what it *must* do. This reflects the deficiencies of the models for reasoning about anything other than safety properties.

In order to take account of liveness properties of nondeterministic programs, the requested model must make distinctions between a reliable program, such as `skip`, and an unreliable program, such as `skip or divergecom`. This means that the model must capture two complementary notions of program equivalence: the possibility of termination (may-termination) and the guarantee of termination (must-termination). To address this issue, the strategy of a program, apart from containing the potential convergent behaviours of the program, must be enriched

with an extra information about the possible divergent behaviours of the program. In [11,12], it is given a game semantic model for Erratic IA (EIA) which is fully abstract with respect to may & must-termination equivalence. EIA represents a nondeterministic extension of IA, i.e. it is IA enriched with an erratic choice ‘or’ operator. The full abstraction result means that the model validates all and only correct (may & must-termination) equivalences between programs, i.e. it is sound and complete. Although this model is appropriate for verifying both, safety and liveness, properties, it is complicated and so equivalence and a range of properties are not decidable within it. However, it has been shown in [14] that the game models of second- (resp., third-) order recursion-free finitary EIA can be represented by finite (resp., visibly pushdown) automata. This gives a decision procedure for a range of verification problems to be solved algorithmically, such as: may-equivalence, must-equivalence, may & must-equivalence, termination and other properties.

In this work we propose a verification tool for analyzing nondeterministic programs. We show how for second-order recursion-free EIA with finite data types game semantic models can be represented as CSP processes, i.e. any program is compositionally modeled as a CSP process whose terminated traces and minimal divergences are exactly all the complete plays and divergences of the strategy for the program. This enables observational may- and must-equivalence between any two programs and a range of (safety and) liveness properties, such as termination and divergence, of programs to be decided by checking traces refinements and divergence-freedom of CSP processes by using the FDR tool.

CSP [15] is a particularly convenient formalism for encoding game semantic models. The FDR model checker can be used to automatically check refinements between two processes and a variety of properties of a process, and to debug interactively when a refinement or a property does not hold. FDR has direct support for three different forms of refinement: traces (\sqsubseteq_T), failures (\sqsubseteq_F), and failures-divergences (\sqsubseteq_{FD}); and for the following properties: deadlock, divergence, and determinism. Then, composition of strategies, which is used in game semantics to obtain the strategy for a program from strategies for its subprograms, is represented in CSP by renaming, parallel composition and hiding operators, and FDR is highly optimised for verification of such networks of processes. Finally, FDR builds the models gradually, at each stage compressing the submodels to produce an equivalent process with many fewer states. A number of hierarchical compression algorithms are available in FDR, which can be applied during either model generation or refinement checking.

The paper is organised in the following way. Section 2 introduces the language considered in this paper. The game semantic model of the language is defined in Section 3, and its CSP representation is presented in Section 4. Correctness of the CSP representation, decidability of observational may- and must-equivalence and termination are shown in Section 5. The effectiveness of this approach is evaluated in Section 6. In the end, we conclude and present some ideas for future work.

Related work. Automated verification of liveness properties of programs is an active research topic. The work in [5] presents an abstraction refinement procedure for proving termination of programs. The procedure successively weakens candidate transition invariants and successively refines transition predicate abstractions of the given program. The approach taken in [3,4] proves termination of programs by generating linear ranking functions, which assign a value from a well-founded domain to each program state. The method represents program invariants and transition relations as polyhedral cones and constructs linear ranking functions by manipulating these cones. Compared to the aforementioned approaches, the main focus of our method is compositionality which is reached in a clean and theoretically firm semantics-based way. Namely, the model of a program is constructed out of the models of its subprograms, which facilitates breaking down the verification of a larger program into verifications of its subprograms. By applying various program analysis techniques, such as predicate abstraction and counter-example guided abstraction refinement [2,7], the efficiency of our method and its applicability to a broader class of programs can be significantly improved.

2 Programming Language

Erractic Idealized Algol [12] is a nondeterministic imperative-functional language which combines the fundamental imperative features, locally-scoped variables, and full higher-order function mechanism based on a typed call-by-name λ -calculus.

The data types D are a finite subset of the integers (from 0 to $n - 1$, where $n > 0$) and the Booleans ($D ::= \text{int}_n \mid \text{bool}$). The phrase types consists of base types (expressions, commands, variables) and function types ($B ::= \text{exp}D \mid \text{var}D \mid \text{com}, T ::= B \mid T \rightarrow T$). Terms are formed by the following grammar:

$$M ::= x \mid v \mid \text{skip} \mid \text{diverge}_B \mid M \text{ op } M \mid M; M \mid \text{if } M \text{ then } M \text{ else } M \mid \text{while } M \text{ do } M \\ \mid M := M \mid !M \mid \text{newvar}D \ x := v \text{ in } M \mid \text{mkvar}MM \mid M \text{ or } M \mid \lambda x.M \mid MM \mid YM$$

where v ranges over constants of type D . The language constants are a “do nothing” command `skip` which always terminates successfully, and for each base type there is a constant `divergeB` which causes a program to enter an unresponsive state similar to that caused by an infinite loop. The usual imperative constructs are employed: sequential composition (`;`), conditional (`if`), iteration (`while`), assignment (`:=`), and de-referencing (`!`). Block-allocated local variables are introduced by a *new* construct, which initializes a variable and makes it local to a given block. There are constructs for nondeterminism, function creation and application, as well as recursion.

Well-typed terms are given by typing judgements of the form $\Gamma \vdash M : T$, where Γ is a type *context* consisting of a finite number of typed free identifiers. Typing rules of the language are those of EIA (e.g. [11,12]), extended with a rule for the `divergeB` constant: $\Gamma \vdash \text{diverge}_B : B$.

The operational semantics of our language is given in terms of *states*. Given a type context $\Gamma = x_1 : \text{var}D_1, \dots, x_k : \text{var}D_k$ where all identifiers are variables, which is called *var-context*, we define a Γ -state s as a (partial) function assigning data values to the variables $\{x_1, \dots, x_k\}$. The canonical forms are defined by $V ::= x \mid v \mid \lambda x : T.M \mid \text{skip} \mid \text{mkvar}MN$. The operational semantics is defined by a big-step reduction relation:

$$\Gamma \vdash M, s \Longrightarrow V, s'$$

where $\Gamma \vdash M : T$ is a term, Γ is a *var-context*, s, s' are Γ -states, and V is a canonical form. Reduction rules are those of EIA (see [112] for details). The diverge_B constant is not reducible.

Since the language is nondeterministic, it is possible that a term may reduce to more than one value. Given a term $\Gamma \vdash M : \text{com}$ where Γ is a *var-context*, we say that M *may terminate* in state s , written $M, s \Downarrow^{\text{may}}$, if there exists a reduction $\Gamma \vdash M, s \Longrightarrow \text{skip}, s'$ for some state s' . We say that M *must terminate* in a state s , written $M, s \Downarrow^{\text{must}}$, if all reductions at start state s end with the term *skip*. If M is a closed term then we abbreviate the relation $M, \emptyset \Downarrow^{\text{may}}$ (resp., $M, \emptyset \Downarrow^{\text{must}}$) with $M \Downarrow^{\text{may}}$ (resp., $M \Downarrow^{\text{must}}$). Next, we define a *program context* $C[-] : \text{com}$ with *hole* to be a term with (possibly several occurrences of) a hole in it, such that if $\Gamma \vdash M : T$ is a term of the same type as the hole then $C[M]$ is a well-typed closed term of type *com*, i.e. $\vdash C[M] : \text{com}$. Then, we say that a term $\Gamma \vdash M : T$ is a *may-approximate* (resp., *must-approximate*) of a term $\Gamma \vdash N : T$, denoted by $\Gamma \vdash M \sqsubseteq_{\text{may}} N$ (resp., $\Gamma \vdash M \sqsubseteq_{\text{must}} N$), if and only if for all program contexts $C[-] : \text{com}$, if $C[M] \Downarrow^{\text{may}}$ (resp., $C[M] \Downarrow^{\text{must}}$) then $C[N] \Downarrow^{\text{may}}$ (resp., $C[N] \Downarrow^{\text{must}}$). If two terms may-approximate (resp., must-approximate) each other they are considered *may-equivalent* (resp., *must-equivalent*), denoted by $\Gamma \vdash M \cong_{\text{may}} N$ (resp., $\Gamma \vdash M \cong_{\text{must}} N$). Combining may- and must-approximation (resp., equivalence) gives rise to *may&must approximation* (resp., *equivalence*). For instance, the following facts hold:

$$\begin{array}{ll} \text{skip or } \text{diverge}_{\text{com}} \cong_{\text{may}} \text{skip} & \text{skip or } \text{diverge}_{\text{com}} \cong_{\text{must}} \text{diverge}_{\text{com}} \\ \text{skip or } \text{diverge}_{\text{com}} \not\cong_{\text{may\&must}} \text{skip} & \text{skip or } \text{diverge}_{\text{com}} \not\cong_{\text{may\&must}} \text{diverge}_{\text{com}} \end{array}$$

Example 1. Consider the term from [14]:

$$f : \text{com} \rightarrow \text{com} \vdash \text{newint}_2 x := 0 \text{ in } \left(f(x := 1) ; \text{if } (x = 1) \text{ then } \text{diverge}_{\text{com}} \right) \\ \text{or } \left(f(x := 1) ; \text{if } (x = 0) \text{ then } \text{diverge}_{\text{com}} \right) : \text{com}$$

in which f is a non-local function. The function-call mechanism is by-name, so every call to the argument of f sets x to 1. In what follows, we will see that this term is may-equivalent to $f(\text{skip})$, must-equivalent to $\text{diverge}_{\text{com}}$, and may&must-equivalent to $f(\text{skip})$ or $\text{diverge}_{\text{com}}$. \square

3 Game Semantics

In this section we give an overview of game semantics for EIA, which is fully abstract with respect to may&must-equivalence. A complete definition can be found in [12, pp. 99–138].

An *arena* A is a triple $\langle M_A, \lambda_A, \vdash_A \rangle$, where M_A is a countable set of *moves*, $\lambda_A : M_A \rightarrow \{O, P\} \times \{Q, A\}$ is a labeling function which indicates whether a move is by *Opponent* (O) or *Player* (P), and whether it is a *question* (Q) or an *answer* (A). Then, \vdash_A is a binary relation between $M_A + \{*\}$ ($* \notin M_A$) and M_A , called *enabling* (if $m \vdash_A n$ we say that m enables move n), which satisfies the following conditions: (i) Initial moves (a move enabled by $*$ is called *initial*) are Opponent questions, and they are not enabled by any other moves besides $*$; (ii) Answer moves can only be enabled by question moves; (iii) Two participants always enable each others moves, never their own.

We denote the set of all initial moves in A as I_A . The simplest arena is the empty arena $I = \langle \emptyset, \emptyset, \emptyset \rangle$. The base types are interpreted by arenas where all questions are initial and P-moves answer them.

$$\begin{aligned} \llbracket \text{expD} \rrbracket &= \langle \{q, v \mid v \in D\}, \{\lambda(q) = \text{OQ}, \lambda(v) = \text{PA}\}, \{(*, q), (q, v) \mid v \in D\} \rangle \\ \llbracket \text{com} \rrbracket &= \langle \{run, done\}, \{\lambda(run) = \text{OQ}, \lambda(done) = \text{PA}\}, \{(*, run), (run, done)\} \rangle \\ \llbracket \text{varD} \rrbracket &= \langle \{read, v, write(v), ok \mid v \in D\}, \{\lambda(read, write(v)) = \text{OQ}, \\ &\quad \lambda(v, ok) = \text{PA}\}, \{(*, read), (*, write(v)), (read, v), (write(v), ok) \mid v \in D\} \rangle \end{aligned}$$

Given arenas A and B , we define new arenas $A \times B$, $A \Rightarrow B$ as follows¹:

$$\begin{aligned} A \times B &= \langle M_A + M_B, [\lambda_A, \lambda_B], \vdash_A + \vdash_B \rangle \\ A \Rightarrow B &= \langle M_A + M_B, [\bar{\lambda}_A, \lambda_B], \vdash_B + (I_B \times I_A) + (\vdash_A \cap (M_A \times M_A)) \rangle \end{aligned}$$

A *justified sequence* s in arena A is a finite sequence of moves of A together with a pointer from each non-initial move n to an earlier move m such that $m \vdash_A n$. We say that n is (explicitly) justified by m . A *legal play* (or just a play) is a justified sequence with some additional constraints: *alternation* (Opponent and Player moves strictly alternate), *well-bracketed* condition (when an answer is given, it is always to the most recent question which has not been answered), and *visibility* condition (a move to be played depends upon a certain subsequence of the play so far, rather than on all of it). The set of all legal plays in arena A is denoted by L_A . We use meta-variables m, n to range over moves, and s to range over sequences of moves. We also write $s m$ or $s \cdot m$ for the concatenation of s and m . The empty sequence is written as ϵ , and \sqsubseteq denotes the prefix ordering on sequences.

A *strategy* σ on an arena A (written as $\sigma : A$) is a pair (T_σ, D_σ) , where:

- T_σ is a non-empty set of even-length plays of A , known as the *traces* of σ , satisfying: if $s \cdot m \cdot n \in T_\sigma$ then $s \in T_\sigma$.
- D_σ is a set of odd-length plays of A , known as the *divergences* of σ , satisfying: if $s \cdot m \in D_\sigma$ then $s \in T_\sigma$; and if $s \in T_\sigma$, $s \cdot m \in L_A$, and $s \cdot m \cdot n \notin T_\sigma$ for $\forall n$ then $\exists d \in D_\sigma. d \sqsubseteq s \cdot m$.

A strategy specifies what options Player has at any given point of a play and it does not restrict the Opponent moves. If Player can not respond at some point of

¹ $\bar{\lambda}_A$ is like λ_A except that it reverses O/P part, and $+$ is a disjoint union.

a play then this is reflected by an appropriate divergence sequence in the strategy. Note that, here we choose a “minimal” representation of divergence, where only the minimal divergences of a strategy, denoted by $div(\sigma)$, are recorded. An alternative “maximal” representation, as is done in CSP, is also possible. It considers the divergences as extension-closed set, which forces the traces set to include all possible sequences after a divergence has been reached. These two representations are equivalent.

Given strategies $\sigma : A \Rightarrow B$ and $\tau : B \Rightarrow C$, the composition $\sigma \circledast \tau = (T_{\sigma \circledast \tau}, D_{\sigma \circledast \tau}) : A \Rightarrow C$ is defined in the following way. Let u be a sequence of moves from A , B , and C . We define $u \upharpoonright B, C$ to be the subsequence of u consisting of all moves from B and C as well as pointers between them (pointers from/to moves of A are deleted). Similarly define $u \upharpoonright A, B$. Define $u \upharpoonright A, C$ to be the subsequence of u consisting of all moves from A and C , but where there was a pointer from a move $m_A \in M_A$ to an initial move $m \in I_B$ extend the pointer to the initial move in C which was pointed to from m . We say that u is an *interaction* of A, B, C if $u \upharpoonright A, B \in L_{A \Rightarrow B}$, $u \upharpoonright B, C \in L_{B \Rightarrow C}$, and $u \upharpoonright A, C \in L_{A \Rightarrow C}$. The set of all such sequences is written as $int(A, B, C)$.

$$T_{\sigma \circledast \tau} = \{u \upharpoonright A, C \mid u \in int(A, B, C) \wedge u \upharpoonright A, B \in T_\sigma \wedge u \upharpoonright B, C \in T_\tau\}$$

So $T_{\sigma \circledast \tau}$ consists of sequences generated by playing T_σ and T_τ in parallel, making them synchronize on moves in B , which are afterwards hidden.

We define an *infinite interaction* of A, B, C to be a sequence $u \in (M_A + M_B + M_C)^\infty$ such that $u \upharpoonright A, C \in L_{A \Rightarrow C}$, and for all $i \in \mathbb{N}_0$, $u_{<i} \upharpoonright A, B \in L_{A \Rightarrow B}$ and $u_{<i} \upharpoonright B, C \in L_{B \Rightarrow C}$, where $u_{<i}$ denotes the finite prefix of u with length i . The set of all such sequences is written as $int_\infty(A, B, C)$. Then, a set of *finitely generated* divergences is defined as:

$$D_\sigma \not\sim D_\tau = \{u \in int(A, B, C) \mid (u \upharpoonright A, B \in T_\sigma \wedge u \upharpoonright B, C \in D_\tau) \vee (u \upharpoonright A, B \in D_\sigma \wedge u \upharpoonright B, C \in T_\tau)\}$$

$D_\sigma \not\sim D_\tau$ consists of sequences containing a trace from σ and a divergence from τ , or vice versa. A set of *infinitely generated* divergences is defined as [2](#):

$$T_\sigma \not\sim T_\tau = \{u \in int_\infty(A, B, C) \mid \forall i \in \mathbb{N}_0. u_{\leq i} \upharpoonright A, B \in T_\sigma \cup dom(\sigma) \wedge u_{\leq i} \upharpoonright B, C \in T_\tau \cup dom(\tau)\}$$

$T_\sigma \not\sim T_\tau$ consists of sequences that have an infinite tail in B . This situation is called *livelock*. We have that: $D_{\sigma \circledast \tau} = \{u \upharpoonright A, C \mid u \in D_\sigma \not\sim D_\tau \vee u \in T_\sigma \not\sim T_\tau\}$.

The *identity strategy*, which is also called *copy-cat*, for an arena A is $Id_A = (id_A, \emptyset)$, where $id_A = \{s \in P_{A \Rightarrow A} \mid \forall s' \sqsubseteq^{even} s. s' \upharpoonright A_l = s' \upharpoonright A_r\}$. We use the l and r tags to distinguish between the two occurrences of A and $s' \sqsubseteq^{even} s$ means that s' is an even-length prefix of s . So in id_A , a move by Opponent in either occurrence of A is immediately copied by Player to the other occurrence.

In general, plays in a strategy may contain several occurrences of initial moves, which define several different *threads* of the play in the following way: two moves

² The domain of a strategy σ is the set $dom(\sigma) = \{s \cdot m \mid \exists n. s \cdot m \cdot n \in T_\sigma\}$.

are in the same thread if they are connected via chains of pointers to the same occurrence of an initial move. We consider the class of *single-threaded* strategies whose behaviour depends only on one thread at a time, i.e. any Player response depends solely on the current thread of the play and any divergence is caused by the play in a single thread. We say that a strategy is *well-opened* if all its plays have exactly one initial move. It can be established one-to-one correspondence between single-threaded and well-opened strategies. It is shown in [12] that arenas as objects and single-threaded (well-opened) strategies as arrows constitute a cpo-enriched cartesian closed category, which can be used for construction of semantic models of programming languages. From now on, we proceed to work only with well-opened strategies and plays with exactly one initial move.

A term $\Gamma \vdash M : T$, where $\Gamma = x_1 : T_1, \dots, x_n : T_n$, is interpreted by a strategy $\llbracket \Gamma \vdash M : T \rrbracket$ for the arena $\llbracket \Gamma \vdash T \rrbracket = \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \Rightarrow \llbracket T \rrbracket$. Language constants and constructs are interpreted by strategies and compound terms are modelled by composition of the strategies that interpret their constituents. For example, some of the strategies are [12]: $\llbracket n : \text{expint} \rrbracket = (\{\epsilon, q n\}, \emptyset)$, $\llbracket \text{skip} : \text{com} \rrbracket = (\{\epsilon, \text{run done}\}, \emptyset)$, $\llbracket \text{diverge}_{\text{com}} : \text{com} \rrbracket = (\{\epsilon\}, \{\text{run}\})$, $\llbracket \text{or} : \text{exp}D_0 \times \text{exp}D_1 \rightarrow \text{exp}D_2 \rrbracket^3 = (\{\epsilon, q_2 q_0, q_2 q_1, q_2 q_0 v_0 v_2, q_2 q_1 v_1 v_2 \mid v \in D\}, \emptyset)$, free identifiers are interpreted by identity strategies, etc. Using standard game-semantic techniques, it has been shown in [12] that the quotient of this model with respect to the so-called intrinsic preorder is fully abstract for may&must-equivalence. However, the model itself is sound [12], so the must-termination of terms follows from this result.

Proposition 1. $\Gamma \vdash M$ must terminate iff $D_{\llbracket \Gamma \vdash M \rrbracket} = \emptyset$.

More explicit characterizations of may- and must-approximation (resp., equivalence) are given in [14]. A play is *complete* if all questions occurring in it have been answered. Given a strategy σ , we write $\text{comp}(\sigma)$ for the set of its non-empty complete plays.

Proposition 2. $\Gamma \vdash M \sqsubseteq_{\text{may}} N$ iff $\text{comp}(\llbracket \Gamma \vdash M \rrbracket) \subseteq \text{comp}(\llbracket \Gamma \vdash N \rrbracket)$.

In order to capture must-approximation, we define a new relation \leq_{must} on strategies over any arena A : $\sigma \leq_{\text{must}} \tau$ iff for any $s \in (T_\tau \cup D_\tau) \setminus (T_\sigma \cup D_\sigma)$ there exists $s' \sqsubseteq^{\text{odd}} s$ such that $s' \sqsubseteq d$ for some $d \in D_\sigma$.

Proposition 3. $\Gamma \vdash M \sqsubseteq_{\text{must}} N$ iff $\llbracket \Gamma \vdash M \rrbracket \leq_{\text{must}} \llbracket \Gamma \vdash N \rrbracket$.

4 CSP Representation

In the rest of the paper, we work with the 2nd-order recursion-free fragment of EIA. In particular, function types are restricted to $T ::= B \mid B \rightarrow T$. Without loss of generality, we consider only terms in β -normal form. We now show how the game semantic model of this fragment of EIA can be given a concrete representation using the CSP process algebra. This translation is an extension of the

³ Every move is tagged with the index of type component where it occurs.

one presented in [6,9], where the considered language is IA and the model takes account of only safety properties.

CSP (Communicating Sequential Processes) [15] is a language for modelling systems which consist of interacting components. Each component is specified through its behaviour which is given as a *process*. Processes are defined in terms of the *events* that they can perform. The set of all possible events is denoted Σ .

Processes can be given denotational semantics by the following sets of their possible behaviours. The set $\text{traces}(P)$ contains all possible finite sequences of events that the process P can perform. The set $\text{failures}(P)$ consists of all pairs (s, X) , where $s \in \text{traces}(P)$ and X is a set of events that P can refuse to do in some stable state after the trace s . And, we define $\text{divergences}(P)$ as the set of traces after which the process can perform an infinite sequence of consecutive internal events called τ . We consider here two semantic models of CSP processes: *traces semantics*, denoted as P_T , and *failures-divergences semantics*, denoted as P_{FD} . We omit the subscripts when they are clear from the context. Traces semantics of a process P is given by the set $\text{traces}(P)$, while failures-divergences semantics of P is given by the pair $(\text{failures}(P), \text{divergences}(P))$. Divergences of a process are not modeled in its traces semantics, but they have “maximal” representation in its failures-divergences semantics. For example, let consider the *div* process. It represents a special divergent process in CSP which does nothing but diverge. It is equivalent to the recursive process $\mu p.p$. We have that $\text{div}_T = \{\epsilon\}$, but $\text{div}_{FD} = (\Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^\checkmark), \Sigma^{*\checkmark})$, where $\Sigma^\checkmark = \Sigma \cup \{\checkmark\}$ ⁴, and $\Sigma^{*\checkmark} = \Sigma \cup \{s \cdot \checkmark \mid s \in \Sigma^*\}$. *Traces refinement* between processes is defined as:

$$P_1 \sqsubseteq_T P_2 \Leftrightarrow \text{traces}(P_2) \subseteq \text{traces}(P_1)$$

CSP processes can also be given operational semantics using *labelled transition systems* (LTS). The LTS of a process is a directed graph whose nodes represent process states and whose edges are labelled by events representing what happens when the given event is performed. LTSs have a distinguished start state, and any edge whose label is \checkmark leads to a special terminated state Ω .

With each type T , we associate a set of possible events: an alphabet $\mathcal{A}_{[T]}$. It contains a set of events $\mathbf{q} \in \mathcal{Q}_{[T]}$, called questions, which are appended to a channel with name Q , and for each question \mathbf{q} , there is a set of events $\mathbf{a} \in \mathcal{A}_{[T]}^{\mathbf{q}}$, called answers, which are appended to a channel with name A .

$$\begin{aligned} \mathcal{A}_{[\text{int}_n]} &= \{0, \dots, n-1\} & \mathcal{A}_{[\text{bool}]} &= \{tt, ff\} \\ \mathcal{Q}_{[\text{exp}D]} &= \{q\} & \mathcal{A}_{[\text{exp}D]}^q &= \mathcal{A}_{[D]} & \mathcal{Q}_{[\text{com}]} &= \{run\} & \mathcal{A}_{[\text{com}]}^{run} &= \{done\} \\ \mathcal{Q}_{[\text{var}D]} &= \{read, write.v \mid v \in \mathcal{A}_{[D]}\} & \mathcal{A}_{[\text{var}D]}^{read} &= \mathcal{A}_{[D]} & \mathcal{A}_{[\text{var}D]}^{write.v} &= \{ok\} \\ \mathcal{Q}_{[B_1 \rightarrow \dots \rightarrow B_k \rightarrow B]} &= \bigcup_{1 \leq i \leq k} \{i.q \mid \mathbf{q} \in \mathcal{Q}_{[B_i]}\} \cup \mathcal{Q}_{[B]} \\ \mathcal{A}_{[B_1 \rightarrow \dots \rightarrow B_k \rightarrow B]}^{i.q} &= \{i.a \mid \mathbf{a} \in \mathcal{A}_{[B_i]}^{\mathbf{q}}\}, \mathbf{q} \in \mathcal{Q}_{[B_i]}, 1 \leq i \leq k \\ \mathcal{A}_{[B_1 \rightarrow \dots \rightarrow B_k \rightarrow B]}^q &= \mathcal{A}_{[B]}^q, \mathbf{q} \in \mathcal{Q}_{[B]} \end{aligned}$$

⁴ *SKIP* is a process that successfully terminates causing the special event \checkmark ($\checkmark \notin \Sigma$).

$$\mathcal{A}_{[T]} = Q.Q_{[T]} \cup A. \bigcup_{q \in Q_{[T]}} A_{[T]}^q$$

For any term $\Gamma \vdash M : T$, we define a CSP process $\llbracket \Gamma \vdash M : T \rrbracket$ which represents the strategy for the term. Events of this process are from the alphabet $\mathcal{A}_{[\Gamma \vdash T]}$ defined as follows: $\mathcal{A}_{[x:T]} = x.\mathcal{A}_{[T]}$, $\mathcal{A}_{[\Gamma]} = \bigcup_{x:T \in \Gamma} \mathcal{A}_{[x:T]}$, and $\mathcal{A}_{[\Gamma \vdash T]} = \mathcal{A}_{[\Gamma]} \cup \mathcal{A}_{[T]}$.

Processes for constants and free identifiers $x : T \vdash x : T$ are defined in Table 1. The process for diverge_B performs the div process after communicating the initial question event.

Table 1. Processes for constants and free identifiers

$$\begin{aligned} \llbracket \Gamma \vdash v : \text{exp}D \rrbracket &= Q.q \rightarrow A.v \rightarrow \text{SKIP}, \quad v \in \mathcal{A}_{[D]} \\ \llbracket \Gamma \vdash \text{skip} : \text{com} \rrbracket &= Q.run \rightarrow A.done \rightarrow \text{SKIP} \\ \llbracket \Gamma \vdash \text{diverge}_B : B \rrbracket &= Q?q : Q_{[B]} \rightarrow \text{div} \\ \llbracket x : \text{exp}D \vdash x : \text{exp}D \rrbracket &= Q.q \rightarrow x.Q.q \rightarrow x.A?a : A_{[\text{exp}D]}^q \rightarrow A.a \rightarrow \text{SKIP} \\ \llbracket x : \text{com} \vdash x : \text{com} \rrbracket &= Q.run \rightarrow x.Q.run \rightarrow x.A.done \rightarrow A.done \rightarrow \text{SKIP} \\ \llbracket x : \text{var}D \vdash x : \text{var}D \rrbracket &= (Q.read \rightarrow x.Q.read \rightarrow x.A?a : A_{[\text{var}D]}^{\text{read}} \rightarrow A.a \rightarrow \text{SKIP}) \\ &\quad \square (Q.write?v : \mathcal{A}_{[D]} \rightarrow x.Q.write.v \rightarrow x.A.ok \rightarrow A.ok \rightarrow \text{SKIP}) \\ \llbracket x : B_1 \rightarrow \dots \rightarrow B_k \rightarrow B \vdash x : B_1 \rightarrow \dots \rightarrow B_k \rightarrow B \rrbracket &= Q?q : Q_{[B]} \rightarrow x.Q.q \rightarrow \\ &\quad \mu L. \left(\text{SKIP} \square \left(\bigcap_{j=1}^k (x.Q.j?q_j : Q_{[B_j]} \rightarrow Q.j.q_j \rightarrow A.j?a_j : A_{[B_j]}^{q_j} \rightarrow \right. \right. \\ &\quad \left. \left. x.A.j.a_j \rightarrow \text{SKIP}) \right) \right) \text{;} x.A?a : A_{[B]}^q \rightarrow A.a \rightarrow \text{SKIP} \end{aligned}$$

For each language construct ‘ c ’, a process P_c which corresponds to its strategy is defined in Table 2. For example, P_{or} nondeterministically runs either its first or its second argument. Events of the first (resp., second) argument of ‘ or ’ occur on channels tagged with index 1 (resp., 2). Then, for each composite term $c(M_1, \dots, M_n)$ consisting of a language construct ‘ c ’ and subterms M_1, \dots, M_n , we define $\llbracket c(M_1, \dots, M_n) \rrbracket$ from the process P_c and processes $\llbracket M_i \rrbracket$ and $\llbracket M_i \rrbracket^{*5}$, using only the CSP operators of renaming, parallel composition and hiding. For example, the process for ‘ or ’ is defined as:

$$\begin{aligned} \llbracket \Gamma \vdash M_1 \text{ or } M_2 : B \rrbracket &= (\llbracket \Gamma \vdash M_1 : B \rrbracket [Q_1/Q, A_1/A] \square \text{SKIP}) \parallel_{\{Q_1, A_1\}} \\ &\quad ((\llbracket \Gamma \vdash M_2 : B \rrbracket [Q_2/Q, A_2/A] \square \text{SKIP}) \parallel_{\{Q_2, A_2\}} \\ &\quad P_{\text{or}} \setminus \{Q_2, A_2\}) \setminus \{Q_1, A_1\} \end{aligned}$$

After renaming the channels Q, A to Q_1, A_1 in the process for M_1 , and to Q_2, A_2 in the process for M_2 respectively, the processes for M_1 and M_2 are composed

⁵ P^* is a process which performs the process P arbitrary many times.

with P_{or} . The composition is performed by synchronising the component processes on events occurring on channels Q_1, A_1, Q_2, A_2 , which are then hidden. Since one of the processes for M_1 and M_2 will not be run in the composition, $SKIP$ is used to enable such empty termination.

Table 2. Processes for constructs

$P_{\text{op}} = Q.q \rightarrow Q_1.q \rightarrow A_1?a_1 : A_{\llbracket \text{exp}D \rrbracket}^q \rightarrow Q_2.q \rightarrow A_2?a_2 : A_{\llbracket \text{exp}D \rrbracket}^q$ $\rightarrow A.a_1 \text{ op } a_2 \rightarrow SKIP$
$P_! = Q?q : Q_{\llbracket B \rrbracket} \rightarrow Q_1.run \rightarrow A_1.done \rightarrow Q_2.q \rightarrow A_2?a : A_{\llbracket B \rrbracket}^q \rightarrow A.a \rightarrow SKIP$
$P_{\text{if}} = Q.q : Q_{\llbracket B \rrbracket} \rightarrow Q_0.q \rightarrow A_0?a_0 : A_{\llbracket \text{expbool} \rrbracket}^q \rightarrow \text{if } (a_0) \text{ then } (Q_1.q \rightarrow$ $A_1?a_1 : A_{\llbracket B \rrbracket}^q \rightarrow A.a_1 \rightarrow SKIP) \text{ else } (Q_2.q \rightarrow A_2?a_2 : A_{\llbracket B \rrbracket}^q \rightarrow A.a_2 \rightarrow SKIP)$
$P_{\text{while}} = Q.run \rightarrow \mu p . Q_1.q \rightarrow A_1?a_1 : A_{\llbracket \text{expbool} \rrbracket}^q \rightarrow \left(\text{if } (a_1) \text{ then } \right.$ $\left. (Q_2.run \rightarrow A_2.done \rightarrow p) \text{ else } (A.done \rightarrow SKIP) \right)$
$P_{:=} = Q.run \rightarrow Q_1.q \rightarrow A_1?a : A_{\llbracket \text{exp}D \rrbracket}^q \rightarrow Q_2.write.a \rightarrow A_2.ok \rightarrow A.done \rightarrow SKIP$
$P_{!} = Q.q \rightarrow Q_1.read \rightarrow A_1?a : A_{\llbracket \text{var}D \rrbracket}^{\text{read}} \rightarrow A.a \rightarrow SKIP$
$P_{\text{or}} = (Q.q : Q_{\llbracket B \rrbracket} \rightarrow Q_1.q \rightarrow A_1?a_1 : A_{\llbracket B \rrbracket}^q \rightarrow A.a_1 \rightarrow SKIP) \square$ $(Q.q : Q_{\llbracket B \rrbracket} \rightarrow Q_2.q \rightarrow A_2?a_2 : A_{\llbracket B \rrbracket}^q \rightarrow A.a_2 \rightarrow SKIP)$
$P_{\text{new}}(x, v) = Q.run \rightarrow Q_1.run \rightarrow U_D(x, v)$
$U_D(x, v) = (x.Q.read \rightarrow x.A.v \rightarrow U_D(x, v)) \square$ $(x.Q.write?v' : A_{\llbracket D \rrbracket} \rightarrow x.A.ok \rightarrow U_D(x, v')) \square (A_1.done \rightarrow A.done \rightarrow SKIP)$

Example 2. Consider the term from Example 1:

$$f : \text{com} \rightarrow \text{com} \vdash \text{newint } x := 0 \text{ in } (f(x := 1) ; \text{if } (x = 1) \text{ then } \text{diverge}_{\text{com}})$$

$$\text{or } (f(x := 1) ; \text{if } (x = 0) \text{ then } \text{diverge}_{\text{com}}) : \text{com}$$

The LTS of the CSP process representing this term is shown in Fig. 1. The first argument of ‘or’ terminates successfully when f does not call its argument; otherwise it diverges. The second argument of ‘or’ terminates successfully when f calls its argument, one or more times; otherwise it diverges. The set of divergences is $Q.run \cdot f.Q.run \cdot (f.Q.1.run \cdot f.A.1.done)^* \cdot f.A.done$, while the set of traces that end with \checkmark is $Q.run \cdot f.Q.run \cdot (f.Q.1.run \cdot f.A.1.done)^* \cdot f.A.done \cdot A.done \cdot \checkmark$. Notice that no references to the variable x appear in the model because it is locally defined. \square

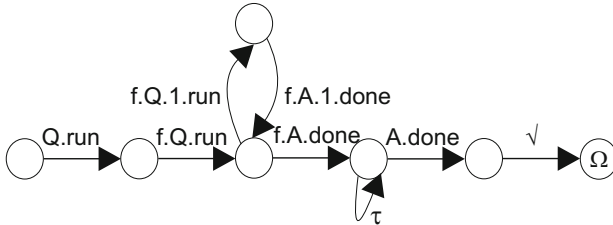


Fig. 1. A strategy as a LTS

5 Correctness and Formal Properties

We now show that for any term from 2nd-order recursion-free EIA with finite data types, the sets of all terminated traces and divergences of its CSP interpretation are isomorphic to the sets of all complete plays and divergences of its fully abstract game semantic model. Given a term $\Gamma \vdash M : T$, we denote by $\llbracket \Gamma \vdash M : T \rrbracket^{GS}$ its game semantic model as described in Section 3, and we denote by $\llbracket \Gamma \vdash M : T \rrbracket^{CSP}$ its CSP interpretation as described in Section 4.

Theorem 1. *For any term $\Gamma \vdash M : T$, we have:*

$$\begin{aligned} \text{traces}^\vee(\llbracket \Gamma \vdash M : T \rrbracket_T^{CSP}) &\stackrel{\phi}{\cong} \text{comp}(\llbracket \Gamma \vdash M : T \rrbracket^{GS}) \\ \text{div}(\text{divergences}(\llbracket \Gamma \vdash M : T \rrbracket_{FD}^{CSP})) &\stackrel{\phi}{\cong} D_{\llbracket \Gamma \vdash M : T \rrbracket^{GS}} \end{aligned}$$

where $\text{traces}^\vee(P_T)$ is the set of all terminated traces of process P that end with \vee in its traces semantics, $\text{div}(\text{divergences}(P_{FD}))$ is the set of all minimal divergences of P in its failures-divergences semantics, and ϕ is an isomorphism defined by:

- For a type T of the form $B_1 \rightarrow \dots \rightarrow B_k \rightarrow B$:
 - $\phi(a) = L.j.a$, for $a \in M_{\llbracket B_j \rrbracket^{GS}}$, $\lambda^{\text{QA}}(a) = L$, $1 \leq j \leq k$
 - $\phi(a) = L.a$, for $a \in M_{\llbracket B \rrbracket^{GS}}$, $\lambda^{\text{QA}}(a) = L$
- For any $x : B'_1 \rightarrow \dots \rightarrow B'_{k_x} \rightarrow B' \in \Gamma$:
 - $\phi(a) = x.L.i.a$, for $a \in M_{\llbracket B'_i \rrbracket^{GS}}$, $\lambda^{\text{QA}}(a) = L$, $1 \leq i \leq k_x$
 - $\phi(a) = x.L.a$, for $a \in M_{\llbracket B' \rrbracket^{GS}}$, $\lambda^{\text{QA}}(a) = L$

Proof. The proof is by a routine induction on the typing rules. □

Corollary 1

$$\Gamma \vdash M \sqsubseteq_{\text{may}} N \Leftrightarrow \llbracket \Gamma \vdash N : T \rrbracket^{CSP} \square \text{RUN}_{\mathcal{A}_{\llbracket \Gamma \vdash T \rrbracket}} \sqsubseteq_T \llbracket \Gamma \vdash M : T \rrbracket^{CSP}$$

where $\text{RUN}_A = \mu p. ?x : A \rightarrow p$, i.e. it is a process which can perform any event from the set A , but it cannot perform \vee or any other event not in A .

Proof. It follows from Proposition 2, Theorem 1, and the traces semantics of the \square operator and the $\text{RUN}_{\mathcal{A}_{\llbracket \Gamma \vdash T \rrbracket}}$ process. □

The checks performed by FDR terminate only for finite-state processes, i.e. those whose labelled transition systems are finite. It is easy to show that this is the case for the processes interpreting the EIA terms by extending the same result for IA terms in [9]. As a corollary, we have that observational may-approximation is decidable using FDR.

Corollary 2. *Observational may-approximation and may-equivalence of EIA terms are decidable by using FDR tool.*

Example 3. Consider the process for the term M from Examples 1 and 2. As shown in Fig. [1](#), we have that $\text{traces}^\vee(\llbracket M \rrbracket_T^{CSP}) = Q.run \cdot f.Q.run \cdot (f.Q.1.run \cdot f.A.1.done)^* \cdot f.A.done \cdot A.done \cdot \checkmark$. But, this is the same as the set of all terminated traces of the process for $f : \text{com} \rightarrow \text{com} \vdash f(\text{skip}) : \text{com}$. So, these two terms are may-equivalent. \square

By Proposition [3](#) and Theorem [1](#), we have that *must-approximation* $\Gamma \vdash M \sqsubseteq_{\text{must}} N$ can be determined by the following procedure:

- (1) Check: $\llbracket \Gamma \vdash M \rrbracket^{CSP} \sqsubseteq_T \llbracket \Gamma \vdash N \rrbracket^{CSP}$, $\llbracket \Gamma \vdash N \rrbracket^{CSP}$ is divergence-free, and $\llbracket \Gamma \vdash M \rrbracket^{CSP}$ is divergence-free. If all three checks hold, then terminate with answer $\Gamma \vdash M \sqsubseteq_{\text{must}} N$, else go to (2).
- (2) Let C_1 , C_2 , and C_3 be the sets of all minimal counterexamples returned by the above three checks respectively. Set $C := C_1 \cup (C_2 \setminus C_3)$. If $C_3 = \emptyset$, then terminate with answer $\Gamma \vdash M \not\sqsubseteq_{\text{must}} N$, else go to (3).
- (3) For each $c \in C$, check whether there exists $s' \sqsubseteq^{\text{odd}} c$, such that $s' \sqsubseteq d$ for some $d \in C_3$. If this is correct, then terminate with $\Gamma \vdash M \sqsubseteq_{\text{must}} N$, otherwise with $\Gamma \vdash M \not\sqsubseteq_{\text{must}} N$.

Proposition 4. *The procedure for determining must-approximation is correct.*

Proof. We can check by inspection that all answers returned by the procedure are correct. Let the procedure terminate in Step (1). Then, $\text{traces}(\llbracket \Gamma \vdash N \rrbracket^{CSP}) \subseteq \text{traces}(\llbracket \Gamma \vdash M \rrbracket^{CSP})$, $\text{divergences}(\llbracket \Gamma \vdash N \rrbracket^{CSP}) = \emptyset$, and $\text{divergences}(\llbracket \Gamma \vdash M \rrbracket^{CSP}) = \emptyset$. So, we have that $(T_{\llbracket \Gamma \vdash N \rrbracket^{GS}} \cup D_{\llbracket \Gamma \vdash N \rrbracket^{GS}}) \setminus (T_{\llbracket \Gamma \vdash M \rrbracket^{GS}} \cup D_{\llbracket \Gamma \vdash M \rrbracket^{GS}}) = \emptyset$, which implies that $\Gamma \vdash M \sqsubseteq_{\text{must}} N$. The other cases are similar. \square

Corollary 3. *Observational must-approximation and must-equivalence of EIA terms are decidable by using FDR tool.*

Example 4. We can verify that the term M from Examples 1 and 2 is must-equivalent with $\vdash \text{diverge}_{\text{com}}$. Let C_1 , C_2 , C_3 , and C_4 be the minimal counterexamples associated with the following checks: $\llbracket \text{diverge}_{\text{com}} \rrbracket^{CSP} \sqsubseteq_T \llbracket M \rrbracket^{CSP}$, $\llbracket M \rrbracket^{CSP} \sqsubseteq_T \llbracket \text{diverge}_{\text{com}} \rrbracket^{CSP}$, $\llbracket M \rrbracket^{CSP}$ and $\llbracket \text{diverge}_{\text{com}} \rrbracket^{CSP}$ are divergence-free, respectively. Then, $C_1 = \{Q.run \cdot f.Q.run\}$, $C_2 = \emptyset$, $C_3 = \{Q.run \cdot f.Q.run \cdot f.A.done\}$, $C_4 = \{Q.run\}$. By following the previously described procedure for determining must-approximation, it is easy to check that these two terms are must-equivalent. \square

In addition to checking observational equivalence of two terms, it is desirable to be able to check properties, safety (see [9,8](#) for details) and liveness, of terms. By Proposition [1](#) and Theorem [1](#), we have that:

Corollary 4. *Must-termination of a term $\Gamma \vdash M$ is decidable using FDR tool by checking one divergence-freedom test:*

$$\llbracket \Gamma \vdash M \rrbracket^{CSP} \text{ is divergence-free} \tag{1}$$

If the test (II) does not hold, then the term *diverges* and one or more counter-examples reported by the FDR debugger can be used to explore the reasons why. Otherwise, the term does not diverge, i.e. it *terminates*.

Example 5. By testing the process for the term `while (true) do skip` for divergence-freedom, we can verify that the term diverges. The counter-example is: $Q.run$. We can also verify that the term from Examples 1 and 2 diverges. The obtained counter-example is:

$$Q.run \ f.Q.run \ f.A.done \quad \square$$

6 Applications

We have implemented a tool, which automatically converts a term into a CSP process which represents its game semantics. The resulting CSP process is defined by a script in machine readable CSP [15] which the tool outputs. In the input syntax, we use simple type annotations to indicate what finite sets of integers will be used to model integer free identifiers and local variables. An integer constant n is implicitly defined of type int_{n+1} . An operation between values of types int_{n_1} and int_{n_2} produces a value of type $\text{int}_{\max\{n_1, n_2\}}$. The operation is performed modulo $\max\{n_1, n_2\}$.

We now analyse an implementation of the linear search algorithm:

```

x[k] : varint2, y : expint2 ⊢
newint2 a[k] := 0 in
newintk+1 i := 0 in
while (i < k) do { a[i] := x[i]; i := i + 1; }
newint2 z := y in
newbool present := false in
while (not present) do {
  if (i < k) then if (a[i] = z) then present := true;
  i := i + 1; } : com
    
```

The code includes a meta variable $k > 0$, representing array size, which will be replaced by several different values. The data stored in the arrays and the expression y is of type int_2 , i.e. two distinct values 0 and 1 can be stored, and the type of index i is int_{k+1} , i.e. one more than the size of the array. The program

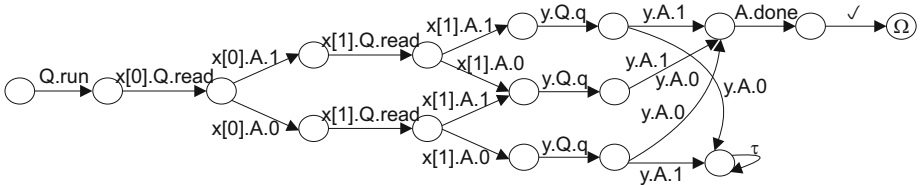


Fig. 2. Model for linear search with $k=2$

Table 3. Model generation of linear search

Arr size k	Time (sec)	Model states
5	2	35
10	5	65
20	39	125
30	145	185

first copies the input array x into a local array a , and the input expression y into a local variable z . Then, the local array is searched for an occurrence of the value y . The array being effectively searched, $a[]$, and the variable z , are not visible from the outside of the term because they are locally defined, so only reads from the non-local identifiers x and y are seen in the model of this term.

A labelled transition system of the CSP process for the term with $k = 2$ is shown in Fig. 2. It illustrates the possible behaviours of this term: if the value read from y has occurred in $x[]$ then the term terminates successfully; otherwise the term diverges. If we test this process for divergence-freedom, we obtain the following counter-example:

$$Q.run\ x[0].Q.read\ x[0].A.1\ x[1].Q.read\ x[1].A.1\ y.Q.q\ y.A.0$$

So the linear search term diverges when the value read from y does not occur in the array $x[]$ making the while loop forever.

Table 3 shows some experimental results for checking divergence-freedom. The experiment consisted of running the tool on the linear search term with different values of k , and then letting FDR generate its model and test its divergence-freedom. The latter stage involved a number of hierarchical compressions, as described in 9. For different values of k , we list the execution time in seconds, and the size of the final model. We ran FDR on a Machine AMD Sempron Processor 3500⁺ with 2GB RAM.

7 Conclusion

We presented a compositional approach for verifying equivalence and liveness properties of nondeterministic sequential programs with finite data types. An interesting direction for extension is to consider infinite integers with all the usual operators. Counter-example guided abstraction refinement procedures [7,8] for verifying safety properties can be adopted to the specific setting for verifying liveness properties. It is also important to extend the proposed approach to programs with concurrency [10], probabilistic constructs [13], and other features.

References

1. Abramsky, S., McCusker, G.: Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. In: O’Hearn, P.W., Tennent, R.D. (eds.) Algol-like languages. Birkhäuser, Basel (1997)

2. Bakewell, A., Ghica, D.R.: On-the-fly techniques for game-based software model checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 78–92. Springer, Heidelberg (2008)
3. Colon, M.A., Sipma, H.B.: Practical Methods for Proving Program Termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
4. Colon, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
5. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction Refinement for Termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
6. Dimovski, A., Lazić, R.: CSP Representation of Game Semantics for Second-order Idealized Algol. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 146–161. Springer, Heidelberg (2004)
7. Dimovski, A., Ghica, D.R., Lazić, R.: Data-Abstraction Refinement: A Game Semantic Approach. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 102–117. Springer, Heidelberg (2005)
8. Dimovski, A., Ghica, D.R., Lazić, R.: A Counterexample-Guided Refinement Tool for Open Procedural Programs. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 288–292. Springer, Heidelberg (2006)
9. Dimovski, A., Lazić, R.: Compositional Software Verification Based on Game Semantics and Process Algebras. *Int. Journal on STTT* 9(1), 37–51 (2007)
10. Ghica, D.R., Murawski, A.S.: Angelic semantics of fine-grained concurrency. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 211–225. Springer, Heidelberg (2004)
11. Harmer, R., McCusker, G.: A fully abstract game semantics for finite nondeterminism. In: Proceedings of LICS, pp. 422–430. IEEE, Los Alamitos (1999)
12. Harmer, R.: Games and Full Abstraction for Nondeterministic Languages. Ph. D. Thesis Imperial College (1999)
13. Legay, A., Murawski, A., Ouaknine, J., Worrell, J.: On Automated Verification of Probabilistic Programs. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 173–187. Springer, Heidelberg (2008)
14. Murawski, A.: Reachability Games and Game Semantics: Comparing Nondeterministic Programs. In: Proceedings of LICS, pp. 173–183. IEEE, Los Alamitos (2008)
15. Roscoe, W.A.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)

Verification Architectures: Compositional Reasoning for Real-Time Systems*

Johannes Faber

Department of Computing Science, University of Oldenburg, Germany
j.faber@uni-oldenburg.de

Abstract. We introduce a conceptual approach to decompose real-time systems, specified by integrated formalisms: instead of showing safety of a system directly, one proves that it is an instance of a Verification Architecture, a safe behavioural protocol with unknowns and local real-time assumptions. We examine how different verification techniques can be combined in a uniform framework to reason about protocols, assumptions, and instantiations of protocols. The protocols are specified in CSP, extended by data and unknown processes with local assumptions in a real-time logic. To prove desired properties, the CSP dialect is embedded into dynamic logic and a sequent calculus is presented. Further, we analyse the instantiation of protocols by combined specifications, here illustrated by CSP-OZ-DC. Using an example, we show that this approach helps us verify specifications that are too complex for direct verification.

1 Introduction

In the analysis of real-time systems, several aspects have to be covered, e.g., (1) behaviour that conforms to communication protocols, (2) rich data structures, and (3) timing constraints such that the system reacts timely to external events. Thus, in practise it turns out that, to adequately handle those systems, engineers fall back on combinations of techniques. An example is the wide acceptance of the UML, combining multiple graphical notations for different views of a design. Similarly, in the world of formal analysis there has been a lot of work on integrating specification techniques to condense advantages of single formalisms into combined formalisms, e.g., [2,25,19,11,10,32,28,15,29]. However, a major problem remains: these integrated techniques are designed for heterogeneous systems, i.e., they cover several different aspects, and when formally verifying those systems we have to cope with their inherent complexity.

We thus propose a Verification Architecture approach to verify global properties by combining local analyses. This idea originates from previous case studies [21], where we decomposed a train control system according to its abstract behavioural protocol that splits the system runs into several phases (e.g., braking, running) with local real-time properties that hold during these phases. After

* This work was partly supported by the German Research Council (DFG) under grant SFB/TR 14 AVACS. See <http://www.avacs.org> for more information.

showing the correctness of a desired global property for this protocol, the global property is also guaranteed by all instances of the protocol for that the local properties are satisfied. We generalise and formalise this approach in the context of combined languages. The approach is structured into several layers:

1. abstract behavioural protocols with unknowns, that have a large degree of freedom to comprise a large class of concrete systems, need to be specified and verified with respect to desired safety properties;
2. since the analysed systems are often time-dependent it is important to allow imposing of additional real-time assumptions on protocol phases and it must be possible to verify the protocols taking these assumptions into account;
3. it needs to be checked that concrete models, given as combined specifications to capture heterogeneous systems, are instantiations of the protocol;
4. it needs to be checked that concrete models actually guarantee the assumptions on the protocol phases.

The challenge is to tackle each layer of the problem with a suitable formalisation and to integrate the heterogeneous formalisations into a uniform framework.

As several combined specification formalisms base on Communicating Sequential Processes (CSP) [14,26], we propose to use CSP to specify system protocols with unknowns. Thus, we define a CSP extension by data constraints and unknown processes and show that it is suited to specify abstract system protocols. In addition, we allow the unknown processes to be constrained by formulae in an arbitrary temporal logic. We call this combination of protocol with local assumptions *Verification Architecture* (VA). To establish safety properties on VAs, we embed our CSP extension into a temporal dynamic logic [12,22] and introduce a sound sequent-style calculus [11] over this logic that allows for establishing desired properties under local real-time assumptions. We prove that all specifications that refine the architecture's CSP part and for that the local assumptions are valid directly inherit the desired properties.

To exemplify the instantiation of VAs by a combined specification language, we choose CSP-OZ-DC (COD) as instantiation language and Duration Calculus (DC) [33] for the local assumptions on protocol phases. We introduce a simple syntactical proof rule to show efficiently that a concrete specification is a VA refinement. The correctness of the local assumptions can be shown using an established model checking approach for COD and DC [21]. Using a running example motivated by the European Train Control System (ETCS) [7], we provide evidence that our method enables the verification of a system that is too large to be verified without decomposition techniques.

We summarise our contributions:

Section 1. We provide a new conceptual approach on how to use behavioural protocols, called Verification Architectures (VA), as a decomposition technique to enable verification of realistic systems specified by combined formalisms.

Section 2. We introduce a CSP dialect with data, unknown process parts, and local real-time assumptions for the specification of VAs.

Section 3. A new sequent-style calculus over this CSP dialect allows us to verify desired properties of VAs. We establish basic properties of the calculus.

Section 4. We examine the instantiation of VAs by COD specifications and give a proof rule to syntactically check refinement relations.

1.1 Discussion of Related Work

Our approach is inspired by [3], where for a fixed DC protocol a design pattern for cooperating traffic agents is introduced. Also, [3] motivates the work of [17], in which CSP-OZ-DC patterns are applied as a formal counterpart to some standard patterns from software engineering. In contrast to our approach no formal framework for the use, application, and verification of design patterns is introduced. A general view on formalisation techniques for design patterns is given in [30], but there, verification of real-time systems is not considered. [6] presents an approach using patterns for a combined real-time language: timed automata patterns for a set of timing constraints are formally linked to TCOZ.

The work [18] introduces context systems that can be instantiated with concrete processes, a concept similar to the unknown processes of this paper. They consider arbitrary process algebras as context systems and the generation of compositional assumptions in Hennessy-Milner logic whereas we use a fixed process algebra but allow arbitrary real-time logics for the assumptions. In [18], no real-time aspects and no data aspects are considered.

Our approach can be seen as Assume-Guarantee reasoning in the context of combined, parametric specifications, because we show validity of global properties assuming local component properties. [4] contains a general introduction into Assume-Guarantee reasoning without time and without the context of joint verification techniques. In [20] a verification approach for CSP-OZ (without time) is investigated that does not consider decompositions by given protocols but instead uses a learning-based algorithm to generate assumptions on layered components. [5] presents an Assume-Guarantee based, sound and complete proof system to reason about CCS processes with Hennessy-Milner assumptions on the environment of processes. In contrast to the approach of this paper, these unknown parts with assumptions are not explicitly represented as process expressions and, thus, are always composed in parallel to the known process of the system. In addition, neither real-time properties, data constraints, nor combined specifications are considered. Both approaches have in common that concrete systems instantiating the unknown parts and satisfying the assumptions inherit the properties of the abstract system.

Our CSP extension by data and unknown processes enables us to specify parametric systems because (1) we can use global data parameters and (2) unknown processes give a parametric view to process components that are not fixed but represent a class of concrete processes. In doing so, we provide a general formalism that is on the one hand flexible enough to express behavioural protocols with a large degree of freedom and on the other hand integrates well with combined specification formalisms based on CSP [19,10,32,28,15,29]. So, our goal was not to introduce a further combination of CSP with data as a replacement for existing formalisms but to provide a notation for VAs that can be used in combination with these formalisms. It turned out that direct usage of a combined formalism

like [15] is not appropriate for a proof rule approach because of the complex combination of languages in an object-oriented structure.

[22,23] introduce a sequent calculus for temporal dynamic logic to verify temporal properties for hybrid systems; they also examine fragments of the ETCS as case study. The work [16] introduces a sequent calculus to verify the Java part of JCSP programs and a translation to Petri nets for the CSP library calls. Recursion in CSP processes and timing constraints are not considered.

Our instantiation rule for COD is not intended to be complete—it is defined as an efficient syntactic refinement check. General results on refinement or subtyping in CSP-OZ and related formalisms can be found in [10] and [31].

1.2 The Verification Architecture Approach

Let $prtcl(\bar{p}, P_1, \dots, P_n)$ be an abstract behavioural protocol depending on a vector of data parameters \bar{p} and process parameters P_i . Additionally, we consider assumptions in a temporal logic on the P_i , $asm_1(\bar{p}), \dots, asm_n(\bar{p})$, that also depend on the parameters. We denote the combination of behavioural protocol and temporal assumptions as *Verification Architecture* (VA). Our aim is to show that a safety property $safe(\bar{p})$ is valid for every possible model that is a refinement of the behavioural protocol and that respects the assumptions.

To apply our approach, we have to show that the VA is correct, i.e., the protocol is correct for all parameters and processes respecting the assumptions:

$$\forall \bar{p}, P_i \bullet \left(\bigwedge_{i=1, \dots, n} P_i \models asm_i(\bar{p}) \right) \Rightarrow (prtcl(\bar{p}, P_1, \dots, P_n) \models safe(\bar{p})) \quad (1)$$

This verification task to verify the correctness of the parametric VA is for realistic systems not necessarily easy and we will provide proof rules for the verification. But once it is verified, this result is reusable as all instantiations of this architecture inherit the correctness property automatically. We only have to show that a potential instantiation is a refinement of the protocol and that the local assumptions are valid, which is due to their locality easier than to verify the global property directly. To be more concrete, we consider the concrete model $spec(\bar{p}, P_1^0, \dots, P_n^0)$, where the P_i^0 are instantiations of the process parameters. Firstly, we have to show that every trace of this model (without assumptions) is also a trace of the protocol:

$$\forall \bar{p} \bullet [spec(\bar{p}, P_1^0, \dots, P_n^0)] \subseteq [prtcl(\bar{p}, P_1^0, \dots, P_n^0)]. \quad (2)$$

This refinement relation on the processes can be shown syntactically for a specific class of instantiations (cf. Sect. 4). Thus, it is easy to verify. Secondly, we have to show that the assumptions are valid for the concrete specification:

$$\forall \bar{p} \bullet P_i^0 \models asm_i(\bar{p}) \text{ for all } i = 1..n. \quad (3)$$

This can be done by applying existing verification techniques for the language of the assumptions. With this, our approach yields that the desired safety property

is valid for the concrete model. We argue that this proposition is correct. From (1) and (3) we can conclude (4) and with (2) we get the desired property (5).

$$\forall \bar{p} \bullet \text{prctl}(\bar{p}, P_1^0, \dots, P_n^0) \models \text{safe}(\bar{p}) \quad (4)$$

$$\forall \bar{p} \bullet \text{spec}(\bar{p}, P_1^0, \dots, P_n^0) \models \text{safe}(\bar{p}) \quad (5)$$

We summarise that if a correct VA is given, we only have to show that, firstly, the model's process is a refinement of the abstract protocol and, secondly, the model respects the assumptions. Then, we say that the model is an *instantiation* of the VA and we can conclude that it inherits the VA's correctness.

2 CSP Processes with Data Constraints and Unknowns

In this section, we introduce a CSP extension by data constraints and unknown processes to specify Verification Architectures (VA).

For specifying VAs, a high degree of freedom is necessary to handle general patterns of parametric systems with data. To this end, we extend CSP by data constraints to define state changes and by a new construct, so-called *unknown processes*. Unknown processes are special processes that allow the occurrence of arbitrary events except for events from a fixed alphabet and arbitrary changes of variables except for variables from a fixed set. They can terminate and may be restricted by constraints from an arbitrary real-time logic.

Syntax. We consider many-sorted first order formulae $Form_\Sigma$ from a signature $\Sigma = (Sort, Symb, Var, Par)$ with primed and unprimed system variables and functions $Symb$ with sorts $Sort$, variables Var , and parameters Par with fixed but arbitrary values. The syntax of *CSP processes with data and unknown processes* is given by

$$P ::= \text{Stop} \mid \text{Skip} \mid (a \bullet \varphi) \rightarrow P \mid P_1 \square P_2 \mid P_1 \parallel P_2 \mid P_1 \parallel_A P_2 \mid P_1 \circledast P_2 \mid X \\ \mid (\text{Proc}_{\setminus A, V} \bullet F) \mid (\text{Proc}_{\setminus A, V}^\infty \bullet F)$$

where $a \in Events$, $A \subseteq Events$, $\varphi \in Form_\Sigma$, $V \subseteq Symb$ and F is a constraint in a temporal logic with the same semantical domain as CSP with data constraints. In this definition, a difference to the standard CSP definition is that we have constrained occurrences of events a by formulae φ , denoted $a \bullet \varphi$. The intuition is that when the event a occurs the state space is changed according to the constraint φ , where unprimed symbols in φ refer to valuations before the occurrence of a and primed symbols to the valuations after a . The intuition behind an unknown process like $(\text{Proc}_{\setminus \{a, b\}, \{v\}} \bullet F)$ is that during the execution of the process arbitrary behaviour is allowed provided that the formula F is not violated. The events a and b are forbidden and the system variable v cannot be changed in this execution. A process Proc^∞ marked with ∞ will never terminate.

Example 1. As a running example we consider a small train control system motivated by the European Train Control System [7]: a so-called Radio Block Center

$$\begin{aligned}
System &\stackrel{c}{=} Go \square Ext & Ext &\stackrel{c}{=} (extend \bullet \varphi_{extend}) \rightarrow System \\
Go &\stackrel{c}{=} FAR \wp (check \bullet \varphi_{check}) \rightarrow ((fail \bullet \varphi_{fail}) \rightarrow REC \square (pass \bullet \varphi_{pass}) \rightarrow System) \\
\varphi_{extend} &= sf' > sf & FAR &\stackrel{c}{=} \mathbf{Proc}_{\setminus A, C} \bullet F_{FAR} \\
\varphi_{check} &= \Xi(sf) \wedge sf \leq RD \wedge \neg ok' & REC &\stackrel{c}{=} \mathbf{Proc}_{\setminus A, C}^{\infty} \bullet F_{REC} \\
&\quad \vee \Xi(sf) \wedge sf > RD \wedge ok' & F_{FAR} &= \neg \diamond(\ell > CT) \wedge \\
\varphi_{fail} &= \Xi(sf) \wedge \neg ok & &\neg \diamond([sf > RD] \wedge \ell < CT \wedge [sf \leq 0]) \\
\varphi_{pass} &= \Xi(sf) \wedge ok & &F_{REC} = \neg \diamond([sf > 0] \wedge [sf \leq 0]) \\
&& A &= \{check, fail, pass, extend\}, C = \{RD, CT\}
\end{aligned}$$

Fig. 1. VA for a train control system

(RBC) grants movement authorities (MA) to a train. The system is considered safe as long as the train stays within the MA. The distance of the train to the end of the MA is given by a real-valued variable sf , reflecting the safety of the system that shall never be below or equal to 0. The position RD is the last position at which the train needs to apply the brakes to stop in time. The train can request extensions of MAs at any time.

Fig. 1 defines this train control system as a VA. The system is described by the CSP process $System$, that consists of a choice of sub-processes Go and Ext . The process Ext can perform an event $extend$, that extends the current MA, which is expressed in the constraint φ_{extend} . The Go process starts with an unknown process FAR that can produce arbitrary behaviour except for events from A , which are all events of this VA, and it cannot change the constants of the specification, RD and CT . We further constrain the unknown process by the real-time DC formula F_{FAR} , that demands that if sf is greater than RD anywhere on the execution of FAR then sf cannot decrease to a value smaller than 0 within CT time units. After termination of FAR the process checks the current value of sf and, depending on that, behaves as the $System$ again or it changes to a safe recovery process REC .

Semantics of CSP processes with data constraints. The semantics of CSP processes with data constraints is given here by interpretations \mathcal{I} that are mappings from a time domain $Time$ (usually \mathbb{N} or \mathbb{R}) into the set of all models: $\mathcal{I} : Time \rightarrow Model$. A model (or valuation) \mathcal{M} for system variables and parameters of sort S is a (partial) mapping into a corresponding domain: $\mathcal{M} : (Symb \cup Par) \mapsto \mathcal{D}_S$. We denote the set of all interpretations by $Intpr$. So, an interpretation is a timed sequence of models that corresponds to state changes performed by constrained event occurrences. Every interpretation belongs to a run of the CSP process. Events are modelled as boolean variables that change their values if a corresponding event occurs. To compute the semantics, we first compute the labelled transition system (LTS) [26, 15] of the CSP process in the standard way as if there were no data part but with events that are annotated with data constraints. We say that an interpretation \mathcal{I} fits to a run $\pi = \langle (a_1 \bullet \varphi_1), (a_2 \bullet \varphi_2), \dots \rangle$ of the LTS

with events a_i iff there are a points $t_0, t_1, \dots \in Time$ with $0 = t_0 < t_1 < t_2 < \dots$ and models $\mathcal{I}(t) = \mathcal{M}_i$ for $t_i \leq t < t_{i+1}$ s.t.

$$\mathcal{M}_i(a_{i+1}) \neq \mathcal{M}_{i+1}(a_{i+1}) \text{ for } i \geq 0,$$

where all a_i are of sort \mathbb{B} in Σ . We call $\langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$ *untimed sequence* of \mathcal{I} . The semantics of a CSP process P (with data constraints) is a mapping from models to sets of interpretations: $[\cdot] : Model \rightarrow \mathbb{P}Intpr$. An interpretation is valid, $\mathcal{I} \in [P]\mathcal{M}$, iff it respects the state changes of the process, i.e., iff

1. there is a run $\pi = \langle (a_1 \bullet \varphi_1), (a_2 \bullet \varphi_2), \dots \rangle$ of the LTS of P such that \mathcal{I} fits to π . Let the resulting untimed sequence be $\langle \mathcal{M}_0, \mathcal{M}_1, \dots \rangle$.
2. $\mathcal{M}_0 = \mathcal{M}$
3. $(\mathcal{M}_{i-1} \cup \mathcal{M}'_i) \models \varphi_i$ for $i > 0$
4. $\mathcal{M}_i(v) = \mathcal{M}_{i+1}(v)$ for all parameter $v \in Par$ and $i \geq 0$
5. if $a_i = \checkmark$ then $\mathcal{M}_{i-1}(v) = \mathcal{M}_i(v)$ for all symbols $v \in Symb$,

where \mathcal{M}'_i is a model for primed symbols, i.e., $\mathcal{M}'_i(f') = \mathcal{M}_i(f)$, and \checkmark the termination symbol of CSP. Note that this definition makes use of CSP's trace semantics. One consequence is that we do not have to distinguish external and internal choice that are equivalent in the trace semantics.

Semantics of unknown processes. The definition above does not capture unknown processes with data constraints. Even though a process $\text{Proc}_{\setminus A, V}$ without temporal constraints can be rewritten to a standard CSP process, we need to take care of the additional constraints. Since these constraints need to be valid everywhere on all traces of the process we do not give (single step) transition rules for constrained unknown processes. Instead, we give transition rules to compute the LTS of unconstrained processes and we additionally demand that for every trace of those processes the constraint is also valid. The set of transition rules for computing the LTS of a CSP process is extended by the rules

$$\frac{}{\text{Proc}_{\setminus A, V}^{(\infty)} \xrightarrow{a \bullet \Xi V} \text{Proc}_{\setminus A, V}^{(\infty)}} \qquad \frac{}{\text{Proc}_{\setminus A, V} \xrightarrow{\checkmark} \Omega}$$

in which $a \in \mathbb{U}_{Events} \setminus A$, i.e., a is in the universe of events (without τ) except A . The process can perform an arbitrary event that is not in the set A and the event's constraint ensures that symbols from the set V are not changed, which is expressed in Z syntax by ΞV . If the process is not marked as an infinite unknown process it can non-deterministically decide to terminate.

The semantics of a constrained process is given by interpretations $\mathcal{I} \in [\text{Proc}_{\setminus A, V}^{(\infty)} \bullet F]\mathcal{M}$ iff $\mathcal{I} \in [\text{Proc}_{\setminus A, V}^{(\infty)}]\mathcal{M}$ and \mathcal{I} is in the semantics of F : $\mathcal{I} \in [F]$. The latter is well-defined because we have demanded that the semantical domain of F is compatible with the semantics of CSP with data constraints. The semantics of a constrained unknown process in the context of a CSP expression can then be computed by exploiting that the trace semantics is a congruence in each CSP operator [26]. So, to compute the semantics of $P \boxtimes \text{Proc}_{\setminus A, V} \bullet F$ with $\boxtimes \in \{\parallel, \square, \circ\}$ we lift the operators to the trace level: iff $\mathcal{I}_1 \in [P]\mathcal{M}$ and $\mathcal{I}_2 \in [\text{Proc}_{\setminus A, V} \bullet F]\mathcal{M}$ then $(\mathcal{I}_1 \boxtimes \mathcal{I}_2) \in [P \boxtimes \text{Proc}_{\setminus A, V} \bullet F]\mathcal{M}$.

3 Verification of Architectures

After introducing the CSP extension for specifying VAs, we go on to the verification of architectures specified by CSP processes. A well-investigated approach for rule-based verification of programs is the sequent calculus [11] over dynamic logic formulae [12]. Hence, to use the advantages of sequent-style reasoning, we first introduce a dynamic logic over CSP processes with data constraints and unknown processes—called dCSP—and then we propose a sequent calculus for this dynamic logic extension.

A dynamic logic over CSP processes. The logic dCSP is a dynamic logic extension that uses CSP processes with data and unknown processes instead of programs within the box operator $[\cdot]$. As we are only interested in safety properties, we omit the diamond operator in this paper. The dynamic logic operator $[P]\delta$ states that after every run of P the formula δ is true, whereas $[P]\Box\varphi$ expresses that on all runs of P always φ holds.

Definition 1 (Syntax of dCSP). *We consider a signature Σ and define the set $Form_{dCSP}$ of dCSP formulae inductively:*

<i>if p is a predicate symbol and θ_i terms</i>	<i>then $p(\theta_1, \dots, \theta_n) \in Form_{dCSP}$</i>
<i>if $\delta_1, \delta_2 \in Form_{dCSP}$</i>	<i>then $(\neg\delta_1), (\delta_1 \wedge \delta_2) \in Form_{dCSP}$</i>
<i>if $\delta \in Form_{dCSP}, x \in Var$</i>	<i>then $(\forall x \bullet \delta), (\exists x \bullet \delta) \in Form_{dCSP}$</i>
<i>if $\delta \in Form_{dCSP}, P$ a CSP process</i>	<i>then $([P]\delta) \in Form_{dCSP}$</i>
<i>if $\varphi \in Form_{dCSP}, P$ a CSP process and φ does not contain a $[\cdot]$</i>	<i>then $([P]\Box\varphi) \in Form_{dCSP}$</i>

We use the convention that a formula φ does not contain $[\cdot]$ -operators, δ does not begin with \Box , whereas γ always represents an arbitrary dCSP formula.

Definition 2 (Semantics of dCSP formulae). *The semantics of a dCSP term θ with sort S is a mapping $[\cdot] : Model \rightarrow \mathcal{D}_S$ defined as usual. The semantics of dCSP formulae is given by models $\mathcal{M} \in Model$:*

$\mathcal{M} \models p(\theta_1, \dots, \theta_n)$	<i>iff $p_{\mathcal{I}}([\theta_1]\mathcal{M}, \dots, [\theta_n]\mathcal{M}) = true$</i>
$\mathcal{M} \models \neg\gamma$	<i>iff $\mathcal{M} \not\models \gamma$</i>
$\mathcal{M} \models \gamma_1 \wedge \gamma_2$	<i>iff $\mathcal{M} \models \gamma_1$ and $\mathcal{M} \models \gamma_2$</i>
$\mathcal{M} \models \forall x \bullet \gamma$	<i>iff for all $d \in \mathcal{D}_S$ holds $\mathcal{M}[x \mapsto d] \models \gamma$</i>
$\mathcal{M} \models \exists x \bullet \gamma$	<i>iff there is a $d \in \mathcal{D}_S$ s.t. $\mathcal{M}[x \mapsto d] \models \gamma$</i>
$\mathcal{M} \models [P]\Box\delta$	<i>iff $\mathcal{I} \models \Box\delta$ holds for all $\mathcal{I} \in [P]\mathcal{M}$</i>
$\mathcal{M} \models [P]\delta$	<i>iff $\mathcal{M}' \models \delta$ holds for all $\mathcal{I} \in [P]\mathcal{M}$ with terminating \mathcal{M}'</i>

Here, S is the sort of variable x . A *terminating model* is the last model of an interpretation that terminates with \checkmark . The formula $\Box\delta$ holds for an interpretation \mathcal{I} , i.e., $\mathcal{I} \models \Box\delta$, iff $\mathcal{I}(t) \models \delta$ for all $t \in Time$.

$$\begin{array}{c}
\frac{\top}{\gamma \top} \quad (\text{P1}) \quad \frac{\top}{\top \gamma} \quad (\text{P2}) \quad \frac{\gamma \top \quad \top \gamma}{\top} \quad (\text{P3}) \quad \frac{}{\varphi \top \varphi} \quad (\text{P4}) \quad \frac{\top \varphi}{\neg \varphi \top} \quad (\text{P5}) \\
\frac{\varphi \top}{\top \neg \varphi} \quad (\text{P6}) \quad \frac{\varphi, \psi \top}{\varphi \wedge \psi \top} \quad (\text{P7}) \quad \frac{\top \varphi \quad \top \psi}{\top \varphi \wedge \psi} \quad (\text{P8}) \quad \frac{\varphi \top \quad \psi \top}{\varphi \vee \psi \top} \quad (\text{P9}) \quad \frac{\top \varphi, \psi}{\top \varphi \vee \psi} \quad (\text{P10}) \\
\frac{\psi \top \quad \top \varphi}{\varphi \Rightarrow \psi \top} \quad (\text{P11}) \quad \frac{\varphi \top \quad \psi}{\top \varphi \Rightarrow \psi} \quad (\text{P12}) \\
\frac{\varphi[t/x], \forall x : T \bullet \varphi \top}{\forall x : T \bullet \varphi \top} \quad (\text{F1}) \quad \frac{\top \varphi[y/x]}{\top \forall x : T \bullet \varphi} \quad (\text{F2}) \quad \frac{\varphi[y/x] \top}{\exists x : T \bullet \varphi \top} \quad (\text{F3}) \quad \frac{\top \varphi[t/x], \exists x : T \bullet \varphi}{\top \exists x : T \bullet \varphi} \quad (\text{F4}) \\
\frac{[P]\gamma}{[Q]\gamma} \quad (\text{C1}) \quad \frac{\delta}{[\text{Skip}]\delta} \quad (\text{C2}) \quad \frac{\varphi}{[\text{Skip}]\square\varphi} \quad (\text{C3}) \quad \frac{[a][P]\delta}{[a \rightarrow P]\delta} \quad (\text{C4}) \quad \frac{[a]\square\varphi \wedge [a][P]\square\varphi}{[a \rightarrow P]\square\varphi} \quad (\text{C5}) \\
\frac{[P_1]\gamma \wedge [P_2]\gamma}{[P_1 \square P_2]\gamma} \quad (\text{C6}) \quad \frac{[P_1][P_2]\delta}{[P_1 \wp P_2]\delta} \quad (\text{C7}) \quad \frac{([P_1]\square\varphi) \wedge ([P_1][P_2]\square\varphi)}{[P_1 \wp P_2]\square\varphi} \quad (\text{C8}) \\
\frac{\psi_{\bar{v}'}^{\bar{v}_0} \Rightarrow \delta_{\bar{v}'}^{\bar{v}_0}}{[a \bullet \psi]\delta} \quad (\text{C9}) \quad \frac{\varphi \wedge [a \bullet \psi]\varphi}{[a \bullet \psi]\square\varphi} \quad (\text{C10}) \quad \frac{[Q]\gamma}{[P]\gamma} \quad (\text{A1}) \quad \frac{[P]\square\varphi}{[P]\varphi} \quad (\text{A2}) \quad \frac{\varphi \top [P]\delta}{[Q]\varphi \top [Q][P]\delta} \quad (\text{A3}) \\
\frac{\Delta \top \varphi_{in}(\bar{y}), \Gamma \quad \varphi_{in}(\bar{y}), \forall \bar{x} \bullet (\varphi_{in}(\bar{x}) \Rightarrow [Q_{\bar{y}}^{\bar{x}}]\gamma(\bar{x})) \top [F(Q)]\gamma(\bar{y})}{\Delta \top [P]\gamma(\bar{y}), \Gamma} \quad (\text{I1}) \\
\frac{}{\psi \top [\text{Proc}_{\mathcal{A}, \mathcal{V}}^{\infty} \bullet F]\delta} \quad (\text{U1}) \quad \frac{}{\Delta, \psi \top [\text{Proc}_{\mathcal{A}, \mathcal{V}} \bullet F]\delta, \Gamma} \quad (\text{U2}) \quad \frac{}{\psi \top [\text{Proc}_{\mathcal{A}, \mathcal{V}}^{(\infty)} \bullet F]\square\varphi} \quad (\text{U3})
\end{array}$$

In (F1) up to (F4), the term t is of type T and y is a fresh variable of type T not occurring elsewhere. The process Q in (C1) is defined by $Q \stackrel{c}{=} P$. A formula $\psi_{\bar{v}'}^{\bar{v}_0}$ denotes the replacement of variables \bar{v} with fresh variables \bar{v}_0 for all v in ψ . In (A1), P and Q are equivalent CSP processes. In (I1), P is a recursive process $P \stackrel{c}{=} F(P)$, $\gamma(\bar{y})$, $\varphi_{in}(\bar{y})$ are formulae over vectors of system variables containing no other system variables besides \bar{y} . For side conditions of (U2), (U3) (that introduce φ) see Sect. 3. We abbreviate $[a \rightarrow \text{Skip}]\gamma$ by $[a]\gamma$ and $a \bullet \varphi \rightarrow P$ by $a \rightarrow P$ if φ is of no relevance.

Fig. 2. Sequent calculus for CSP with data constraints

Sequent Calculus. To prove validity of dCSP formulae, we define a set of verification rules in a sequent-style proof calculus. Given finite sets of formulae Δ and Γ , a *sequent* $\Delta \top \Gamma$ is an abbreviation of the formula $\bigwedge_{\varphi \in \Delta} \varphi \Rightarrow \bigvee_{\psi \in \Gamma} \psi$. Our sequent calculus contains rule schemata of the shape $\frac{\varphi_1 \top \psi_1 \quad \dots \quad \varphi_n \top \psi_n}{\varphi \top \psi}$ that can be instantiated with arbitrary contexts, i.e., for every Δ and Γ the rule $\frac{\Delta, \varphi_1 \top \psi_1, \Gamma \quad \dots \quad \Delta, \varphi_n \top \psi_n, \Gamma}{\Delta, \varphi \top \psi, \Gamma}$ is part of the calculus. As usual, formulae above the line are premises and the formula below the line the consequence: if the premises (and possibly some side-conditions) are true then the consequence also holds.

Symbolic execution of dCSP formulae. The rules of our calculus can be found in Fig. 2. The rules (P1) up to (F4) are standard (cf. e.g., [11]) and resolve First-order formulae. Note that some of the rules like the cut rule (P3) are actually not necessary [11] but included to simplify proofs.

Our new proof rules are given in Fig. 2 from (C1) up to (U3). They symbolically unwind the CSP processes with data constraints and unknown processes along their process structure. The rules correspond to the process operators introduced in Sect. 2. Rules that do not contain a sequent symbol can be applied on both sides of a sequent. Generally, we have two rules for every operator because we need different rules to cover the temporal case and the non-temporal case. In the non-temporal case of the prefix operator (C4), we check that after all executions of $a \rightarrow P$ the property δ holds. In the temporal case (C5), we need to check that φ is valid everywhere on all executions of $a \rightarrow P$: we prove that $\Box\varphi$ holds everywhere on $a \rightarrow \text{Skip}$ and that $[P]\Box\varphi$ holds *after* every execution of a . Rule (C6) splits up choice of processes into a conjunction of formulae. The sequence rules (C7), (C8) are built-up identical to the prefix rules.

The rules (C9) and (C10) perform the symbolical execution of an event step with a corresponding data constraint: the event and the constraint are consumed and replaced by a new constraint representing the data change. Events are only required for synchronisation and can be reduced in this sequential situation. In rule (C9), the constraint ψ of the event a contains primed and unprimed symbols, where the former relates to the post-state of the operation and the latter to the pre-state. After an execution of the data change in ψ the post-state of a system variable needs to coincide with the pre-state of this variable in δ . Hence, we show that the constraint ψ , where every primed symbol v' is replaced by a fresh v_0 , implies $\delta_{\frac{\psi_0}{v}}$, i.e., δ , where every v is replaced by the corresponding v_0 .

The rules (A1) up to (A3) are auxiliary rules for, e.g., \Box -introduction and replacement of equivalent processes. The latter is also used to cope with parallel composition: parallelism is replaced by an equivalent choice of processes.

Induction rules. In contrast to standard dynamic logic, dCSP expresses properties over recursive processes. Hence, we provide induction rule (I1) here to allow reduction of recursion in CSP expressions. The rule is a variant of the *Fixed-point Induction Rule* of [26], but it is adapted to our needs. The premise of rule (I1) consists of two proof commitments. First, we need to show that an initial condition $\varphi_{in}(\bar{y})$ holds in the current context. The intuition is that this formula φ_{in} holds after every cycle of the recursion and implies validity of the desired formula γ for the next cycle and so on. The second commitment¹ contains the inductive argument: assuming that φ_{in} implies that γ holds for an arbitrary process Q we must show that γ also holds for $F(Q)$. The induction hypothesis is hereby given by $\forall \bar{x} \bullet (\varphi_{in}(\bar{x}) \Rightarrow [Q_{\bar{y}}^{\bar{x}}]\gamma(\bar{x}))$. It states that regardless of how φ_{in} is instantiated with system variables \bar{x} , if φ_{in} is valid for these \bar{x} then $[Q_{\bar{y}}^{\bar{x}}]\gamma(\bar{x})$ is also valid. We need to replace system variables \bar{y} in Q by \bar{x} because during symbolical execution of the process $F(Q)$ new symbols are introduced.

¹ The right formula of the premise (and likewise the premise of (U2)) does actually not contain the context formulae Δ and Γ , that are implicit in the remaining rules.

Symbolic execution of constrained unknown processes. Finally, we need rules to handle unknown processes with temporal constraints. The idea is not to handle these constraints in our calculus directly. Instead, we call an external (semi-)decision procedure that checks if the constraints of the unknown process actually ensure properties by which the remaining proof can be completed. Thus, the rules we give here for handling of unknown process can be seen as oracle rules that access external techniques to reason over the temporal constraints. Hence, the rules directly reflect the semantics of constrained unknown processes. Rule (U1) is an axiom expressing the trivial fact that after termination of all non-terminating processes everything is true. For the remaining two rules, the interesting part is contained in the side-conditions. Rule (U2) is correct if for all models \mathcal{M} with $\mathcal{M} \models \psi$ and all interpretations $\mathcal{I} \in [\text{Proc}_{\setminus A, V} \bullet F]\mathcal{M}$ with terminating model $\overline{\mathcal{M}}$ the formula φ holds: $\overline{\mathcal{M}} \models \varphi$. This represents exactly the semantics of $[\text{Proc}_{\setminus A, V} \bullet F]\varphi$. Further, δ has to follow from φ . The rule only applies to terminating unknown processes (for the infinite case we cannot allow to consume the unknown process). Analogously, the side condition that must be proven for application of rule (U3) is that for all models \mathcal{M} with $\mathcal{M} \models \psi$ and all interpretations $\mathcal{I} \in [\text{Proc}_{\setminus A, V}^{(\infty)} \bullet F]\mathcal{M}$ the formula $\Box\varphi$ holds: $\mathcal{I} \models \Box\varphi$.

In this way, we can use an arbitrary proof method for the temporal logic if it is possible to check the side-conditions of rules (U2) and (U3). By this means, our approach flexibly integrates arbitrary timed logics to formulate assumptions on unknown processes.

Example 2. The safety condition we want to prove for the architecture from Fig. 1 is that sf never reaches 0, in terms of dCSP $sf > RD > 0 \vdash [\text{System}]\Box sf > 0$ and we use our sequent calculus to prove its validity. To apply the proof rules for *FAR* and *REC* we need to verify the corresponding formulae F_{FAR} and F_{REC} via the standard model checking approach for COD and DC [21]. We demonstrate this for one branch of the proof tree:

$$\begin{array}{c}
 \text{(U3)} \\
 \frac{sf > RD > 0 \vdash [\text{Proc}_{\setminus A, C} \bullet F_{FAR}]\Box sf > 0}{sf > RD > 0 \vdash [FAR]\Box sf > 0} \text{(C1)} \quad \vdots \\
 \frac{sf > RD > 0 \vdash [FAR]\Box sf > 0 \wedge [FAR][check \rightarrow \dots \rightarrow \text{Skip}]\Box sf > 0}{sf > RD > 0 \vdash [FAR] \wp \dots \rightarrow \text{Skip}]\Box sf > 0} \text{(P8)} \\
 \text{(C8)}
 \end{array}$$

To close the left branch of the tree (we omit the right branch indicated by dots) we apply rule (U3), i.e., we need to verify the side-condition of the rule, which can be done by a DC formula expressing that for all runs fulfilling the F_{FAR} constraints and starting with $sf > RD > 0$ the constraint $\Box sf > 0$ is true. We verified this property automatically with the abstraction refinement model checker ARMC [24] using the approach of [21]. In this way, we successfully applied our sequent calculus to verify the validity of the desired safety property. The entire proof tree (which can be found in an extended version of this paper [9]) consists of 157 nodes and 22 branches. Eight ARMC calls (finished in less than 8 seconds) to solve branches on T_{FAR} and T_{REC} were necessary.

3.1 Correctness and Incompleteness of the Calculus

Theorem 1 (Soundness). *The calculus as presented in Fig. 2 is sound, i.e., validity follows from derivability in the calculus.*

Proof. To prove the soundness of the calculus, we need to prove soundness of every single rule in Fig. 2, which is exactly like for the standard sequent calculus for rules (P1) up to (F4) and which is relatively straightforward for most of the rules for dCSP formulae. Thus, we give only the proof idea of the most interesting case, the induction rule (II):

The process P is recursively defined by $P \stackrel{c}{=} F(P)$, i.e., it is given by the fixed point of F . The set of non-empty, prefix-closed traces of a CSP process is a complete lattice wrt. the \subseteq -order [26]. First, we need to show that, likewise,

$$L_\gamma := \{[Q] \mid \forall \mathcal{N} \in Model : (\mathcal{N} \models \varphi_{in}(\overline{y}) \Rightarrow [Q]\mathcal{N} \subseteq [\gamma(\overline{y})])\}$$

is a complete lattice. We abbreviate $[Q] := \bigcup_{\mathcal{M}} [Q]\mathcal{M}$. The set L_γ contains all traces of processes P for that $[Q]\mathcal{N} \subseteq [\gamma(\overline{y})]$ holds when \mathcal{N} is a model satisfying $\varphi_{in}(\overline{y})$. Second, we define a monotone function on traces by $f([Q]) := [F(Q)]$ and show using the induction hypothesis from (II) that $[Q] \in L_\gamma$ implies $f([Q]) \in L_\gamma$. With this we can conclude that f has a fixed point in L_γ that is equal to $[P]$. By construction of L_γ this means that $[P]\mathcal{M} \subseteq [\gamma(\overline{y})]$ for all \mathcal{M} with $\mathcal{M} \models \varphi_{in}(\overline{y})$. To that we apply the left premise of rule (II) and get the validity of the rule's conclusion. \square

Theorem 2 (Incompleteness). *The calculus as presented in Fig. 2 is incomplete, i.e., we cannot derive every valid formula in the calculus.*

Proof. This is a direct consequence of the integration of an arbitrary temporal logic to constrain unknown processes. By this, we can choose an undecidable logic like the full DC [33] and, thus, cannot resolve the constraints of those unknown processes in every case. \square

4 Refinement of Verification Architectures

We now show how a verified VA can be instantiated by specifications in the combined formalism CSP-OZ-DC (COD) [15]. We recall that in our approach we verify instantiation relations in two steps: assumptions on unknown processes are verified by customary verification approaches for the temporal logic and a refinement relation for the process structure needs to be established. Thus, we here provide a rule to prove refinement by COD specifications syntactically.

CSP-OZ-DC [15,8] combines three well-investigated formalisms into a single language: it uses CSP to model the control flow of a system, Object-Z (OZ) [27] to specify data space and state changes via OZ schemata, and for defining (dense) real-time constraints, it applies DC counterexample traces. A key feature of CSP-OZ-DC is its separation of concerns, because every part, control flow, data space, and timing part can be specified on its own. Its semantic is given in

terms of interpretations $\llbracket \text{cod} \rrbracket$ and it is compositional, thus, if one can establish a safety property for a single part of the specification the property automatically holds likewise for the entire specification. Note that the CSP part is defined in terms of standard CSP (using trace semantics), i.e., it does not support unknown processes and it does not contain data (which is integrated via the OZ part).

Definition 3 (Refinement by COD specifications). *Given a process P and a COD specification cod , a refinement of P by cod , written $P \sqsubseteq \text{cod}$, is given iff*

$$\{\mathcal{I} \mid \mathcal{M}_{\text{init}} \in \text{Init}(\text{cod}), \mathcal{I} \in \llbracket P \rrbracket(\mathcal{M}_{\text{init}})\} \supseteq \llbracket \text{cod} \rrbracket, \quad (6)$$

where $\text{Init}(\text{cod})$ is the set of all models that are valid initial models of cod .

This definition entails that the symbols, which are introduced in cod and thus are interpreted by the model $\mathcal{M}_{\text{init}}$, coincide with the symbols of the signature Σ of a refined process P . Even though this definition does not impose explicitly restrictions on how symbols are declared and used in cod , it implicitly enforces the desired behaviour: whenever a symbol is changed in each execution of P the symbol must be declared in cod .

We now give a proof rule that establishes a refinement relation between a CSP process with data and unknown processes (but without assumptions on unknowns, which are verified separately; cf. Sect. 1.2) and a COD specification: a COD specification refines a process if there is a syntactic matching between them. The rule is not complete, i.e., not all valid refinements can be shown applying the rule. But this is not our goal here, because in our application scenario concrete realisations are modelled with respect to a given VA and thus we assume that the concrete model reflects the structure of the VA directly.

Definition 4 (Matching). *Given a process P with unknown processes $X_1 \stackrel{c}{=} \text{Proc}_{A_1, V_1}^{(\infty)}, \dots, X_n \stackrel{c}{=} \text{Proc}_{A_n, V_n}^{(\infty)}$ and a COD class cod , cod matches P if*

1. *The symbols of the signature Σ of P coincide with the symbols introduced in cod . That is, for $\Sigma = (\text{Sort}, \text{Symb}, \text{Par}, \text{Var})$ the types of cod correspond to the sorts Sort , state and message variables of cod to symbols from Symb , global constants to Par .*
2. *The CSP process of cod (the main process) structurally equals P except that all unknown processes are replaced by implementing processes. We demand that processes implementing Proc^∞ do not contain the **Skip** process.*
3. *For every process P_i implementing $\text{Proc}_{A, V}$ we demand that (1) the forbidden events from A are respected, i.e., $\text{alphabet}(P_i) \cap A_i = \emptyset$, and (2) the function symbols from V are not changed, i.e., given an operation $a \in \text{alphabet}(P_i)$ with a delta list² $\Delta(s_1, \dots, s_n)$ it holds $s_i \notin V$ for $i \in 1..n$.*
4. *For every occurrence of an event $a \bullet \varphi$ in P , where φ has function symbols s_1, \dots, s_m and primed function symbols u'_1, \dots, u'_l , there is a schema*

$$\text{com}_a = [\Delta(u_1, \dots, u_l); x_1 : S_1; \dots; x_n : S_n \mid \varphi]$$

² Operations in COD carry a delta list $\Delta(s_1, \dots, s_n)$ of symbols that can be changed.

in *cod* and the function symbols are declared in *cod*. The variables x_1 to x_n are the message variables of channel a used for communications.

The last condition also implies that for events $a \bullet \varphi_1$ and $a \bullet \varphi_2$, φ_1 and φ_2 are always equal, because different definitions of com_a are not allowed in COD.

Theorem 3 (Proof rule: Matching implies refinement). *Let P be a CSP process and let cod a COD specification such that cod matches P . Then, $P \sqsubseteq cod$.*

Proof. We give the proof idea here. The COD specification *cod* and the process P have the same semantical domain, timed interpretations. State changes are executed by constraints alongside the occurrence of events in both cases. By construction of the refinement rule, all constraints (and by this all state changes) coincide and hence we only need to prove that all traces of events performed by *cod*, which are traces of its `main` process, are also valid traces of the process P . We show this by proving that P down-simulates the process of *cod*, that is, there is a relation \preceq on processes s.t. for all R, S, \bar{S}

$$R \preceq S \text{ and } S \xrightarrow{a} \bar{S} \text{ implies that there is an } \bar{R} \text{ with } R \xrightarrow{a} \bar{R} \text{ and } \bar{R} \preceq \bar{S}.$$

It is then a standard result [13] that we can conclude from $P \preceq \text{main}$ that `main` is a refinement of P and by this $P \sqsubseteq cod$. \square

The notion of refinement introduced here is rather restricted because the symbols and the constraints of COD specification and VA process have to coincide. It is straightforward to extend the definitions as well as the refinement rule to arbitrary refinement relations on the symbols of COD specification and VA process by which more sophisticated connections are possible. E.g., an abstract data type like a list over arbitrary objects can be mapped by the refinement relation to a concrete list of integer values.

Example 3. In our running example, we proved the correctness of an instantiation of the VA from Fig. 11. This instantiation was given as a concrete COD model [9], for that direct verification was not possible (timeout after 80h) due to its complexity with 17 real-valued variables and clocks, over 300 program locations, and 17000 transitions. Since the model is a refinement of the VA, which can be syntactically checked, we only needed to verify the local DC formulae F_{FAR} and F_{REC} to conclude the safety of the entire system (cf. Sect. 1.2). This was done automatically with ARMC in 7h (F_{FAR}) and 4m (F_{REC}), respectively.

5 Conclusion

The main theme in this work was to uniformly integrate verification techniques in a formal framework to allow compositional verification of real-time systems: VAs combine CSP with data and additional real-time constraints to define behavioural protocols for classes of systems. Our new sequent-style proof calculus allows us to verify VAs by a combination of proof rule based reasoning and a

suitable verification technique for timed constraints. As a proof of concept, we considered instantiations of VAs by COD specifications and gave a syntactic proof rule to establish refinement relations. We were able to verify a COD-specified train control system that is too complex to be verified without further decomposition techniques. However, the basic ideas of our VA approach carry over to other formalisms. Particularly, our new CSP dialect is not bound to COD but can be used with other CSP-based combined formalisms, e.g., [32,29], for which syntactic refinement rules can be defined similarly to the rule from Sect. 4.

Even though we do not have tool support for the proof calculus from Sect. 3, the VA approach is dedicated to automated verification and there are promising results in automated verification for similar calculi [23].

A question that we have not investigated in this paper is the completeness of the VA approach: can appropriate assumptions be found for every possible instantiation of an architecture? The answer depends on the languages used for the assumptions and the instantiations. Present results for COD and DC suggest that this is actually the case because every COD instantiation can be translated into an equivalent DC expression.

Additionally, we have first achievements in extending the logic and the calculus to reason about more complex real-time properties like DC traces.

Acknowledgements. The author thanks Ernst-Rüdiger Olderog and Anders P. Ravn for helpful comments.

References

1. Abrial, J.R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
2. Butler, M.J.: A CSP Approach To Action Systems. Ph.D. thesis, University of Oxford (1992)
3. Damm, W., Hungar, H., Olderog, E.R.: Verification of cooperating traffic agents. *Int. J. Control.* 79(5), 395–421 (2006)
4. de Roever, W.P., et al.: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, Cambridge (2001)
5. D’Errico, L., Loreti, M.: Assume-Guarantee Verification of Concurrent Systems. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 288–305. Springer, Heidelberg (2009)
6. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed patterns: TCOZ to timed automata. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 483–498. Springer, Heidelberg (2004)
7. ERTMS User Group, UNISIG: ERTMS/ETCS System requirements specification (2002), <http://www.aeif.org/ccm/default.asp> (version 2.2.2)
8. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 233–252. Springer, Heidelberg (2007)
9. Faber, J.: Verification Architectures: Compositional reasoning for real-time systems. Reports of SFB/TR 14 AVACS 65 (2010), <http://www.avacs.org>
10. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. Ph.D. thesis, University of Oldenburg (2000)

11. Gentzen, G.: Untersuchungen über das logisches Schließen. *Mathematische Zeitschrift* 1, 176–210 (1935)
12. Harel, D., Kozen, D., Tiuryn, J.: *Dynamic Logic*. MIT Press, Cambridge (2000)
13. He, J.: Process simulation and refinement. *Form. Asp. Comput.* 1(3), 229–241 (1989)
14. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice Hall International, Englewood Cliffs (1985)
15. Hoenicke, J.: *Combination of Processes, Data and Time*. Ph.D. thesis, University of Oldenburg (2006)
16. Klebanov, V., Rümmer, P., Schlager, S., Schmitt, P.H.: Verification of JCSP programs. In: Broenink, J.F., Roeblers, H.W., Sunter, J.P.E., Welch, P.H., Wood, D.C. (eds.) *CPA. CSES*, vol. 63, pp. 203–218. IOS Press, Amsterdam (2005)
17. Knudsen, J., Ravn, A.P., Skou, A.: Design verification patterns. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) *Formal Methods and Hybrid Real-Time Systems*. LNCS, vol. 4700, pp. 399–413. Springer, Heidelberg (2007)
18. Larsen, K.G., Xinxin, L.: Compositionality through an operational semantics of contexts. *J. Log. Comput.* 1(6), 761–795 (1991)
19. Mahony, B.P., Dong, J.S.: Blending object-Z and timed CSP: An introduction to TCOZ. In: *ICSE*, pp. 95–104 (1998)
20. Metzler, B., Wehrheim, H., Wonisch, D.: Decomposition for compositional verification. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) *ICFEM 2008*. LNCS, vol. 5256, pp. 105–125. Springer, Heidelberg (2008)
21. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. *Form. Asp. Comput.* 20(4-5), 481–505 (2008)
22. Platzer, A.: A temporal dynamic logic for verifying hybrid system invariants. In: Artemov, S., Nerode, A. (eds.) *LFCS 2007*. LNCS, vol. 4514, pp. 457–471. Springer, Heidelberg (2007)
23. Platzer, A., Quesel, J.D.: Logical verification and systematic parametric analysis in train control. In: Egerstedt, M., Mishra, B. (eds.) *HSCC 2008*. LNCS, vol. 4981, pp. 646–649. Springer, Heidelberg (2008)
24. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) *PADL 2007*. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
25. RAISE Language Group: *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall International, Englewood Cliffs (1992)
26. Roscoe, A.: *Theory and Practice of Concurrency*. Prentice Hall International, Englewood Cliffs (1998)
27. Smith, G.: An integration of real-time object-Z and CSP for specifying concurrent real-time systems. In: Butler, M.J., Petre, L., Sere, K. (eds.) *IFM 2002*. LNCS, vol. 2335, pp. 267–285. Springer, Heidelberg (2002)
28. Stühl, C.: An overview of the integrated formalism RT-Z. *Form. Asp. Comput.* 13(2), 94–110 (2002)
29. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: Introducing a process analysis toolkit. In: *ISoLA 2008*. CCIS, vol. 17, pp. 307–322. Springer, Heidelberg (2008)
30. Taibi, T.: *Design Pattern Formalization Techniques*. IGI Publishing (2007)
31. Wehrheim, H.: *Behavioural subtyping in object-oriented specification formalisms*. University of Oldenburg, Habilitation (2002)
32. Woodcock, J.C.P., Cavalcanti, A.L.C.: A concurrent language for refinement. In: Butterfield, A., Pahl, C. (eds.) *IWFM 2001*. BCS Elec. Works. in Computing (2001)
33. Zhou, C., Hansen, M.R.: *Duration Calculus*. Springer, Heidelberg (2004)

Automatic Verification of Parametric Specifications with Complex Topologies^{*}

Johannes Faber¹, Carsten Ihlemann²,
Swen Jacobs³, and Viorica Sofronie-Stokkermans²

¹ Department of Computing Science, University of Oldenburg, Germany

² Max-Planck-Institut für Informatik, Saarbrücken, Germany

³ École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. The focus of this paper is on reducing the complexity in verification by exploiting modularity at various levels: in specification, in verification, and structurally. For specifications, we use the modular language CSP-OZ-DC, which allows us to decouple verification tasks concerning data from those concerning durations. At the verification level, we exploit modularity in theorem proving for rich data structures and use this for invariant checking. At the structural level, we analyze possibilities for modular verification of systems consisting of various components which interact. We illustrate these ideas by automatically verifying safety properties of a case study from the European Train Control System standard, which extends previous examples by comprising a complex track topology with lists of track segments and trains with different routes.

1 Introduction

Parametric real-time systems arise in a natural way in a wide range of applications, including controllers for systems of cars, trains, and planes. Since many such systems are safety-critical, there is great interest in methods for ensuring that they are safe. In order to verify such systems, one needs (i) suitable formalizations and (ii) efficient verification techniques. In this paper, we analyze both aspects. Our main focus throughout the paper will be on reducing complexity by exploiting modularity at various levels: in the specification, in verification, and also structurally. The main contributions of the paper are:

- (1) We exploit modularity at the specification level. In Sect. 2, we use the modular language CSP-OZ-DC (COD), which allows us to separately specify processes (as Communicating Sequential Processes, CSP), data (using Object-Z, OZ) and time (using the Duration Calculus, DC).
- (2) We exploit modularity in verification (Sect. 3).
 - First, we consider transition constraint systems (TCSs) that can be automatically obtained from the COD specification, and address verification tasks such as invariant checking. We show that for pointer data structures, we can obtain decision procedures for these verification tasks.

^{*} This work was partly supported by the German Research Council (DFG) under grant SFB/TR 14 AVACS. See <http://www.avacs.org> for more information.

- Then we analyze situations in which the use of COD specifications allows us to decouple verification tasks concerning data (OZ) from verification tasks concerning durations (DC). For systems with a parametric number of components, this allows us to impose (and verify) conditions on the single components which guarantee safety of the overall complex system.
- (3) We also use modularity at a structural level. In Sect. 4, we use results from [24] to obtain possibilities for modular verification of systems with complex topologies by decomposing them into subsystems with simpler topologies.
 - (4) We describe a tool chain which translates a graphical UML version of the CSP-OZ-DC specification into TCSs, and automatically verifies the specification using our prover H-PILoT and other existing tools (Sect. 5).
 - (5) We illustrate the ideas on a running example taken from the European Train Control System standard (a system with a complex topology and a parametric number of components—modeled using pointer data structures and parametric constraints), and present a way of fully automatizing verification (for given safety invariants) using our tool chain.

Related work. Model-based development and verification of railway control systems with a complex track topology are analyzed in [10]. The systems are described in a domain-specific language and translated into SystemC code that is verified using bounded model checking. Neither verification of systems with a parametric number of components nor pointer structures are examined there.

In existing work on the verification of parametric systems often only few aspects of parametricity are studied together. [21] addresses the verification of temporal properties for hybrid systems (in particular also fragments of the ETCS as case study) but only supports parametricity in the data domain. [2] presents a method for the verification of a parametric number of timed automata with real-valued clocks, while in [5] only finite-state processes are considered. In [3], regular model checking for a parametric number of homogeneous linear processes and systems operating on queues or stacks is presented. There is also work on the analysis of safety properties for parametrized systems with an arbitrary number of processes operating on unbounded integer variables [17][16]. In contrast to ours, these methods sacrifice completeness by using either an over-approximation of the transition relation or abstractions of the state space. We, on the other hand, offer complete methods (based on decision procedures for data structures) for problems such as invariant checking and bounded model checking.

Motivating example. Consider a system of trains on a complex track topology as depicted in Fig. 1, and a radio block center (RBC) that has information about track segments and trains, like e.g. length, occupying train and allowed maximal speed for segments, and current position, segment and speed for trains. We will show under which situations safety of the system with complex track topology is a consequence of safety of systems with linear track topology. Such modular verification possibilities allow us to consider the verification of a simplified version of this example, consisting of a *linear track* (representing a concrete route in the track topology), on which trains are allowed to enter or leave at given points. We model a general RBC controller for an area with a linear track topology and an

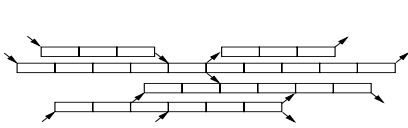


Fig. 1. Complex Track Topology

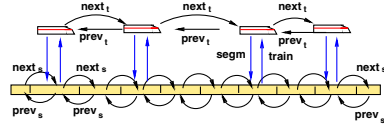


Fig. 2. Linear Track Topology

arbitrary number of trains. For this, we use a theory of pointers with sorts t (for trains; $next_t$ returns the next train on the track) and s (for segments; with $next_s$, $prev_s$ describing the next/previous segment on the linear track). The link between trains and segments is described by appropriate functions `train` and `segm` (cf. Fig. 2). In addition, we integrated a simple timed train controller `Train` into the model. This allowed us to certify that certain preconditions for the verification of the RBC are met by every train which satisfies the specification of `Train`, by reasoning on the timed and the untimed part of the system independently.

2 Modular Specifications: CSP-OZ-DC

We start by presenting the specification language CSP-OZ-DC (COD) [12,11] which allows us to present in a modular way the control flow, data changes, and timing aspects of the systems we want to verify. We use *Communicating Sequential Processes* (CSP) to specify the control flow of a system using processes over events; *Object-Z* (OZ) for describing the state space and its change, and the *Duration Calculus* (DC) for modeling (dense) real-time constraints over durations of events. The operational semantics of COD is defined in [11] in terms of a timed automata model. For details on CSP-OZ-DC and its semantics, we refer to [12,11,9]. Our benefits from using COD are twofold:

- COD is compositional in the sense that it suffices to prove safety properties for the separate components to prove safety of the entire system [11]. This makes it possible to use different verification techniques for different parts of the specification, e.g. for control structure and timing properties.
- We benefit from high-level tool support given by Syspect [4], a UML editor for a dedicated UML profile [20] proposed to formally model real-time systems. It has a semantics in terms of COD. Thus, Syspect serves as an easy-to-use front-end to formal real-time specifications, with a graphical user interface.

2.1 Example: Systems of Trains on Linear Tracks

To illustrate the ideas, we present some aspects of the case study mentioned in Sect. 1 (the full case study is presented in [8]). We exploit the benefits of COD in (i) the specification of a complex RBC controller; (ii) the specification of a controller for individual trains; and (iii) composing such specifications. Even though space does not allow us to present all details, we present aspects of the

¹ <http://csd.informatik.uni-oldenburg.de/~syspect/>

example which cannot be considered with other formalisms, and show how to cope in a natural way with parametricity.

CSP part. The processes and their interdependency is specified using the CSP specification language. The RBC system passes repeatedly through four phases, modeled by events with corresponding COD schemata *updSpd* (*speed update*), *req* (*request update*), *alloc* (*allocation update*), and *updPos* (*position update*).

CSP:

```

method enter : [s1? : Segment; t0? : Train; t1? : Train; t2? : Train]
method leave  : [ls? : Segment; lt? : Train]
local_chan alloc, req, updPos, updSpd

main ≜ ((updSpd → State1) State1 ≜ ((req → State2) State2 ≜ ((alloc → State3) State3 ≜ ((updPos → main)
□(leave → main) □(leave → State1) □(leave → State2) □(leave → State3)
□(enter → main) □(enter → State1) □(enter → State2) □(enter → State3))

```

The *speed update* models the fact that every train chooses its speed according to its knowledge about itself and its track segment as well as the next track segment. The *request update* models how trains send a request for permission to enter the next segment when they come close to the end of their current segment. The *allocation update* models how the RBC may either grant these requests by allocating track segments to trains that have made a request, or allocate segments to trains that are not currently on the route and want to enter. The *position update* models how all trains report their current positions to the RBC, which in turn de-allocates segments that have been left and gives movement authorities to the trains. Between any of these four updates, we can have trains *leaving* or *entering* the track at specific segments using the events *leave* and *enter*. The effects of these updates are defined in the OZ part.

OZ part. The OZ part of the specification consists of data classes, axioms, the *Init* schema, and update rules.

Data classes. The data classes declare function symbols that can change their values during runs of the system, and are used in the OZ part of the specification.

<i>SegmentData</i>	<i>TrainData</i>
<i>train</i> : <i>Segment</i> → <i>Train</i>	<i>segm</i> : <i>Train</i> → <i>Segment</i>
<i>req</i> : <i>Segment</i> → \mathbb{Z}	<i>next</i> : <i>Train</i> → <i>Train</i>
<i>alloc</i> : <i>Segment</i> → \mathbb{Z}	<i>spd</i> : <i>Train</i> → \mathbb{R}
	<i>pos</i> : <i>Train</i> → \mathbb{R}
	<i>prev</i> : <i>Train</i> → <i>Train</i>

Axioms. The axiomatic part defines properties of the data structures and system parameters which do not change during an execution of the system: *gmax* : \mathbb{R} (the global maximum speed), *decmax* : \mathbb{R} (the maximum deceleration of trains), *d* : \mathbb{R} (a safety distance between trains), and *bd* : $\mathbb{R} \rightarrow \mathbb{R}$ (mapping the speed of a train to a safe approximation of the corresponding braking distance). We specify properties of those parameters, among which an important one is $d \geq bd(gmax) + gmax \cdot \Delta t$ stating that the safety distance *d* to the end of the segment is greater than the braking distance of a train at maximal speed *gmax* plus a further safety margin (distance for driving Δt time units at speed *gmax*). Furthermore, unique, non-negative ids for trains (sort *Train*) and track segments (sort *Segment*) are defined. The route is modeled as a doubly-linked

list² of track segments, where every segment has additional properties specified by the constraints in the state schema.

E.g., *sid* is increasing along the *nexts* pointer, the *length* of a segment is bounded from below in terms of d and $gmax \cdot \Delta t$, and the difference between local maximal speeds on neighboring segments is bounded by $decmax \cdot \Delta t$. Finally, we have a function *incoming*, the value of which is either a train which wants to enter the given segment from outside the current route, or *tnil* if there is no such train. Although the valuation of *incoming* can change during an execution, we consider the constraint (*) as a property of our environment that always holds. Apart from that, incoming may change arbitrarily and is not explicitly updated. Note that *Train* and *Segment* are pointer sorts with a special null element (*tnil* and *snil*, respectively), and all constraints implicitly only hold for non-null elements. So, constraint (*) actually means

$$\forall s1, s2 : Segment \mid s1 \neq snil \neq s2 \wedge incoming(s1) \neq tnil \wedge train(s2) \neq tnil \\ \bullet tid(incoming(s1)) \neq tid(train(s2))$$

Init schema. The *init schema* describes the initial state of the system. It essentially states that trains are arranged in a doubly-linked list, that all trains are initially placed correctly on the track segments and that all trains respect their speed limits.

$$\begin{array}{l} \text{Init.} \\ \hline \forall t : Train \bullet train(seg(t)) = t \\ \forall t : Train \bullet next(prev(t)) = t \\ \forall t : Train \bullet prev(next(t)) = t \\ \forall t : Train \bullet 0 \leq pos(t) \leq length(seg(t)) \\ \forall t : Train \bullet 0 \leq spd(t) \leq lmax(seg(t)) \\ \forall t : Train \bullet alloc(seg(t)) = tid(t) \\ \forall t : Train \bullet alloc(nexts(seg(t))) = tid(t) \\ \quad \vee length(seg(t)) - bd(spd(t)) > pos(t) \\ \forall s : Segment \bullet seg(train(s)) = s \end{array}$$

Update rules. Updates of the state space, that are executed when the corresponding event from the CSP part is performed, are specified with *effect schemata*. The schema for *updSpd*, for instance, consists of three rules, distinguishing (i) trains whose distance to the end of the segment is greater than the safety distance d (the first two lines of the constraint), (ii) trains that are beyond the safety distance near the end of the segment, and for which the next segment is allocated, and (iii) trains that are near the end of the segment without an allocation. In case (i), the train can choose an arbitrary speed below the maximal speed of the current segment. In case (ii), the train needs to brake if the speed limit of the next segment is below the current limit. In case (iii), the train needs to brake such that it safely stops before reaching the end of the segment.

$$\begin{array}{l} \text{effect_updSpd} \\ \hline \Delta(spd) \\ \hline \forall t : Train \mid pos(t) < length(seg(t)) - d \wedge spd(t) - decmax \cdot \Delta t > 0 \\ \quad \bullet max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq lmax(seg(t)) \\ \forall t : Train \mid pos(t) \geq length(seg(t)) - d \wedge alloc(nexts(seg(t))) = tid(t) \\ \quad \bullet max\{0, spd(t) - decmax \cdot \Delta t\} \leq spd'(t) \leq \min\{lmax(seg(t)), lmax(nexts(seg(t)))\} \\ \forall t : Train \mid pos(t) \geq length(seg(t)) - d \wedge \neg alloc(nexts(seg(t))) = tid(t) \\ \quad \bullet spd'(t) = max\{0, spd(t) - decmax \cdot \Delta t\} \end{array}$$

² Note that we use relatively loose axiomatizations of the list structures for both trains and segments, also allowing for disjoint families of linear, possibly infinite lists.

Timed train controller.

In the DC part of a specification, real-time constraints are specified: A second, timed controller **Train** (for one train only) interacts with the RBC controller, which is presented

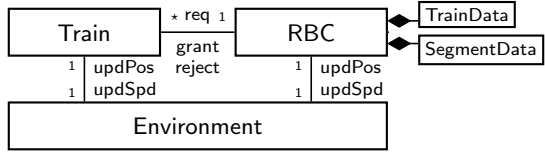


Fig. 3. Structural overview

in the overview of the case study in Fig. 3. The train controller **Train** consists of three timed components running in parallel. The first updates the train’s position. This component contains e.g. the DC formula

$$\neg(true \wedge \uparrow updPos \wedge (\ell < \Delta t) \wedge \downarrow updPos \wedge true),$$

that specifies a lower time bound Δt on $updPos$ events. The second component checks periodically whether the train is beyond the safety distance to the end of the segment. Then, it starts braking within a short reaction time. The third component requests an extension of the movement authority from the RBC, which may be granted or rejected. The full train controller can be found in [8].

3 Modular Verification

In this section, we combine two approaches for the verification of safety properties of COD specifications:

- We introduce the invariant checking approach and present decidability results for local theory extensions that imply decidability of the invariant checking problem for a large class of parameterized systems.
- We illustrate how we can combine this invariant checking for the RBC specification with a method for model checking of real-time properties (introduced in [19]) for the COD specification for a single train **Train**.

Formally, our approach works on a *transition constraint system* (TCS) obtained from the COD specification by an automatic translation (see [9]) which is guaranteed to capture the defined semantics of COD (as defined in [11]).

Definition 1. *The tuple $T = (V, \Sigma, (\text{Init}), (\text{Update}))$ is a transition constraint system, which specifies: the variables (V) and function symbols (Σ) whose values may change over time; a formula (Init) specifying the properties of initial states; and a formula (Update) which specifies the transition relation in terms of the values of variables $x \in V$ and function symbols $f \in \Sigma$ before a transition and their values (denoted x', f') after the transition.*

In addition to the TCS, we obtain a *background theory* \mathcal{T} from the specification, describing properties of the used data structures and system parameters that do not change over time. Typically, \mathcal{T} consists of a family of standard theories (like the theory of real numbers), axiomatizations for data structures, and constraints on system parameters. In what follows $\phi \models_{\mathcal{T}} \psi$ denotes logical entailment and means that every model of the theory \mathcal{T} which is a model of ϕ is also a model for ψ . We denote false by \perp , so $\phi \models_{\mathcal{T}} \perp$ means that ϕ is unsatisfiable w.r.t. \mathcal{T} .

3.1 Verification Problems

We consider the problem of *invariant checking* of safety properties³. To show that a safety property, represented as a formula (**Safe**), is an invariant of a TCS T (for a given background theory \mathcal{T}), we need to identify an *inductive invariant* (**Inv**) which strengthens (**Safe**), i.e., we need to prove that

- (1) $(\text{Inv}) \models_{\mathcal{T}} (\text{Safe})$,
- (2) $(\text{Init}) \models_{\mathcal{T}} (\text{Inv})$, and
- (3) $(\text{Inv}) \wedge (\text{Update}) \models_{\mathcal{T}} (\text{Inv}')$, where (Inv') results from (Inv) by replacing each $x \in V$ by x' and each $f \in \Sigma$ by f' .

Lemma 2. *If (Inv) , (Init) and (Update) belong to a class of formulae for which the entailment problems w.r.t. \mathcal{T} above are decidable then the problem of checking that (Inv) is an invariant of T (resp. T satisfies the property (Safe)) is decidable.*

We use this result in a verification-design loop as follows: We start from a specification written in COD. We use a translation to TCS and check whether a certain formula (**Inv**) (usually a safety property) is an inductive invariant.

- (i) If invariance can be proved, safety of the system is guaranteed.
- (ii) If invariance cannot be proved, we have the following possibilities:
 1. Use a specialized prover to construct a counterexample (model in which the property (**Inv**) is not an invariant) which can be used to find errors in the specification and/or to strengthen the invariant⁴.
 2. Using results in [25] we can often derive additional (weakest) constraints on the parameters which guarantee that **Inv** is an invariant.

Of course, the decidability results for the theories used in the description of a system can be also used for checking consistency of the specification.

If a TCS models a system with a parametric number of components, the formulae in problems (1)–(3) may contain universal quantifiers (to describe properties of all components), hence standard SMT methods – which are only complete for ground formulae – do not yield decision procedures. In particular, for (ii)(1,2) and for consistency checks we need possibilities of reliably detecting satisfiability of sets of universally quantified formulae for which standard SMT solvers cannot be used. We now present situations in which this is possible.

3.2 Modularity in Automated Reasoning: Decision Procedures

We identify classes of theories for which invariant checking (and bounded model checking) is decidable. Let \mathcal{T}_0 be a theory with signature $\Pi = (S_0, \Sigma_0, \text{Pred})$, where S_0 is a set of sorts, and Σ_0 and Pred are sets of function resp. predicate symbols. We consider extensions of \mathcal{T}_0 with new function symbols in a set Σ , whose properties are axiomatized by a set \mathcal{K} of clauses.

³ We can address bounded model checking problems in a similar way, cf. [15,9,13].

⁴ This last step is the only part which is not fully automatized. For future work we plan to investigate possibilities of automated invariant generation or strengthening.

Local theory extensions. We are interested in theory extensions in which for every set G of ground clauses we can effectively determine a finite (preferably small) set of instances of the axioms \mathcal{K} sufficient for checking satisfiability of G without loss of completeness. If G is a set of Π^c -clauses (where Π^c is the extension of Π with constants in a set Σ_c), we denote by $\text{st}(\mathcal{K}, G)$ the set of ground terms starting with a Σ -function symbol occurring in \mathcal{K} or G , and by $\mathcal{K}[G]$ the set of instances of \mathcal{K} in which the terms starting with Σ -functions are in $\text{st}(\mathcal{K}, G)$. $\mathcal{T}_0 \cup \mathcal{K}$ is a *local extension* of \mathcal{T}_0 [23] if the following condition holds:

(Loc) For every set G of ground clauses, $G \models_{\mathcal{T}_0 \cup \mathcal{K}} \perp$ iff $\mathcal{K}[G] \cup G \models_{\mathcal{T}_0^\Sigma} \perp$

where \mathcal{T}_0^Σ is the extension of \mathcal{T}_0 with the free functions in Σ . We can define *stable locality* (SLoc) in which we use the set $\mathcal{K}^{[G]}$ of instances of \mathcal{K} in which the variables below Σ -functions are instantiated with terms in $\text{st}(\mathcal{K}, G)$. In local theory extensions, sound and complete hierarchical reasoning is possible.

Theorem 3 ([23]). *With the notations introduced above, if $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition ((S)Loc) then the following are equivalent to $G \models_{\mathcal{T}_0 \cup \mathcal{K}} \perp$:*

- (1) $\mathcal{K}^*[G] \cup G \models_{\mathcal{T}_0^\Sigma} \perp$ ($\mathcal{K}^*[G]$ is $\mathcal{K}[G]$ for local; $\mathcal{K}^{[G]}$ for stably local extensions).
- (2) $\mathcal{K}_0 \cup G_0 \cup D \models_{\mathcal{T}_0^\Sigma} \perp$, where $\mathcal{K}_0 \cup G_0 \cup D$ is obtained from $\mathcal{K}^*[G] \cup G$ by introducing (bottom-up) new constants c_t for subterms $t = f(g_1, \dots, g_n)$ with $f \in \Sigma$, g_i ground $\Sigma_0 \cup \Sigma_c$ -terms; replacing the terms with the corresponding constants; and adding the definitions $c_t \approx t$ to the set D .
- (3) $\mathcal{K}_0 \cup G_0 \cup N_0 \models_{\mathcal{T}_0} \perp$, where

$$N_0 = \left\{ \bigwedge_{i=1}^n c_i \approx d_i \rightarrow c = d \mid f(c_1, \dots, c_n) \approx c, f(d_1, \dots, d_n) \approx d \in D \right\}.$$

The hierarchical reduction method is implemented in the system H-PILoT [14].

Corollary 4 ([23]). *If the theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_1$ satisfies ((S)Loc) then satisfiability of sets of ground clauses G w.r.t. \mathcal{T}_1 is decidable if $\mathcal{K}^*[G]$ is finite and $\mathcal{K}_0 \cup G_0 \cup N_0$ belongs to a decidable fragment \mathcal{F} of \mathcal{T}_0 . Since the size of $\mathcal{K}_0 \cup G_0 \cup N_0$ is polynomial in the size of G (for a given \mathcal{K}), locality allows us to express the complexity of the ground satisfiability problem w.r.t. \mathcal{T}_1 as a function of the complexity of the satisfiability of \mathcal{F} -formulae w.r.t. \mathcal{T}_0 .*

3.3 Examples of Local Theory Extensions

We are interested in reasoning efficiently about data structures and about updates of data structures. We here give examples of such theories.

Update axioms. In [13] we show that update rules $\text{Update}(\Sigma, \Sigma')$ which describe how the values of the Σ -functions change, depending on a set $\{\phi_i \mid i \in I\}$ of mutually exclusive conditions, define local theory extensions.

Theorem 5 ([13]). *Assume that $\{\phi_i \mid i \in I\}$ are formulae over the base signature such that $\phi_i(\bar{x}) \wedge \phi_j(\bar{x}) \models_{\mathcal{T}_0} \perp$ for $i \neq j$, and that s_i, t_i are (possibly equal)*

terms over the signature Σ such that $\mathcal{T}_0 \models \forall \bar{x}(\phi_i(\bar{x}) \rightarrow s_i(\bar{x}) \leq t_i(\bar{x}))$ for all $i \in I$. Then the extension of \mathcal{T}_0 with axioms of the form $\text{Def}(f)$ is local.

$$\text{Def}(f) \quad \forall \bar{x}(\phi_i(\bar{x}) \rightarrow s_i(\bar{x}) \leq f(\bar{x}) \leq t_i(\bar{x})) \quad i \in I.$$

Data structures. Numerous locality results for data structures exist, e.g. for fragments of the theories of arrays [6,13], and pointers [18,13]. As an illustration – since the model we used in the running example involves a theory of linked data structures – we now present a slight extension of the fragment of the theory of pointers studied in [18,13], which is useful for modeling the track topologies and successions of trains on these tracks. We consider a set of pointer sorts $P = \{p_1, \dots, p_n\}$ and a scalar sort s .⁵ Let $(\Sigma_s, \text{Pred}_s)$ be a scalar signature, and let Σ_P be a set of function symbols with arguments of pointer sort consisting of sets $\Sigma_{\bar{p} \rightarrow s}$ (the family of functions of arity $\bar{p} \rightarrow s$), and $\Sigma_{\bar{p} \rightarrow p}$ (the family of functions of arity $\bar{p} \rightarrow p_i$). (Here \bar{p} is a tuple $p_{i_1} \dots p_{i_k}$ with $k \geq 0$.) We assume that for every pointer sort $p \in P$, Σ_P contains a constant null_p of sort p .

Example 6. The fact that we also allow scalar fields with more than one argument is very useful because it allows, for instance, to model certain relationships between different nodes. Examples of such scalar fields could be:

- $\text{distance}(p, q)$ associates with non-null p, q of pointer type a real number;
- $\text{reachable}(p, q)$ associates with non-null p, q of pointer type a boolean value (true (1) if q is reachable from p using the next functions, false (0) otherwise).

Let $\Sigma = \Sigma_P \cup \Sigma_s$. In addition to allowing several pointer types and functions of arbitrary arity, we loosen some of the restrictions imposed in [18,13].

Definition 7. An extended pointer clause is a formula of form $\forall \bar{p}. (\mathcal{E} \vee \varphi)(\bar{p})$, where \bar{p} is a set of pointer variables including all free variables of \mathcal{E} and φ , and:

- (1) \mathcal{E} consists of disjunctions of pointer equalities, and has the property that for every term $t = f(t_1, \dots, t_k)$ with $f \in \Sigma_P$ occurring in $\mathcal{E} \vee \varphi$, \mathcal{E} contains an atom of the form $t' = \text{null}_p$ for every proper subterm (of sort p) t' of t ;
- (2) φ is an arbitrary formula of sort s .

\mathcal{E} and φ may additionally contain free scalar and pointer constants, and φ may contain additional quantified variables of sort s .

Theorem 8. Let $\Sigma = \Sigma_P \cup \Sigma_s$ be a signature as defined before. Let \mathcal{T}_s be a theory of scalars with signature Σ_s . Let Φ be a set of Σ -extended pointer clauses. Then, for every set G of ground clauses over an extension Σ^c of Σ with constants in a countable set c the following are equivalent:

- (1) G is unsatisfiable w.r.t. $\Phi \cup \mathcal{T}_s$;
- (2) $\Phi^{[G]} \cup G$ is an unsatisfiable set of clauses in the disjoint combination $\mathcal{T}_s \cup \mathcal{EQ}_P$ of \mathcal{T}_s and \mathcal{EQ}_P , the many-sorted theory of pure equality over pointer sorts,

⁵ We assume that we only have one scalar sort for simplicity of presentation; the scalar theory can itself be an extension or combination of theories.

where $\Phi^{[G]}$ consists of all instances of Φ in which the universally quantified variables of pointer type occurring in Φ are replaced by ground terms of pointer type in the set $\text{st}(\Phi, G)$ of all ground terms of sort p occurring in Φ or in G .

The proof is similar to that in [13]. H-PILoT can be used as a decision procedure for this theory of pointers – if the theory of scalars is decidable – and for any extension of this theory with function updates in the fragment in Thm. 5.

Example 9. Let $P = \{\text{sg}(\text{segment}), \text{t}(\text{train})\}$, and let $\text{next}_t, \text{prev}_t : \mathbf{t} \rightarrow \mathbf{t}$, and $\text{next}_s, \text{prev}_s : \mathbf{sg} \rightarrow \mathbf{sg}$, and $\text{train} : \mathbf{sg} \rightarrow \mathbf{t}$, $\text{segm} : \mathbf{t} \rightarrow \mathbf{sg}$, and functions of scalar sort as listed at the beginning of Sect. 2.1. All axioms describing the background theory and the initial state in Sect. 2.1 are expressed by extended pointer clauses. The following formula expressing a property of reachability of trains can be expressed as a pointer clause:

$$\forall p, q (p \neq \text{null}_t \wedge q \neq \text{null}_t \wedge \text{next}_t(q) \neq \text{null}_t \rightarrow (\text{reachable}(p, q) \rightarrow \text{reachable}(p, \text{next}_t(q))).$$

Decidability for verification. A direct consequence of Thm. 3 and Cor. 4 is the following decidability result for invariant checking:

Corollary 10 ([25]). *Let T be the transition constraint system and \mathcal{T} be the background theory associated with a specification. If the update rules **Update** and the invariant property **Inv** can be expressed as sets of clauses which define a chain of local theory extensions $\mathcal{T} \subseteq \mathcal{T} \cup \text{Inv}(\bar{x}, \bar{f}) \subseteq \mathcal{T} \cup \text{Inv}(\bar{x}, \bar{f}) \cup \text{Update}(\bar{x}, \bar{x}', \bar{f}, \bar{f}')$ then checking whether a formula is an invariant is decidable.*

In this case we can use H-PILoT as a decision procedure (and also to construct a model in which the property **Inv** is not an invariant). We can also use results from [25] to derive additional (weakest) constraints on the parameters which guarantee that **Inv** is an invariant.

3.4 Example: Verification of the Case Study

We demonstrate how the example from Sect. 1 can be verified by using a combination of the invariant checking approach presented in Sect. 3.1 and a model checking approach for timing properties. This combination is necessary because the example contains both the RBC component with its discrete updates, and the train controller **Train** with real-time safety properties. Among other things, the specification of the RBC assumes that the train controllers always react in time to make the train brake before reaching a critical position.

Using the modularity of COD, we can separately use the invariant checking approach to verify the RBC for a parametric number of trains, and the approach for model checking DC formulae to verify that every train satisfies the timing assumptions made in the RBC specification.

Verification of the RBC. The verification problems for the RBC are satisfiability problems containing universally quantified formulae, hence cannot be decided by standard methods of reasoning in combinations of theories. Instead, we use the hierarchical reasoning approach from Sect. 3.2.

Safety properties. As safety property for the RBC we want to prove that we never have two trains on the same segment:

$$(\text{Safe}) := \forall t_1, t_2 : \text{Train}. t_1 \neq t_2 \rightarrow \text{id}_s(\text{segm}(t_1)) \neq \text{id}_s(\text{segm}(t_2)).$$

To this end, we need to find a formula (Inv) such that we can prove

- (1) $(\text{Inv}) \cup \neg(\text{Safe}) \models_{\mathcal{T}} \perp$,
- (2) $(\text{Init}) \cup \neg(\text{Inv}) \models_{\mathcal{T}} \perp$, and
- (3) $(\text{Inv}) \cup (\text{Update}) \cup \neg(\text{Inv}') \models_{\mathcal{T}} \perp$,

where (Update) is the update formula associated with the transition relation obtained by translating the COD specification into TCS [11,9], and (Init) consists of the constraints in the Init schema. The background theory \mathcal{T} is obtained from the state schema of the OZ part of the specification: it is the combination of the theories of real numbers and integers, together with function and constant symbols satisfying the constraints given in the state schema.

Calling H-PILoT on problem (3) with $(\text{Inv}) = (\text{Safe})$ shows us that (Safe) is not inductive over all transitions. Since we expect the updates to preserve the well-formedness properties in (Init) , we tried to use this as our invariant, but with the same result. However, inspection of counterexamples provided by H-PILoT allowed us to identify the following additional constraints needed to make the invariant inductive:

$$\begin{aligned} (\text{Ind}_1) &:= \forall t : \text{Train}. pc \neq \text{InitState} \wedge \text{alloc}(\text{nexts}(\text{segm}(t))) \neq \text{tid}(t) \\ &\quad \rightarrow \text{length}(\text{segm}(t)) - \text{bd}(\text{spd}(t)) > \text{pos}(t) + \text{spd}(t) \cdot \Delta t \\ (\text{Ind}_2) &:= \forall t : \text{Train}. pc \neq \text{InitState} \wedge \text{pos}(t) \geq \text{length}(\text{segm}(t)) - d \\ &\quad \rightarrow \text{spd}(t) \leq \text{lmax}(\text{nexts}(\text{segm}(t))) \end{aligned}$$

The program counter pc is introduced in the translation process from COD to TCS and we use the constraint $pc \neq \text{InitState}$ to indicate that the system is not in its initial location. Thus, define (Inv) as the conjunction $(\text{Init}) \wedge (\text{Ind}_1) \wedge (\text{Ind}_2)$. Now, all of the verification tasks above can automatically be proved using Syspect and H-PILoT, in case (3) after splitting the problem into a number of sub-problems. To ensure that our system is not trivially safe because of inconsistent assumptions, we also check for consistency of \mathcal{T} , (Inv) and (Update) . Since by Thm. 5 all the update rules in the RBC specification define local theory extensions, and the axioms specifying properties of the data types are extended pointer clauses, by Cor. 10 we obtain the following decidability result.

Corollary 11. *Checking properties (1)–(3) is decidable for all formulae Inv expressed as sets of extended pointer clauses with the property that the scalar part belongs to a decidable fragment of the theory of scalars.*

Topological invariants. We also considered certain topological invariants of the system – e.g. that if a train t is inserted between trains t_1 and t_2 , the next and prev links are adjusted properly, and if a train leaves a track then its next_t and prev_t links become null. We also checked that if certain reachability conditions – modeled using a binary transitive function reachable with Boolean output which

is updated when trains enter or leave the line track – are satisfied before an insertion/removal of trains then they are satisfied also after. We cannot include these examples in detail here; they will be presented in a separate paper.

Verification of the timed train controller. Using the model checking approach from [19], we can automatically prove real-time properties of COD specifications. In this case, we use the approach only on the train controller part Train (Fig. 3). We show that the safety distance d and the braking distance bd postulated in the RBC controller model can actually be achieved by trains that comply with the train specification. That is, we prove that (for an arbitrary train) the train position $curPos$ is never beyond its movement authority ma :

$$(\text{Safe}_T) := \neg \diamond (curPos > ma).$$

Safety of the overall system. The safety property for trains (Safe_T) implies that train controllers satisfying the specification also satisfy the timing assumptions made implicitly in the RBC controller. Compositionality of COD guarantees [11] that it is sufficient to verify these components separately. Thus, by additionally proving that (Inv) is a safety invariant of the RBC, we have shown that the system consisting of a combination of the RBC controller and arbitrarily many train controllers is safe.

4 Modular Verification for Complex Track Topologies

We now consider a complex track as described in Fig. 1. Assume that the track can be modeled as a directed graph $G = (V, E)$ with the following properties:

- (i) The graph G is acyclic (the rail track does not contain cycles);
- (ii) The in-degree of every node is at most 2 (at every point at which two lines meet, at most two linear tracks are merged).

Theorem 12. *For every track topology satisfying conditions (i) and (ii) above we can find a decomposition $\mathcal{L} = \{\text{ltrack}_i \mid i \in I\}$ into linear tracks such that if $(x, y) \in E$ then $y = \text{next}_s^{\text{ltrack}_i}(x)$ for some $i \in I$ and for every $\text{ltrack} \in \mathcal{L}$ identifiers are increasing w.r.t. $\text{next}_s^{\text{ltrack}}$.*

We assume that for each linear track ltrack we have one controller RBC^{ltrack} which uses the control protocol described in Sect. 2.1, where we label the functions describing the train and segment succession using indices (e.g. we use $\text{next}_t^{\text{ltrack}}$, $\text{prev}_t^{\text{ltrack}}$ for the successor/predecessor of a train on ltrack , and $\text{next}_s^{\text{ltrack}}$, $\text{prev}_s^{\text{ltrack}}$ for the successor/predecessor of a segment on ltrack). Assume that these controllers are compatible on their common parts, i.e. (1) if two tracks $\text{track}_1, \text{track}_2$ have a common subtrack track_3 then the corresponding fields agree, i.e. whenever $s, \text{next}_s^{\text{track}_i}(s)$ are on track_3 , $\text{next}_s^{\text{track}_1}(s) = \text{next}_s^{\text{track}_2}(s) = \text{next}_s^{\text{track}_3}(s)$ (the same for prev_s , and for $\text{next}_t, \text{prev}_t$ on the corresponding tracks); (2) the update rules are compatible for trains jointly controlled [6]. Under these conditions, proving safety for the complex track can be reduced to checking safety of linear train tracks with incoming and outgoing trains (for details cf. [8]).

⁶ We also assume that all priorities of the trains on the complex track are different.

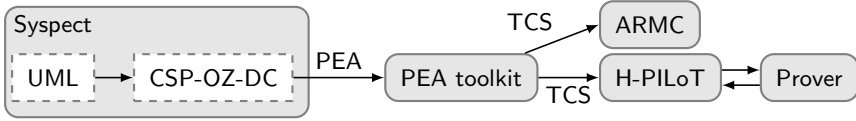


Fig. 4. Tool chain

Lemma 13. *A state s of the system is a model $(P_t, P_s, \mathbb{R}, \mathbb{Z}, \{\text{next}^{\text{ltrack}}, \text{prev}^{\text{ltrack}}, \text{next}_s^{\text{ltrack}}, \text{prev}_s^{\text{ltrack}}\}_{\text{ltrack} \in \mathcal{L}} \cup \{\text{segm}, \text{train}, \text{pos}, \dots\})$, where all the functions relativized to tracks are compatible on common subtracks. The following hold:*

- Every state s of the system of trains on the complex track restricts to a state s_{ltrack} of the system of trains on its component linear track.
- Any family $\{s_{\text{ltrack}_i} \mid i \in I\}$ of states on the component tracks which agree on the common sub-tracks can be “glued together” to a state s of the system of trains on the complex track topology.

(a) and (b) also hold if we consider initial states (i.e. states satisfying the initial conditions) and safe states (i.e. states satisfying the safety conditions in the invariant Inv). Similar properties hold for parallel actions and for transitions.

Theorem 14. *Consider a complex track topology satisfying conditions (i)–(ii) above. Let $\mathcal{L} = \{\text{ltrack}_i \mid i \in I\}$ be its decomposition into a finite family of finite linear tracks such that for all $\text{ltrack}_1, \text{ltrack}_2 \in \mathcal{L}$, \mathcal{L} contains all their common maximal linear subtracks. Assume that the tracks $\text{ltrack}_i \in \mathcal{L}$ (with increasing segment identifiers w.r.t. $\text{next}_s^{\text{ltrack}}$) are controlled by controllers $\text{RBC}^{\text{ltrack}_i}$ using the protocols in Sect. 2.1 which synchronize on common subtracks. Then we can guarantee safety of the control protocol for the controller of the complex track obtained by interconnecting all linear track controllers $\{\text{RBC}^{\text{ltrack}_i} \mid i \in I\}$.*

5 From Specification to Verification

For the practical application of verification techniques tool support is essential. For this reason, in this section we introduce a full tool chain for automatically checking the invariance of safety properties starting from a given specification and give some experimental results for our RBC case study.

Tool chain. The tool chain is sketched in Fig. 4. In order to capture the systems we want to verify, we use the COD front-end Syspect (cf. Sect. 2). [11] defines the semantics of COD in terms of a timed automata model called Phase Event Automata (PEA). A translation from PEA into TCS is given in [11], which is implemented in the PEA toolkit [7] and used by Syspect.

Given an invariance property, a Syspect model can directly be exported into a TCS in the syntax of H-PILoT. If the specification’s background theory consists of chains of local theory extensions, the user needs to specify via input dialog

⁷ <http://csd.informatik.uni-oldenburg.de/projects/epea.html>

(i) that the pointer extension of H-PILoT is to be used; (ii) which level of extension is used for each function symbol of the specification. With this information, our tool chain can verify invariance of a safety condition fully automatically by checking its invariance for each transition update (cf. Sect. 3.1). Therefore, for each update, Syspect exports a file that is handed over to H-PILoT. The safety invariance is proven if H-PILoT detects the unsatisfiability of each verification task. Otherwise, H-PILoT generates a model violating the invariance of the desired property, which may be used to fix the problems in the specification.

In addition, the PEA toolkit also supports output of TCS into the input language of the abstraction refinement model checker ARMC [22], which we used to verify correctness of the timed train controller from our example.

Experimental results. Table 1 gives experimental results for checking the RBC controller. The table lists execution times for the involved tools: (sys) contains the times needed by Syspect and the PEA toolkit to write the TCS, (hpi) the time of H-PILoT to compute the reduction and to check satisfiability with Yices as back-end, (yic) the time of Yices to check the proof tasks without reductions by H-PILoT. Due to some semantics-preserving transformations during the translation process the resulting TCS consists of 46 transitions. Since our invariant (Inv) is too complex to be handled by the clausifier of H-PILoT, we check the invariant for every transition in two parts yielding 92 proof obligations. In addition, results for the most extensive proof obligation are stated: one part of the speed update. Further, we performed tests to ensure that the specifications are consistent.

The table shows that the time to compute the TCS is insignificant and that the overall time to verify all transition updates with Yices and H-PILoT does not differ much. On the speed update H-PILoT was 5 times faster than Yices alone. During the development of the case study H-PILoT helped us finding the correct transition invariants by providing models for satisfiable transitions. The table lists our tests with the verification of condition (Safe), which is not inductive over all transitions (cf. Sect. 3): here, H-PILoT was able to provide a model after 8s whereas Yices detected unsatisfiability for 17 problems, returned “unknown” for 28, and timed out once (listed as (t.o) in the table). For the consistency check H-PILoT was able to provide a model after 3s, whereas Yices answered “unknown” (listed as (U)).

In addition, we used ARMC to verify the property (Safe_T) of the timed train controller. The full TCS for this proof tasks comprises 8 parallel components, more than 3300 transitions, and 28 real-valued variables and clocks (so it is an infinite state system). For this reason, the verification took 26 hours (on a standard desktop computer).

Table 1. Results

	(sys)	(hpi)	(yic)
(Inv) <i>unsat</i>			
Part 1	11s	72s	52s
Part 2	11s	124s	131s
speed update	11s	8s	45s
(Safe) <i>sat</i>	9s	8s	t.o.
Consistency	13s	3s	(U) 2s

(obtained on: AMD64, dual-core
2 GHz, 4 GB RAM)

⁸ Note that even though our proof methods fully support parametric specifications, we instantiated some of the parameters for the experiments because the underlying provers Yices and ARMC do not support non-linear constraints.

6 Conclusion

We augmented existing techniques for the verification of real-time systems to cope with rich data structures like pointer structures. We identified a decidable fragment of the theory of pointers, and used it to model systems of trains on linear tracks with incoming and outgoing trains. We then proved that certain types of complex track systems can be decomposed into linear tracks, and that proving safety of train controllers for such complex systems can be reduced to proving safety of controllers for linear tracks. We implemented our approach in a new tool chain taking high-level specifications in terms of COD as input. To uniformly specify processes, data and time, [17,4,26] use similar combined specification formalisms. We preferred COD due to its strict separation of control, data, and time, and its compositionality (cf. Sect. 2), which is essential for automatic verification. There is also sophisticated tool support given by Syspect and the PEA toolkit. Using this tool chain we automatically verified safety properties of a complex case study, closing the gap between a formal high-level language and the proposed verification method for TCS. We plan to extend the case study to also consider emergency messages (like in [9]), possibly coupled with updates in the track topology, or updates of priorities. Concerning the track topology, we are experimenting with more complex axiomatizations (e.g. for connectedness properties) that are not in the pointer fragment presented in Sect. 3.3; we already proved various locality results. We also plan to study possibilities of automated invariant generation in such parametric systems.

Acknowledgments. Many thanks to Werner Damm, Ernst-Rüdiger Olderog and the anonymous referees for their helpful comments.

References

1. Abdulla, P.A., Delzanno, G., Rezine, A.: Approximated parameterized verification of infinite-state processes with global conditions. *Form. Method Syst. Des.* 34(2), 126–156 (2009)
2. Abdulla, P.A., Jonsson, B.: Verifying networks of timed processes. In: Steffen, B. (ed.) *TACAS 1998*. LNCS, vol. 1384, pp. 298–312. Springer, Heidelberg (1998)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) *CONCUR 2004*. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
4. Abrial, J.R., Mussat, L.: Introducing dynamic constraints in B. In: Bert, D. (ed.) *B 1998*. LNCS, vol. 1393, pp. 83–128. Springer, Heidelberg (1998)
5. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) *CAV 2001*. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
6. Bradley, A., Manna, Z., Sipma, H.: What’s decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) *VMCAI 2006*. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
7. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) *VMCAI 2006*. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006)

8. Faber, J., Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: Automatic verification of parametric specifications with complex topologies. Reports of SFB/TR 14 AVACS No. 66, SFB/TR 14 AVACS (2010), <http://www.avacs.org>
9. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 233–252. Springer, Heidelberg (2007)
10. Haxthausen, A.E., Peleska, J.: A domain-oriented, model-based approach for construction and verification of railway control systems. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 320–348. Springer, Heidelberg (2007)
11. Hoenicke, J.: Combination of Processes, Data, and Time. Ph.D. thesis, University of Oldenburg, Germany (2006)
12. Hoenicke, J., Olderog, E.R.: CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nordic J. Comput.* 9(4), 301–334 (2002)
13. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
14. Ihlemann, C., Sofronie-Stokkermans, V.: System description: H-PiLoT. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 131–139. Springer, Heidelberg (2009)
15. Jacobs, S., Sofronie-Stokkermans, V.: Applications of hierarchic reasoning in the verification of complex systems. *ENTCS* 174(8), 39–54 (2007)
16. Lahiri, S.K., Bryant, R.E.: Indexed predicate discovery for unbounded system verification. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 135–147. Springer, Heidelberg (2004)
17. Mahony, B.P., Dong, J.S.: Blending Object-Z and timed CSP: An introduction to TCOZ. In: ICSE 1998, pp. 95–104 (1998)
18. McPeak, S., Necula, G.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
19. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. *Form. Asp. Comput.* 20(4-5), 481–505 (2008)
20. Möller, M., Olderog, E.R., Rasch, H., Wehrheim, H.: Integrating a formal method into a software engineering process with UML and Java. *Form. Asp. Comput.* 20, 161–204 (2008)
21. Platzer, A., Quesel, J.D.: European train control system: A case study in formal verification. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)
22. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
23. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
24. Sofronie-Stokkermans, V.: Sheaves and geometric logic and applications to modular verification of complex systems. *ENTCS* 230, 161–187 (2009)
25. Sofronie-Stokkermans, V.: Hierarchical reasoning for the verification of parametric systems. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 171–187. Springer, Heidelberg (2010)
26. Woodcock, J.C.P., Cavalcanti, A.L.C.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) IWFm 2001. BCS Elec. Works. Comp. (2001)

Satisfaction Meets Expectations^{*}

Computing Expected Values of Probabilistic Hybrid Systems with SMT

Martin Fränzle, Tino Teige, and Andreas Eggers

Carl von Ossietzky Universität, Oldenburg, Germany
{fraenzle,teige,eggers}@informatik.uni-oldenburg.de

Abstract. Stochastic satisfiability modulo theories (SSMT), which is an extension of satisfiability modulo theories with randomized quantification, has successfully been used as a symbolic technique for computing reachability probabilities in probabilistic hybrid systems. Motivated by the fact that several industrial applications call for quantitative measures that go beyond mere reachability probabilities, this paper extends SSMT to compute expected values of probabilistic hybrid systems like, e.g., mean-times to failure. Practical applicability of the proposed approach is demonstrated by a case study from networked automation systems.

1 Introduction

Hybrid discrete-continuous dynamics arises when discrete and continuous processes become connected, as in the case of embedded computers and their physical environment. An increasing number of the technical artifacts shaping our ambience are relying on such, often invisible, embedded computer systems, provoking the quest for automatic analysis methods for hybrid behavior. Ideally, such a method would be able to analyze the intricate feedback behavior between the discrete and the continuous components in detail and to rigorously quantify the probability of erroneous behavior, as 100% absence of error is unrealistic in practice and perhaps more an artifact of the model than a reasonable certificate. Unfortunately, procedures for such probabilistic analysis of hybrid systems are still rare, in contrast to a steadily growing body of ever more powerful tools for their qualitative analysis. Suggestions include abstraction-based model-checking procedures like [10], which compute safe upper bounds on the probability of erroneous situations and thus provide verification capabilities, and bounded model-checking (BMC) procedures exploiting stochastic extensions of satisfiability modulo theories [4]. As inherent to BMC, the latter underapproximate the reach set and are thus versatile debugging tools, able to falsify a design.

Industrial applications often call for quantitative measures distinct from the reachability probabilities computed by the aforementioned techniques, as reachability probabilities and related figures frequently tend to 1 in the long run and

^{*} This work has been partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS, www.avacs.org).

$$\forall_{[0 \rightarrow 0.5, 1 \rightarrow 0.5]} x_1 \in \{0, 1\} \exists x_2 \in \{0, 1\} \forall_{[0 \rightarrow 0.8, 1 \rightarrow 0.2]} x_3 \in \{0, 1\} :$$

$$(x_1 = 1 \vee x_2 = 1 \vee x_3 = 0) \wedge (x_1 = 1 \vee x_2 = 0 \vee x_3 = 1) \wedge (y = 4 \cdot x_1 + (x_2 + x_3)^2)$$

maximum probability of satisfaction
maximum conditional expectation of $y \in [0, 8]$

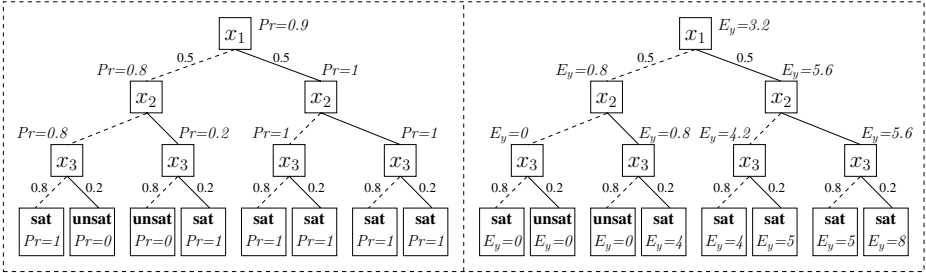


Fig. 1. Semantics of SSMT: probability of satisfaction (left) and conditional expectation (right). Dashed and solid lines denote variable assignments 0 and 1, respectively.

thus are not sufficiently discriminative in practice when applied to systems with unbounded lifetime. Motivated by this fact, this paper provides state-exploratory methods for computing expected values of probabilistic hybrid systems like, e.g., mean-time to failure (MTTF). The suggested method generalizes the probabilistic BMC approach of [48], yet owing to the shape of the requirements has fundamentally different properties than conventional BMC: the resulting BMC procedures are no longer falsification procedures, yet turn to verification procedures being able to verify that a probabilistic hybrid system meets requirements on, e.g., its expected MTTF. Supporting a demonic interpretation of non-determinism, the procedure permits the analysis of partial designs or of open systems in an unknown environment. The method builds on the notion of stochastic satisfiability modulo theories [49], which extends satisfiability modulo theories [1] by randomized quantification over variables [76].

2 Conditional Expectations in SSMT

Stochastic satisfiability modulo theories (SSMT), as introduced in [4], extends the reasoning power of satisfiability modulo theories (SMT) [1] to probabilistic logics. It achieves this by adopting the concept of *randomized quantification* from stochastic propositional satisfiability [76]. An SSMT formula $\Phi = \mathcal{Q} : \varphi$ consists of a quantifier prefix $\mathcal{Q} = Q_1 x_1 \in \mathcal{D}_{x_1} \dots Q_n x_n \in \mathcal{D}_{x_n}$ binding some variables x_i with domains \mathcal{D}_{x_i} by quantifiers Q_i , and of an SMT formula φ in conjunctive normal form (CNF) over some quantifier-free arithmetic theory \mathcal{T} over the reals, integers, and Booleans. That is, φ is a logical *conjunction* of clauses, and a *clause* is a logical *disjunction* of (atomic) arithmetic predicates from \mathcal{T} , as in $\varphi = (x > 0 \vee 2a \cdot \sin(4b) \geq 3) \wedge (y > 0 \vee 2a \cdot \sin(4b) < 1)$. We demand that the individual domains \mathcal{D}_x of quantified variables x are finite, but φ may also contain free variables ranging over infinite domains (in particular, subranges of \mathbb{R}), with the individual types and range bounds being defined

through explicit declarations. Free variables are understood as being innermost existentially quantified. A quantifier Q_i , associated with variable x_i , is either *existential*, denoted as \exists , or *randomized*, denoted as \mathfrak{D}_{d_i} , where d_i is a discrete probability distribution over \mathcal{D}_{x_i} . The value of a variable x_i bound by a randomized quantifier (randomized variable for short) is determined stochastically according to the corresponding distribution d_i , while the value of an existentially quantified variable can be set arbitrarily. We denote a probability distribution d_i by explicit enumeration $[v_1 \rightarrow p_1, \dots, v_m \rightarrow p_m]$ of a finite-domain function associating probability $p_j \geq 0$ to value v_j . The mapping $v_j \rightarrow p_j$ is understood as p_j is the probability of setting variable x_i to value v_j . The distribution satisfies $v_k \neq v_l$ for $k \neq l$, $\sum_{j=1}^m p_j = 1$, and $\mathcal{D}_{x_i} = \{v_1, \dots, v_m\}$. For instance, $\mathfrak{D}_{[0 \rightarrow 0.2, 1 \rightarrow 0.5, 2 \rightarrow 0.3]} x \in \{0, 1, 2\}$ expresses that the variable x is assigned the values 0, 1, and 2 with probabilities 0.2, 0.5, and 0.3, respectively. Adopting the semantics of stochastic formulae established in stochastic SAT [7] and stochastic constraint programming, the original definition of SSMT [4] defines the semantics of an SSMT formula Φ as the *maximum probability of satisfaction*, denoted $Pr(\Phi)$ and illustrated in Fig. 1. Intuitively, a solution here is a strategy for assigning values to the existential variables depending on the values previously assigned to variables with earlier appearance in the prefix, where the optimal strategy maximizes the probability that the SMT formula φ is satisfiable after substituting all quantified variables by their respective values. (As standard, a quantifier-free formula φ in CNF is *satisfiable* iff there exists an assignment σ to the variables in φ s.t. each clause is satisfied under σ , i.e., iff at least one atom in each clause is satisfied under σ . Otherwise, φ is *unsatisfiable*.) For a game-theoretic interpretation, SSMT formulae can be seen as $1\frac{1}{2}$ player games (or $2\frac{1}{2}$ if universal quantifiers are also admitted) between a player selecting values for the existential variables as well as the free variables and a probabilistic enemy selecting those for randomized variables, where the winning condition is satisfaction of φ after substituting the selected values for the respective variables. The game is played in turns defined by the quantifier prefix plus, finally, a selection of values for the free variables by the existential player.

In this paper, we propose a conservative extension of the aforementioned semantics of SSMT formulae which adds considerably to the expressiveness of SSMT. The new semantics is based on the maximum conditional expectation of a designated free variable in an SSMT formula. Game-theoretically, it adds a reward, where the reward is 0 if φ is not satisfied after substituting the selected values for the respective variables and the reward equals the value of the designated variable else. Let $\Phi = \mathcal{Q} : \varphi$ be an SSMT formula with prefix $\mathcal{Q} = Q_1 x_1 \in \mathcal{D}_{x_1} \dots Q_n x_n \in \mathcal{D}_{x_n}$, and let y be a free variable in φ with interval domain $[l_y, u_y] \subset \mathbb{R}$, i.e. $y \notin \{x_1, \dots, x_n\}$. The *maximum conditional expectation of y given Φ* , denoted as $E_y(\Phi)$, is recursively defined as follows:

- (1)
$$E_y(\varepsilon : \varphi) = \max_{\sigma \models \varphi} \sigma(y) \text{ ,}$$
- (2)
$$E_y(\exists x \in \mathcal{D}_x : \mathcal{Q} : \varphi) = \max_{v \in \mathcal{D}_x} E_y(\mathcal{Q} : \varphi[v/x]) \text{ ,}$$
- (3)
$$E_y(\mathfrak{D}_d x \in \mathcal{D}_x : \mathcal{Q} : \varphi) = \sum_{v \in \text{dom}(d)} d(v) \cdot E_y(\mathcal{Q} : \varphi[v/x]) \text{ ,}$$

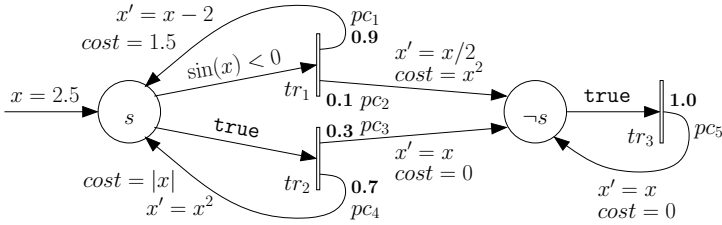


Fig. 2. A probabilistic hybrid automaton \mathcal{A} with costs

where ε denotes the empty and \mathcal{Q} an arbitrary quantifier prefix. Rules 2 and 3 are identical to the semantic rules of SSAT and SSMT conventionally used for defining the maximum probability of satisfaction [74]. Rule 1 generalizes the classical scheme by, instead of just checking for satisfiability of φ [74], determining the maximal value $\sigma(y) \in [l_y, u_y]$ of the random variable y over all satisfying assignments σ to φ . In case no such satisfying assignment exists, we follow the order-theoretic convention that the maximum over the empty subset of the ordered set $[l_y, u_y]$ is the minimum domain value l_y . For an example confer Fig. 11.

Observe that the *maximum conditional expectation* is a conservative extension of the classical semantics based on *maximum probability of satisfaction* in the sense that $Pr(\mathcal{Q} : \varphi) = E_y(\mathcal{Q} : (\varphi \wedge y = 1))$ with a fresh variable y ranging over $[0, 1]$.

3 Probabilistic Hybrid Automata with Costs

The computational entities we want to analyze by SSMT solving are discrete-time probabilistic hybrid automata extended by a notion of *costs*. Formally, a *discrete-time probabilistic hybrid automaton \mathcal{A} with cost function* (PHA, for short), as depicted in Fig. 2, consists of the following.

- A finite set $D = \{d_1, \dots, d_k\}$ of *discrete variables* with *finite* ranges $\text{range}(d_j)$ spanning the discrete state space (sometimes called the *locations*) of \mathcal{A} , plus a finite vector $R = \{x_1, \dots, x_m\}$ of *continuous state components*, where each x_j ranges over an interval $\text{range}(x_j) = [l_{x_j}, u_{x_j}]$ within the reals \mathbb{R} . Thus, the *hybrid state space* of \mathcal{A} is given by $\mathcal{S} = \prod_{j=1}^k \text{range}(d_j) \times \prod_{j=1}^m \text{range}(x_j)$.
- A predicate *init* in an arithmetic theory \mathcal{T} with free variables in D and R describing the *initial state* of \mathcal{A} . W.l.o.g., we demand that there is exactly one initial state of \mathcal{A} , i.e. exactly one valuation in \mathcal{S} satisfies *init*.
- A finite family $Tr = \{tr_1, \dots, tr_\ell\}$ of *symbolic transitions*, each comprising
 - an arithmetic predicate $g(tr_j)$ in theory \mathcal{T} over variables in D and R , called the *transition guard*, which states the conditions on the discrete as well as the continuous state under which the transition may be taken. To avoid pathological situations otherwise arising in the definition of an

adversary in a probabilistic game, we demand that the automaton is non-blocking in the sense that $\forall s \in \mathcal{S} \exists t \in Tr : s \models g(t)$ [\[1\]](#)

- a *probability distribution* $p(tr_j) \in P(PC_{tr_j})$, where PC_{tr_j} is a finite, nonempty set of symbolic transition alternatives and $P(PC_{tr_j})$ denotes the set of probability distributions over PC_{tr_j} . $p(tr_j)$ assigns to transition tr_j a distribution over $|PC_{tr_j}|$ many transition alternatives.
- for each transition alternative $pc \in PC_{tr_j}$ of transition tr_j an *assignment predicate* $asgn(tr_j, pc)$ defining the successor state. $asgn(tr_j, pc)$ is an arithmetic predicate in \mathcal{T} over variables in D and R as well as *primed* variants thereof, i.e. $D' = \{d'_1, \dots, d'_k\}$ and $R' = \{x'_1, \dots, x'_m\}$, the latter representing the successor states. We demand that assignments uniquely determine the successor state for each state satisfying the guard, i.e. for each $tr \in Tr$, $pc \in PC_{tr}$, $g(tr) \Rightarrow \exists^1 d'_1, \dots, d'_k x'_1, \dots, x'_m : asgn(tr, pc)$ holds where \exists^1 denotes unique existence.
- The function $cost : Tr \times PC \times \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$ with $PC = \bigcup_{tr \in Tr} PC_{tr}$ associates to each transition alternative $pc \in PC_{tr}$ of each transition tr a non-negative cost value $cost(tr, pc, s)$ that also depends on the current state s . We demand that $cost$ is defined by a \mathcal{T} -term.

The semantics of such an automaton \mathcal{A} is defined by its runs. A *run* r of \mathcal{A} is an alternating finite or infinite sequence of states and selections of transitions and probabilistic transition alternatives, i.e. $r = \langle s_0, (t_1, p_1), s_1, \dots, (t_k, p_k), s_k, \dots \rangle$ with $s_i \in \mathcal{S}$, $t_i \in Tr$, $p_i \in PC_{t_i}$ s.t. r starts in the initial state, i.e. $s_0 \models init$, the transition guards are satisfied $s_{k-1} \models g(t_k)$, and the successor states are defined by the assignments $(s_{k-1}, s_k) \models asgn(t_k, p_k)$. The cost of step k in r is given by $cost(t_k, p_k, s_{k-1})$. For an example, consider the PHA \mathcal{A} from Fig. [2](#). The unique initial state of \mathcal{A} is $(s, x = 2.5)$. Transition tr_1 cannot be taken since the guard is not satisfied due to $\sin(2.5) > 0.59$. \mathcal{A} decides to take tr_2 followed by the probabilistic choice pc_4 of probability 0.7. The cost of this step is $|x| = 2.5$, and \mathcal{A} enters successor state $(s, x = 6.25)$. The guard of tr_1 is now satisfied since $\sin(6.25) < 0$. \mathcal{A} thus can execute tr_1 and may select transition alternative pc_1 with probability 0.9. The transition cost is in this case given by the constant 1.5, and the post state is $(s, x = 4.25)$. Thereafter selecting tr_2 non-deterministically and pc_3 probabilistically, \mathcal{A} performs a step to $(\neg s, x = 4.25)$ at zero cost, altogether yielding a run of length 3 reaching $(\neg s, x = 4.25)$ with an accumulated cost of $2.5 + 1.5 + 0 = 4$.

In the remainder of this article, we will be interested in the *expected value of the accumulated cost* when \mathcal{A} reaches a target state. Due to the presence of non-determinism, we first define an *adversary* that resolves the non-determinism in \mathcal{A} . While in general adversaries may depend on the entire history of the system, we do here restrict ourselves to Markovian adversaries that depend on the current system state only. A Markovian adversary for \mathcal{A} is a total function $adv : \mathcal{S} \rightarrow Tr$ that maps the current state to an enabled transition, i.e. $\forall s \in \mathcal{S} : s \models g(adv(s))$.

¹ Note that while this condition is undecidable due to non-computability of the reachable state space of a disc.-time HA, there are sufficient conditions that can be verified by SMT solving, like enabledness of transitions throughout the static state space.

Definition 1 (Cost expectation). Let \mathcal{A} be a PHA with cost function, ts be a \mathcal{T} -predicate over variables in D and R defining the target states, and $adv : \mathcal{S} \rightarrow Tr$ be a Markovian adversary. The cost expectation for \mathcal{A} under adversary adv is the least (wrt. the product order) solution of the equation system

$$\left(CE_{\mathcal{A},ts,adv}(z) = \begin{cases} 0 & \text{if } z \models ts \\ \sum_{pc \in PC_{tr}} p(tr)(pc) \cdot \left(\begin{array}{l} cost(tr, pc, z) \\ + CE_{\mathcal{A},ts,adv}(z') \end{array} \right) & \text{if } z \not\models ts \end{cases} \right)_{z \in \mathcal{S}}$$

with $tr = adv(z)$, and $(z, z') \models asgn(tr, pc)$. For a particular state $s \in \mathcal{S}$, the cost expectation for reaching a target state from state s under adversary adv is $CE_{\mathcal{A},ts,adv}(s)$, while the worst-case cost expectation for reaching a target state from state s is $CE_{\mathcal{A},ts}(s) := \inf_{adv: \mathcal{S} \rightarrow Tr} CE_{\mathcal{A},ts,adv}(s)$.

Note that the cost expectation is non-negative. In the remainder of the article, we will consider the problem of deciding whether the cost expectation for \mathcal{A} is acceptable, where acceptability is defined by a threshold value θ to be exceeded irrespective of the actual adversary. An example is a setting where costs of steps correspond to their durations and the target states denote system failures. The expected cost then is \mathcal{A} 's mean time to failure (MTTF) and the threshold θ can be interpreted as a requirement on the MTTF of the design under inspection. Adopting a demonic interpretation of non-determinism, the latter has to be guaranteed irrespective of the actual adversary.

Definition 2 (Cost-expectation model-checking). Given a PHA \mathcal{A} with cost function, a \mathcal{T} -predicate ts over variables in D and R defining the target states, and a target threshold $\theta \geq 0$, the cost-expectation model-checking problem (CEMC) is to determine whether $CE_{\mathcal{A},ts}(i) \geq \theta$ for the initial state $i \models init$.

4 Reducing CEMC to SSMT

We facilitate the automatic verification of CEMC problems by a reduction to SSMT. Unfortunately, only step-bounded behavior of a PHA \mathcal{A} can be directly encoded into SSMT due to the absence of quantifier alternation over real-valued variables, which would be essential for a complete characterization of the reachable state set. To alleviate this problem, we first introduce a *step-bounded* variant of the CEMC problem which provides a lower bound on the worst-case cost expectation of \mathcal{A} . In a second step, we present an SSMT encoding of PHA such that the additive inverse of the maximum conditional expectation of the cost-variable given the resulting SSMT formula coincides with the cost expectation of the step-bounded CEMC problem. Thus, together with the SSMT algorithm in Section 5, the proposed approach constitutes a *verification* procedure for the general CEMC problem with target threshold θ in the sense that once a lower expectation bound $l \geq \theta$ is computed, the unbounded CEMC problem is decided to be true.

Step-bounded CEMC problem. The step-bounded CEMC problem replaces the mutually recursive equation defining the unbounded cost expectation for each state in Def. [□](#) by a well-founded recursion over the remaining step depth.

Definition 3 (*k-step minimum cost expectation*). *Be \mathcal{A} a PHA with cost function $cost$, ts a \mathcal{T} -predicate defining the target states, and $k \in \mathbb{N}$ a step bound. The k -step minimum cost expectation for reaching a target state from a state $s \in \mathcal{S}$ and cost seeds $c \in \mathbb{R}_{\geq 0}$, $d \in \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$, denoted $CE_{\mathcal{A},ts}^k(s, c, d)$, is defined as follows. With $En = \{tr \in Tr : s \models g(tr)\}$ and $(s, s') \models asgn(tr, pc)$,*

$$CE_{\mathcal{A},ts}^k(s, c, d) = \begin{cases} c & \text{if } s \models ts \text{ and} \\ c + d(s) & \text{if } k = 0, s \not\models ts \text{ and else} \\ \min_{tr \in En} \sum_{pc \in PC_{tr}} (p(tr)(pc) \cdot CE_{\mathcal{A},ts}^{k-1}(s', c + cost(tr, pc, s), d)) & \end{cases}$$

The k -step minimum cost expectation can be used for obtaining safe estimates of the worst-case cost expectation, as the following lemma shows.

Lemma 1. *Assume a PHA \mathcal{A} with cost function and a \mathcal{T} -predicate ts defining the target states. Let $k \in \mathbb{N}$. Then $CE_{\mathcal{A},ts}(s) \geq CE_{\mathcal{A},ts}^k(s, 0, \mathbf{0})$, where $\mathbf{0}$ is the constant function assigning 0 to all states. Furthermore, the sequence $CE_{\mathcal{A},ts}^k(s, 0, \mathbf{0})$ is (not necessarily strictly) monotonically increasing.*

Likewise, if $C : \mathcal{S} \rightarrow \mathbb{R}_{\geq 0}$ satisfies $C \geq CE_{\mathcal{A},ts}$ then $CE_{\mathcal{A},ts}(s) \leq CE_{\mathcal{A},ts}^k(s, 0, C)$.

Proof. We show by induction on k that $CE_{\mathcal{A},ts}^k(s, c, CE_{\mathcal{A},ts}) = c + CE_{\mathcal{A},ts}(s)$ for arbitrary $c \geq 0$. As $CE_{\mathcal{A},ts}^k(s, c, d)$ is monotonic in d and as C is such that $\mathbf{0} \leq CE_{\mathcal{A},ts} \leq C$, this implies both inequalities $CE_{\mathcal{A},ts}(s) \geq CE_{\mathcal{A},ts}^k(s, 0, \mathbf{0})$ and $CE_{\mathcal{A},ts}(s) \leq CE_{\mathcal{A},ts}^k(s, 0, C)$ by taking $c = 0$.

For the base case of the induction, it is immediate from the definition of CE^k that $CE_{\mathcal{A},ts}^0(s, c, CE_{\mathcal{A},ts,adv}) = c + CE_{\mathcal{A},ts,adv}(s)$ holds. For $k > 0$,

$$\begin{aligned} & CE_{\mathcal{A},ts}^k(s, c, CE_{\mathcal{A},ts}) \\ = & \begin{cases} c & \text{if } s \models ts \text{ holds, otherwise} \\ \min_{tr \in En} \sum_{pc \in PC_{tr}} (p(tr)(pc) \cdot CE_{\mathcal{A},ts}^{k-1}(s', c + cost(tr, pc, s), CE_{\mathcal{A},ts})) & \end{cases} \\ [\text{Ind. Hyp.}] & \begin{cases} c & \text{if } s \models ts \text{ holds, otherwise} \\ \min_{tr \in En} \sum_{pc \in PC_{tr}} (p(tr)(pc) \cdot (c + cost(tr, pc, s) + CE_{\mathcal{A},ts}(s'))) & \end{cases} \\ [\text{Def. } \square] & \begin{cases} c & \text{if } s \models ts \text{ holds, otherwise} \\ \min_{tr \in En} \sum_{pc \in PC_{tr}} (p(tr)(pc) \cdot (c + cost(tr, pc, s) + \inf_{adv} CE_{\mathcal{A},ts,adv}(s'))) & \end{cases} \\ = & \begin{cases} c & \text{if } s \models ts \text{ holds, otherwise} \\ \inf_{adv} \sum_{pc \in PC_{tr^*}} (p(tr^*)(pc) \cdot (c + cost(tr^*, pc, s) + CE_{\mathcal{A},ts,adv}(s^*))) & \end{cases} \\ [\text{Def. } \square] & \inf_{adv} (c + CE_{\mathcal{A},ts,adv}(s)) \quad [\text{Def. } \square] \quad c + CE_{\mathcal{A},ts}(s) \end{aligned}$$

where $tr^* = adv(s)$ and $(s, s^*) \models asgn(tr^*, pc)$. The monotonic increase of the sequence $CE_{\mathcal{A},ts}^k(s, 0, \mathbf{0})$ can be shown by a straightforward induction. \square

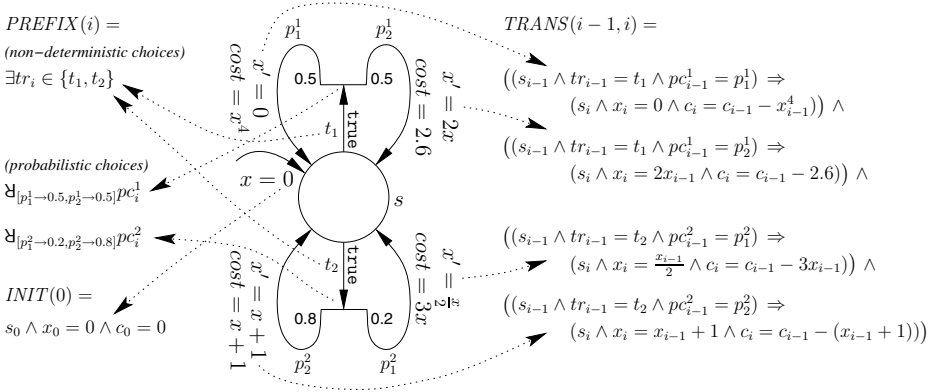


Fig. 3. Example of SSMT encoding. The inverted sign of the cost compensates for the duality betw. existential quantifier in SSMT and demonic non-determinism in PHA.

SSMT encoding of step-bounded CEMC. In [4], we have developed a reduction of step-bounded probabilistic reachability problems of discrete-time PHA to SSMT problems. In the following, we adapt this SSMT reduction to the more general step-bounded CEMC problem, which subsumes probabilistic bounded reachability as a special case. The key idea of the SSMT encoding, however, remains the same as in [4] and we repeat it only in brief due to space limitations.

Given a discrete-time PHA \mathcal{A} with costs, we can encode the initial state of \mathcal{A} by a predicate $INIT(0)$ using a copy of the discrete and continuous variables in D and R indexed with 0, the non-deterministic selection of a transition and the probabilistic choice of a transition alternative for step i by a block of quantified variables $PREFIX(i)$ with index i , and the transition relation of step i by a predicate $TRANS(i-1, i)$ where undecorated and primed variables carry index $i-1$ and i , resp., e.g. $x' = 2x \rightsquigarrow x_i = 2x_{i-1}$. An illustrating example is shown in Fig. 3. Observe that *non-deterministic* and *probabilistic* choices in \mathcal{A} are described by *existential* and *randomized* quantification, resp., in $PREFIX(i)$. Furthermore, we take indexed copies of a fresh variable $c \in [l_c, 0]$ that accumulates the *negative* costs of the individual transition steps: the predicate $INIT(0)$ forces $c_0 = 0$ as the costs are initially zero, and the cost progress is determined in $TRANS(i-1, i)$ by predicates $c_i = c_{i-1} - cost(tr, pc)$ for each valid transition and transition alternative pair $(tr, pc) \in Tr \times PC$ (cf. Fig. 3). Note that we subtract step costs from the accumulated cost rather than add them to it as in the original PHA. The SSMT encoding thus features negative costs to be maximized rather than positive costs to be minimized, reflecting the duality between the SSMT semantics, where the player wants to maximize the expectation, and PHA, where the adversary minimizes it. Hence, the *additive inverse* $-E$ of the maximum expectation E of negative costs, as described by the SSMT encoding, then obviously gives the (step-bounded) minimum cost expectation for the PHA \mathcal{A} . We need to remark that variable c must range over an interval domain, which

is required by the SSMT definition, and thus calls for a lower domain bound l_c . The reason for that is inherent in an optimization of the SSMT algorithm introduced in Section 5. However, practically this need not to be a huge restriction as, first, l_c can be chosen arbitrarily small and, second, in cases where the maximum value $cost_{\max}$ of the individual transition costs exists and is known — which is most often the case in industrial applications when considering entities like time, position, or velocity — the value of l_c can be safely set to $k \cdot (-cost_{\max})$, where k is the step bound from Definition 3.

Given a predicate ts over variables in D and R defining the target states, we denote by $TARGET(i-1)$ the predicate that results from replacing all variables $v \in D \cup R$ in ts by v_{i-1} . For any given step bound $k \in \mathbb{N}$, the k -bounded CEMC formula $CEMC_{\mathcal{A},ts}(k)$ is the SSMT formula

(4)

$$PREFIX(1, k) : INIT(0) \wedge \bigwedge_{i=1}^k \left((TARGET(i-1) \Rightarrow c_i = c_{i-1}) \wedge (\neg TARGET(i-1) \Rightarrow TRANS(i-1, i)) \right)$$

where $PREFIX(1, k)$ abbreviates $PREFIX(1) \dots PREFIX(k)$. Please note that the quantifier-free body of this formula has as its models both runs reaching the target (in which case the cost is kept constant after reaching the target, cf. upper line of the conjunction) and runs not reaching the target within k steps. The rationale is that the costs of the latter runs still provide safe lower bounds on the costs of their extensions. Actually, this approximation is the same as in Lemma 4 such that we have obtained a symbolic encoding of step-bounded CEMC:

Proposition 1. *Be \mathcal{A} a PHA with costs and ts be a target states predicate. Given a step-bound $k \in \mathbb{N}$, the equation $CE_{\mathcal{A},ts}^k(\iota, 0, \mathbf{0}) = -E_{c_k}(CEMC_{\mathcal{A},ts}(k))$ holds, where c_k is the k -th copy of cost variable c and $\iota \models init$.*

Note that $CEMC_{\mathcal{A},ts}(k)$ is an SSMT formula and that its maximum expected value $E_{c_k}(CEMC_{\mathcal{A},ts}(k))$ thus can be determined by a procedure for computing expected values of SSMT formulae, as explained in the next section.

5 SSMT Algorithm for Conditional Expectation

The key idea of the algorithm computing the maximum conditional expectation of a variable $y \in [l_y, u_y]$ in an SSMT formula is a branch-and-bound search that implements the semantic rules (1) to (3) by branching through the quantifier tree, as illustrated in Fig. 1, yet optimizes this by pruning the tree. The algorithm basically traverses the Cartesian product of the domains of the quantified variables and upon each complete assignment to the quantified variables, seeks for a solution of the remaining quantifier-free SMT problem which maximizes y .

For maximizing y in quantifier-free SMT problems, we use bifurcation search in interval constraint propagation (ICP), which yields safe estimates on the actual maximum due to the conservative approximation provided by interval arithmetic. Such a procedure has been integrated into our ICP-based satisfiability solver iSAT [3], which addresses large Boolean combinations of non-linear

arithmetic constraints. This procedure adequately handles the quantifier-free base case corresponding to rule (1). Being based on ICP, iSAT offers guaranteed termination despite the undecidable arithmetic domain addressed and is refutationally sound, yet not refutationally complete, which means that **unsat** results are reliable while some unsatisfiable problems may be classified as potentially satisfiable. Note that the latter (which only happens if the problem fails to be δ -robustly unsatisfiable, i.e. has a δ -small perturbation in the constants which renders it satisfiable for some small δ) can only lead to overestimation of the maximum expectation, and thus to a safe lower estimate of the additive inverse.

For dealing with quantifiers in the recursive cases (2) and (3), we have to traverse the quantifier tree, thereby—if done naively—generating one SMT problem per element of the Cartesian product of the quantifier ranges, i.e. exponentially many problems. To mitigate this explosion, we integrate an algorithmic optimization that saves visits to major parts of the quantifier ranges based on *thresholding* [64]. Here, we assume that the algorithm is given a lower and an upper threshold $t_l \leq t_u$ in the domain of the cost variable and that it is not required to return an exact value if the actual expectation lies outside $[t_l, t_u]$. Instead, a witness $e_y < t_l$ suffices if $E_y(\Phi) < t_l$ and a witness $e_y > t_u$ if $E_y(\Phi) > t_u$. Note that this suffices for deciding the CEMC problem with threshold θ by setting both the lower and upper threshold to θ . Conversely, the algorithm can still be used for actually computing the expectation by setting the lower and the upper thresholds to the respective domain limits of the cost variable.

The algorithm for computing $E_y(\mathcal{Q} : \varphi)$ with thresholds t_l, t_u is presented in pseudo-code as Alg. 1. Following rules (2) and (3), the algorithm *MaxExp*($y, \mathcal{Q} : \varphi, t_l, t_u$) descends recursively through the quantifiers in \mathcal{Q} , yet prunes branches by *thresholding*, which skips subproblems when detecting that a lower threshold can no longer be reached or that an upper threshold already is exceeded. This optimization build on the monotonicity argument that the intermediate values e_i of the exact expectation e computed while branching over the values in the domain of a quantifier are never decreasing, i.e. $e_{i+1} \geq e_i$, and do all establish lower bounds on e , i.e. $e \geq e_i$. Such monotonicity cannot be expected from a straightforward implementation of the semantics, as the terms of the sum in rule (3) of $E_y(\Phi)$ may be both positive and negative if the domain of the designated variable y features positive and negative values. Therefore, we initialize the return value e_y in Alg. 1 by the minimum domain value l_y of y . For an existential quantifier, we replace the provisional return value by the actual value of the current subproblem whenever the latter is larger, which is a monotonic process. For the randomized case, we renormalize and accumulate the non-negative increases $p_v \cdot (e'_y - l_y) \geq 0$ wrt. the minimum value l_y of the individual expectations $e_y \geq l_y$. The increasing partial sums for e_y finally yield the desired expectation as $l_y + \sum_{v \in \mathcal{D}} p_v \cdot (e_y^v - l_y) = \sum_{v \in \mathcal{D}} p_v \cdot e_y^v$ due to $\sum_{v \in \mathcal{D}} p_v = 1$.

With such monotonic estimates, we are able to safely conclude that an upper threshold t_u is exceeded based on the intermediate expectation e_i by checking $e_i > t_u$. In case of success, we skip investigation of all remaining subproblems and return the witness value e_i . For randomized variables, we can also safely

Algorithm 1. $MaxExp(y, \mathcal{Q} : \varphi, t_l, t_u)$

```

1:  $e_y := l_y$ . {Initialize return value with domain minimum of  $y$ .}
2: {Existential quantifier.}
3: if  $\mathcal{Q} = \exists x \in \mathcal{D} \mathcal{Q}'$  then
4:   while  $\mathcal{D} \neq \emptyset$  do
5:     if  $e_y > t_u$  or  $e_y = u_y$  then
6:       return  $e_y$ . {Upper threshold exceeded or maximum possible value.}
7:     end if
8:     select  $v \in \mathcal{D}$ ,  $\mathcal{D} := \mathcal{D} - \{v\}$ .
9:      $new\_t_l := \max(t_l, e_y)$ . {New lower threshold: neglect expectations  $< e_y$ .}
10:     $e'_y := MaxExp(y, \mathcal{Q}' : \varphi[v/x], new\_t_l, t_u)$ . {Compute expectation for  $x = v$ .}
11:     $e_y := \max(e_y, e'_y)$ . {Store greater expectation.}
12:   end while
13:   return  $e_y$ . {Return maximum expectation.}
14: end if
15: {Randomized quantifier.}
16: if  $\mathcal{Q} = \forall_d x \in \mathcal{D} \mathcal{Q}'$  then
17:   while  $\mathcal{D} \neq \emptyset$  do
18:     if  $e_y > t_u$  or  $e_y = u_y$  then
19:       return  $e_y$ . {Upper threshold exceeded or maximum possible value.}
20:     end if
21:      $p_{remain} := \sum_{w \in \mathcal{D}} d(w)$ . {Probability mass of remaining branches.}
22:      $max\_inc := p_{remain} \cdot (u_y - l_y)$ . {Maximum possible remaining increase of  $e_y$ .}
23:     if  $e_y + max\_inc < t_l$  then
24:       return  $e_y$ . {Lower threshold cannot be reached by remaining branches.}
25:     end if
26:     select  $v \in \mathcal{D}$ ,  $\mathcal{D} := \mathcal{D} - \{v\}$ ,  $p_v := d(v)$ .
27:      $needed\_inc := t_l - e_y$ . {Total increase needed to reach lower threshold.}
28:      $max\_inc\_others := (p_{remain} - p_v) \cdot (u_y - l_y)$ . {Maximum possible increase of remaining branches except  $x = v$ .}
29:      $needed\_inc\_v := needed\_inc - max\_inc\_others$ . {Needed increase of branch  $x = v$  to reach  $t_l$ .}
30:      $new\_t_l := l_y + needed\_inc\_v / p_v$ . {Branch  $x = v$  must yield expectation  $\geq new\_t_l$  to reach  $t_l$ .}
31:      $new\_t_u := l_y + (t_u - e_y) / p_v$ . {Expectation  $> new\_t_u$  is enough to exceed  $t_u$ .}
32:      $e'_y := MaxExp(y, \mathcal{Q}' : \varphi[v/x], new\_t_l, new\_t_u)$ . {Expectation for  $x = v$ .}
33:      $e_y := e_y + p_v \cdot (e'_y - l_y)$ . {Increment by current weighted increase.}
34:   end while
35:   return  $e_y$ . {Return weighted sum of expectations.}
36: end if
37: {No quantifier left. Solve quantifier-free SMT formula, return max value for  $y$ .}
38: return  $solve_{SMT}(y, \varphi)$ .

```

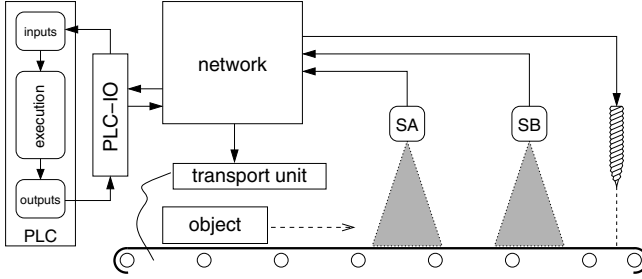


Fig. 4. A networked automation system from [5]. (Source of figure: [8])

determine whether the lower threshold can no longer exceed t_l . Detection depends on the probability mass p_{remain} of all values $v \in \mathcal{D}$ not yet explored. As the maximum increase $e_{i+1} - e_i$ of the partial sum per subproblem is the domain width $u_y - l_y$ of y times the probability of the branch, the maximum possible increase due to all remaining branches is $m = p_{remain} \cdot (u_y - l_y)$. If $e_i + m < t_l$ then witness value $e_i + m$ can be returned without exploring any remaining branch.

It remains to explain how the thresholds are updated for analyzing the subordinate quantifiers. In the existential case, we can hand over $t'_l = \max(t_l, e_y)$ to the subordinate quantifier, where e_y is the current intermediate expectation, since subproblems with expectation less than e_y will not modify the maximum. The upper threshold remains unchanged. For the randomized case, the upper threshold may be decreased by considering the already computed expectation e_y : the increase $t_u - e_y$ needed to reach t_u is first normalized wrt. the probability p_v of the current value v , i.e. divided by p_v , as the increase of the result of the subproblem will be weighted with p_v , and then added to the domain minimum l_y to obtain an expectation threshold. Formally, to check whether the intermediate expectation $e_y + p_v \cdot (e'_y - l_y)$, with e'_y being the result of the next subproblem, is greater than t_u , we can check the equivalent inequality $e'_y > l_y + (t_u - e_y)/p_v$. For computing the lower threshold, we first determine the total increase $t_l - e_y$ needed to reach t_l , and then subtract from it the maximum possible increase of the remaining branches except for $x = v$. This gives the minimum increase for branch $x = v$ required to reach t_l . Again, dividing the result by p_v and adding it to the domain minimum l_y yields the lower threshold for the next subproblem.

The algorithm including the aforementioned optimizations has been implemented in an extension of the SSMT-solver SiSAT [49]. SiSAT does also handle the generation of the BMC formula (4) from a symbolic description of a PHA.

6 Practical Application to Networked Automation Systems

In this section, we illustrate CEMC on a case study of a *networked automation system* (NAS) from [58]. As shown in Fig. 4, it involves networked control by programmable logic controllers (PLCs) connected to several sensors and actuators via communication networks. Its objective is to transport a workpiece from

its initial position to the drilling position by means of a conveyor belt. The PLC can set the deceleration of the belt via network messages to the transportation unit, but cannot determine the position of the object unless it hits two sensors SA and SB close to the drilling position. The sensors are connected to the IO card of the PLC over the network. When the object reaches sensor SA, the PLC reacts with sending a command to the transportation unit that forces the belt to decelerate to slow speed. Likewise, the belt is asked to decelerate to stand-still when the PLC notices that SB has been reached. The goal is to stop the object close to the drilling position despite the uncontrollable latencies in the network.

Using abstract length units (lu) and time steps (ts)², the positions of SA and SB are 699 lu and 470 lu, resp., while the desired drilling position is at 0 lu. The initial speed of the object is 24 lu/ts, its slow speed is 4 lu/ts, while the decelerations for the two types of speed changes at SA and SB are 2 and 4 lu/ts², resp. The network routing time is determined stochastically, needing 1 ts for delivery with probability 0.9 and 2 ts with probability 0.1. The cycle time of the PLC-IO card is 10 ts, and of the PLC is 7 ts. This yields 70 equally probable initial phase shifts. Furthermore, as the minimum sampling interval is 1 ts while the initial speed is 24 lu/ts, the initial position of the object when sampling starts is distributed equally over the 24 values 999...976 lu.

We have previously modeled the NAS case study in [8] as a system of 10 parallel PHA. When flattening the parallelism, the overall model would consist of more than 24 million discrete states while the continuous state space is spanned by 23 real-valued variables. The predicative SSMT encoding, however, avoids an explicit construction of the product automaton³. In [8], we were interested in the probability that the workpiece stops close enough to the drilling position. To do this, we first determined the number of steps k s.t. the object has stopped in all system runs of length k , which applies for $k = 44$. Then, we were able to compute the probability of stopping in a desired target region within $k = 44$ steps. Taking for example the region 100...0 lu, this probability is $\approx 39.7\%$.

Our extended setting now permits to address more sophisticated questions than just classical reach probabilities: it can compute or check expected values like, e.g., the mean-time to stop or the expected final position of the object. Note that the SSMT algorithm from Sect. 5 can compute the expectation of any monotonic variable in the model. If we can provide a bound k on the length of paths reaching target states ($k = 44$ here), the computed expectation furthermore is exact (up to the rather small interval width accepted for models in interval constraint propagation) rather than a safe approximation.

For the automatic analysis, we used our novel extension of the SiSAT tool [4,9] and our SSMT encoding of the NAS from [8]. The progression of the expected object position, the expected speed, and the expected time⁴ over the number of transition steps are shown in Fig. 5. These stabilize at step depth 44, yielding expected final object position 57.84 lu and expected mean-time to stop 46.32 ts.

² One length unit corresponds to 0.01 mm of the real system [5], a time step to 1 ms.

³ For details of the formal NAS model and its SSMT encoding confer to [8, Sect. 6].

⁴ The model employs a scheduled event semantics and thus uses discrete, yet not equidistant time. Consequently, expected times are not in 1-1 relation to step counts.

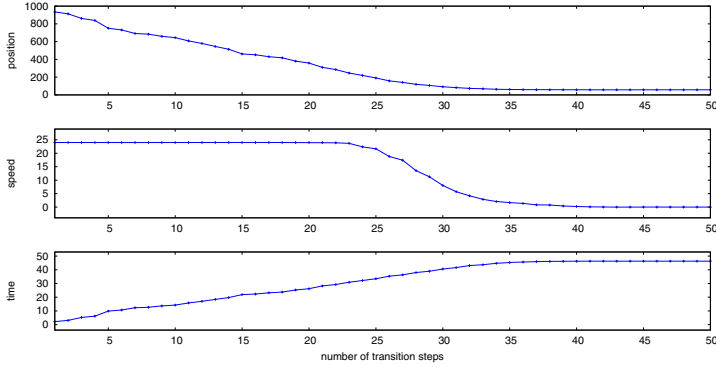


Fig. 5. Expected values of position, speed, and time as developing over the step depth

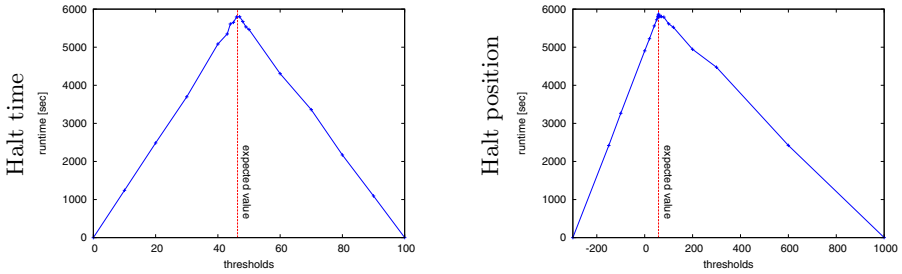


Fig. 6. Runtime effect of thresholding on checking expected halt time and position

Probabilistic requirements for industrial applications frequently demand a certificate that certain target thresholds on expected values are exceeded, e.g. that the mean-time to failure is greater than 1 year. This motivates the algorithmic optimization of *thresholding* (cf. Alg. [II](#)) for improving performance of the tool on such decision problems. The experimental results (obtained on a 2.4 GHz AMD Opteron machine with 128 GB physical memory running Linux) for the *expected time* and *expected position* at step $k = 44$ are shown in Fig. [6](#). The thresholds are provided on the x-axis (lower and upper thresholds coincide, i.e. $t_l = t_u$) and the corresponding runtimes for solving on the y-axis. The graphs confirm that runtimes are reduced significantly when thresholds occurring in the decision problem are distinct from the actual expectation value. These empirical results show that thresholding saves visits to major parts of the quantifier ranges and thus is an effective algorithmic optimization in practice.

7 Conclusion

We have extended previous work on the symbolic analysis of reachability probabilities for probabilistic hybrid systems to the computation of expectation values in such systems under the influence of adversaries. This facilitates the computation of mean-time-to-failure and of related figures for hybrid systems under

a demonic interpretation of non-determinism, thus permitting the analysis of partially developed systems and of open systems in an unknown environment. The overall procedure, which is based on a stochastic extension to satisfiability modulo theories and on a reduction of step-bounded expectations to such satisfiability problems, provides exact (modulo the ability of the underlying SMT solver to decide formulae; otherwise, safe lower bounds) figures for step-bounded expectation values and convergent safe lower estimates for unbounded expectations. Consequently, requirements on e.g. the minimum MTTF of a system under design can be *verified* (yet not falsified due to depth boundedness) with these bounded procedures, even in the presence of an unknown environment. This is dual to previous closely related work by the authors addressing reach probabilities [48], where probabilistic bounded model checking was able to provide falsification yet not verification, as usual for bounded model checking. To the best of our knowledge, our new procedure provides the first procedure safely estimating MTTFs and related figures in discrete-time probabilistic hybrid automata. Extensions to continuous time are underway.

References

1. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, et al. (eds.) [2], ch. 26, pp. 825–885.
2. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (February 2009)
3. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. JSAT Special Issue on SAT/CP Integration 1, 209–236 (2007)
4. Fränzle, M., Hermanns, H., Teige, T.: Stochastic Satisfiability Modulo Theory: A Novel Technique for the Analysis of Probabilistic Hybrid Systems. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 172–186. Springer, Heidelberg (2008)
5. Greifeneder, J., Frey, G.: Probabilistic hybrid automata with variable step width applied to the analysis of networked automation systems. In: Proc. 3rd IFAC Workshop on Discrete Event System Design. IFAC, pp. 283–288 (2006)
6. Majercik, S.M.: Stochastic Boolean satisfiability. In: Biere, et al. (eds.) [2], ch. 27, pp. 887–925
7. Papadimitriou, C.H.: Games against nature. J. Comput. Syst. Sci. 31(2), 288–301 (1985)
8. Teige, T., Eggers, A., Fränzle, M.: Constraint-based analysis of concurrent probabilistic hybrid systems: An application to networked automation systems. In: Non-linear Analysis: Hybrid Systems (accepted for publication 2010)
9. Teige, T., Fränzle, M.: Stochastic Satisfiability modulo Theories for Non-linear Arithmetic. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 248–262. Springer, Heidelberg (2008)
10. Zhang, L., She, Z., Ratschan, S., Hermanns, H., Hahn, E.M.: Safety verification for probabilistic hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. LNCS, vol. 6174, pp. 196–211. Springer, Heidelberg (2010)

Showing Full Semantics Preservation in Model Transformation – A Comparison of Techniques*

Mathias Hülsbusch¹, Barbara König¹, Arend Rensink², Maria Semenyak³,
Christian Soltenborn³, and Heike Wehrheim³

¹ Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany

² Department of Computer Science, University of Twente, The Netherlands

³ Institut für Informatik, Universität Paderborn, Germany

Abstract. Model transformation is a prime technique in modern, model-driven software design. One of the most challenging issues is to show that the semantics of the models is not affected by the transformation. So far, there is hardly any research into this issue, in particular in those cases where the source and target languages are different.

In this paper, we are using two different state-of-the-art proof techniques (explicit bisimulation construction versus borrowed contexts) to show bisimilarity preservation of a given model transformation between two simple (self-defined) languages, both of which are equipped with a graph transformation-based operational semantics. The contrast between these proof techniques is interesting because they are based on different model transformation strategies: triple graph grammars versus in situ transformation. We proceed to compare the proofs and discuss scalability to a more realistic setting.

1 Background

One of today's most promising approaches for building complex software systems is the Object Management Group's *Model Driven Architecture* (MDA). The core idea of MDA is to first model the target system in an abstract, platform-independent way, and then to refine that model step by step, finally producing platform-specific, executable code. The refinement steps are to be performed automatically using so-called *model transformations*; the knowledge needed for each refinement step is contained in the respective transformation.

As a consequence, in addition to the source model's correctness, the correctness of the model transformations is crucial for MDA; if they contain errors, the target system might be seriously flawed. But how to ensure the correctness of a model transformation? In this paper, we take a formal approach: We want to *prove* that the presented model transformation is *semantics preserving*, i.e., we prove that the behaviour of source and generated target model is equivalent (in a very strict sense, discussed below) for *every* source model we potentially start with.

As an example of a realistically sized case for which behavioural preservation is desirable, in [5] we have presented a model transformation from UML Activity Diagrams

* Partially supported by DFG project Behaviour-GT.

[19] (called AD below) to TAAL [11], a Java-like textual language. The choice of this case is motivated by two reasons:

- It involves a transformation from an abstract visual language into a more concrete textual one, and hence it perfectly fits into the MDA philosophy.
- The semantics of both the source and target language (AD and TAAL) have been formally specified by means of graph transformation systems ([8] and [11], resp.).

The latter means that every AD model and every TAAL program give rise to a *transition system* modelling its execution. This in turn allows the application of standard concepts from concurrency theory in order to compare the executions and to decide whether they are indeed equivalent or not. Our aim is eventually to show *weak bisimilarity* between the transition system of any Activity Diagram and that of the TAAL program resulting from its transformation. Since weak bisimilarity is one of the most discriminating notions of behavioural equivalence (essentially preserving all properties in any reasonable temporal logic), we call this *full semantic preservation*.

Unfortunately, the size and complexity of the above problem are such that we have decided to first develop proof strategies for the intended result on a much more simplified version of the languages. In the current paper, we therefore apply the same question to two toy languages, inspired by AD and TAAL. Especially we model one non-trivial aspect: the token offer-based semantics of AD. Then, we solve the problem using two contrasting proof strategies.

The contribution of this paper lies in developing these two general strategies, carrying out the proofs for our example and afterwards comparing the strategies. Although simple, our example exhibits general characteristics of complex model-to-model transformations: different source and target languages, different levels of granularity of operational steps in the semantics and different labellings of steps. Our two proof strategies represent general approaches to proving semantics preservation of such model transformations.

The *first strategy* relies on a triple graph grammar-based definition of the model transformation (see [12,24]). Based on the resulting (static) triple graphs, we define an explicit bisimulation relation between the dynamic, run-time state graphs.

The *second strategy* relies instead on an in-situ definition of the model transformation and an extension of the operational semantics to the intermediate (hybrid) models. Using the theory of borrowed contexts (see [4]), we show that each individual model transformation step preserves the semantics.

The rest of the paper is structured as follows: Section 2 sets up a formal basis for the paper. Additionally, the source and target language and their respective semantics are introduced. Sect. 3 defines the model transformation, in both variants (triple graph grammar-based and in-situ). The actual proofs are worked out in Sections 4 and 5, respectively. Finally, Sect. 6 discusses and evaluates the results. Detailed proofs and additional information are contained in the extended version of this paper [10].

2 Definitions

2.1 Graphs and Morphisms

Definition 1 (Graph). A graph is a tuple $G = \langle V, E, src, tgt, lab \rangle$, where V is a finite set of nodes, E a finite set of edges, $src, tgt: E \rightarrow V$ are source and target functions

associating nodes with every edge, and $lab : E \rightarrow \text{Lab}$ is an edge labelling function. We always assume $V \cap E = \emptyset$.

For a given graph G , we use V_G, E_G etc. to denote its components. Note that there is a straightforward (component-wise) definition of union and intersection over graphs, with the caveat that these operators may be undefined if the source, target or labelling functions are inconsistent.

In example graphs, we use the convention that self-edges may be displayed through node labels. That is, every node label in a figure actually represents an edge from that node to itself, with the given label. We now define morphisms as structure-preserving maps between graphs.

Definition 2 (Morphism). *Given two graphs G, H , a morphism $f : G \rightarrow H$ is a pair of functions $(f_V : V_G \rightarrow V_H, f_E : E_G \rightarrow E_H)$ from the nodes and edges of G to those of H which are consistent with respect to the source and target functions of G and H in the sense that $src_H \circ f_E = f_V \circ src_G$, $tgt_H \circ f_E = f_V \circ tgt_G$ and $lab_H \circ f_E = lab_G$. If both f_V and f_E are injective (bijective), we call f injective (bijective).*

A bijective morphism is often called an *isomorphism*: if there exists an isomorphism from G to H , we call them *isomorphic*. A frequently used notion of graph structuring is obtained by *typing* graphs over a fixed *type graph*.

Definition 3 (Typing). *Given two graphs G, T , the graph G is said to be typable over T if there exists a typing morphism $t : G \rightarrow T$. A typed graph is a graph G together with such a typing morphism, say t_G . Given two graphs G, H typed over the same type graph (using typing morphisms t_G and t_H), a typed graph morphism $f : G \rightarrow H$ is a morphism that preserves the typing, i.e., such that $t_G = t_H \circ f$.*

Besides imposing some structural constraints over graphs, typing also provides an easy way to restrict to subgraphs:

Definition 4 (Type restriction). *Let T, U be graphs such that $U \subseteq T$, and let G be an arbitrary graph typed over T via $t : G \rightarrow T$. The restriction of G to U , denoted $\pi_U(G)$, is defined as the graph H such that*

- $V_H = \{v \in V_G \mid t(v) \in V_U\}$, $E_H = \{e \in E_G \mid t(e) \in E_U\}$,
- $src_H = src_G \upharpoonright_{E_H}$, $tgt_H = tgt_G \upharpoonright_{E_H}$ and $lab_H = lab_G \upharpoonright_{E_H}$.

The set of graphs with their morphisms form a category, which we will denote by Graph .

2.2 Graph Languages

In this paper we consider model transformation between two languages. In particular, we consider *graph languages*, i.e. sets of graphs; the models are the graphs themselves. We concentrate on a running example where there are two distinct, very simple graph languages denoted \mathcal{A} and \mathcal{B} . Fig. [1](#) shows type graphs for the languages, denoted $T_{\mathcal{A}}^{\text{st}}$ and $T_{\mathcal{B}}^{\text{st}}$, respectively. They describe the typing of the *static* parts of our two languages. We will sometimes also call these the (static) metamodels of the two languages. The

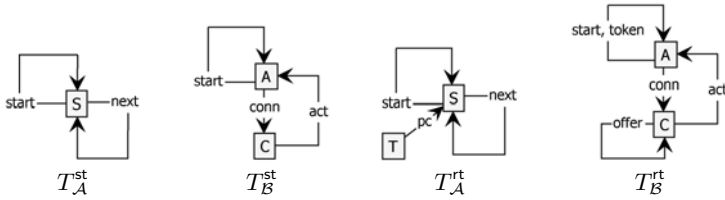


Fig. 1. Static (st) and run-time (rt) type graphs for graph languages \mathcal{A} and \mathcal{B}

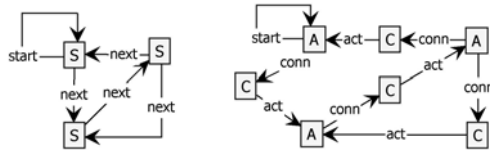


Fig. 2. Example graphs of languages \mathcal{A} (left) and \mathcal{B} (right)

figure also shows the corresponding extended *run-time* type graphs, which will be discussed below (Section 2.4).

The type graphs themselves impose only weak structure: not all graphs that can be typed over the \mathcal{A} - and \mathcal{B} -type graphs are considered to be part of the languages. Instead, we impose the following further constraints on the static structure:

Language \mathcal{A} consists of next-connected S -labelled nodes (*statements*). There should be a single S -node with a start-edge to itself, from which all other nodes are reachable (via paths of next-edges). Furthermore no next-loops are allowed.

Language \mathcal{B} consists of bipartite graphs of A - (*action*) and C -labelled (*connector*) nodes. Every C -node has exactly one incoming conn-edge and exactly one outgoing act-edge; the opposite nodes of those edges must be distinct. Like \mathcal{A} -graphs, \mathcal{B} -graphs have exactly one node with a start-self-edge, from which all other nodes are reachable (via paths of conn- and act-edges).

Small example graphs are shown in Fig. 2. We use $\mathcal{G}_{\mathcal{A}}^{st}$ ($\mathcal{G}_{\mathcal{B}}^{st}$) to denote the set of all well-formed (static) \mathcal{A} -graphs (\mathcal{B} -graphs).

2.3 Rules and Rule Systems

To specify the semantics of our languages, we have to formally describe changes on our graphs. This is done by means of *graph transformation rules*. A rule describes the change of (parts of) a graph by means of a before and after template (the left-hand and right-hand hand side of a rule); the interface fixes the part on which left and right hand side have to agree.

Definition 5 (Transformation rule). A graph transformation rule is a tuple $r = \langle L, I, R, \mathcal{N} \rangle$, consisting of a left hand side (LHS) graph L , an interface graph I , a right hand side (RHS) graph R , and a set $\mathcal{N} \subseteq \text{Graph}$ of negative application conditions (NAC’s), which are such that $L \subseteq N$ for all $N \in \mathcal{N}$. The interface I is the intersection of L and R ($I = L \cap R$).

We let *Rule* denote the set of rules. A rule (without a NAC) is basically a pair of injective morphisms in *Graph*: $L \leftarrow I \rightarrow R$. The diagram for a rule with NACs is this basic span together with injective morphisms from L to the elements of \mathcal{N} . For a single NAC N , a rule has the following form: $N \leftarrow L \leftarrow I \rightarrow R$. There are other definitions of graph transformation rules in the literature, the one used here is the one for double-pushout rewriting (DPO-rewriting).

A transformation rule $r = \langle L, I, R, \mathcal{N} \rangle$ is *applicable* to a graph G (called the *host graph*) if there exists an injective *match* $m: L \rightarrow G$ such that for no $N \in \mathcal{N}$ there exists a match $n: N \rightarrow G$ with $m = n \upharpoonright_L$ (i.e., all negative application conditions are *satisfied*), and moreover, the *dangling edge condition* holds: for all $e \in E_G$, $src(e) \in m(V_L \setminus V_I)$ or $tgt(e) \in m(V_L \setminus V_I)$ implies $e \in m(E_L \setminus V_I)$. This condition can be understood by realising that the elements of G that are in $m(L)$, but not in $m(I)$, are scheduled to be deleted by the rule, whereas the elements in $m(I)$ are preserved (see below). Hence we can not delete a node without explicitly deleting all adjacent edges.

Given such a match m , the *application* of r to G is defined by extending m to $L \cup R$, by choosing distinct “fresh” nodes and edges (outside V_G and E_G , respectively) as images for $V_R \setminus V_L$ and $E_R \setminus E_L$ and adding those to G . This extension results in a morphism $\bar{m}: (L \cup R) \rightarrow C$ for some extended graph $C \supseteq G$. Now let H be given by $V_H = V_C \setminus m(V_L \setminus V_R)$, $E_H = E_C \setminus m(E_L \setminus E_R)$, together with the obvious restriction of src_C , tgt_C and lab_C to E_H . The graph H is called the *target* of the rule application; we write $G \xrightarrow{r, m} H$ to denote that m is a valid match on host graph G , giving rise to target graph H , and $G \xrightarrow{r} H$ to denote that there is a match m such that $G \xrightarrow{r, m} H$. Note that H is not uniquely defined, due to the freedom in choosing the fresh images for $V_R \setminus V_L$ and $E_R \setminus E_L$; however, it is well-defined up to isomorphism.

Definition 6 (Rule system). A rule system is a partial mapping $\mathcal{R}: \text{Sym} \dashrightarrow \text{Rule}$. Here, *Sym* is a universe of rule names.

2.4 Language Semantics

In the context of the two languages defined in Section 2.2, we can use graph transformation rules for two separate purposes: to give a grammar that precisely and formally defines the languages or to specify the operational language semantics. In the latter case, the transformation rules describe patterns of state changes.

We will demonstrate the second usage here, by giving operational rules for \mathcal{A} -graphs and \mathcal{B} -graphs. This means that the graphs will represent run-time states. As we will see, this will involve auxiliary node and edge types that do not occur in the language type graphs. Fig. 1 shows extended type graphs T_A^{rt} and T_B^{rt} that include these *run-time* types. For \mathcal{A} , a T-node (of which there can be at most one) models a *thread*, through a single program counter (pc-labelled edge). For \mathcal{B} , we use token- and offer-loops which play a similar role; details will become clear below. Similar to the static part, we use $\mathcal{G}_A^{\text{rt}}$ ($\mathcal{G}_B^{\text{rt}}$) to denote the set of well-formed (run-time) \mathcal{A} -graphs (\mathcal{B} -graphs). The semantics of \mathcal{A} - and \mathcal{B} -models is defined in Fig. 3. Note that the figure shows the rules in DPO style, i.e. the middle part gives the interface I , and the sides are L and R , given as $L \leftarrow I \rightarrow R$. Additionally, NACs might be present.

We let $dom(\mathcal{R}_A) = \{\text{initA}, \text{movePC}\}$ and $dom(\mathcal{R}_B) = \{\text{initB}, \text{createO}, \text{moveT}\}$ be the names in the rule systems for the \mathcal{A} - and \mathcal{B} -models, the mapping to rules follows

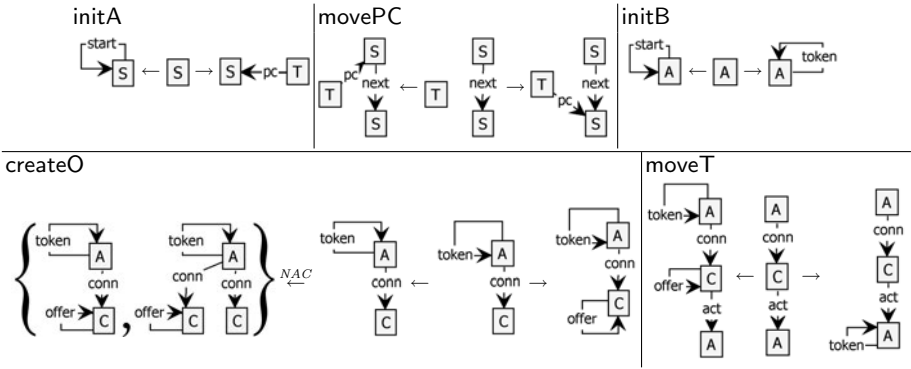


Fig. 3. Operational rules for \mathcal{A} (initA and movePC) and \mathcal{B} (initB, createO and moveT)

Fig. 3. Intuitively, the init-rules perform an initialisation of the run-time system, setting the program counter to the start statement (in \mathcal{A}) or putting a token onto a start action (in \mathcal{B}). Rule movePC simply moves the program counter to the next statement, createO moves an offer to a C-node and moveT moves the token. The semantics of \mathcal{A} - and \mathcal{B} -graphs is completely fixed by these rules, giving rise to a labelled transition system summarizing all these executions.

Definition 7 (Labelled transition system). An L -labelled transition system (LTS) is a structure $S = \langle Q, \rightarrow, \iota \rangle$, where Q is a set of states and $\rightarrow \subseteq Q \times L \times Q$ is a set of transitions labelled over some set of labels L . Furthermore $\iota \in Q$ is the start state.

In our case, the states are graphs and the transitions are rule applications. That is, given a rule system \mathcal{R} and a start graph G , we obtain a $dom(\mathcal{R})$ -labelled transition system by recursively applying all rules to all graphs. We will denote this transition system by $S(G)$ (leaving the rule system \mathcal{R} implicit). For instance, the LTS of an \mathcal{A} -graph G is $S(G) = (\mathcal{G}_{\mathcal{A}}^{\tau}, \rightarrow_{\mathcal{A}}, G)$, where $\rightarrow_{\mathcal{A}}$ is defined by the rules in $\mathcal{R}_{\mathcal{A}}$.

Semantic equivalence comes down to equivalence of the LTSs generated by two different graphs. There are several notions of equivalence over LTSs; see, e.g., [25]. In this paper, we use *weak bisimulation*. Weak bisimulation requires two states to mutually simulate each other, where a simulation may however involve internal (unobservable) steps. As usual, we use the special transition label τ to denote such internal steps.

For states $q, q' \in Q$ and a label α , we write $q \xrightarrow{\alpha} q'$ if $q \xrightarrow{\tau}^* \alpha \xrightarrow{\tau}^* q'$ and use $\xrightarrow{\varepsilon}$ to stand for $\xrightarrow{\tau}^*$. Furthermore, we define for (visible or invisible) labels α the following function $\hat{\cdot}$: $\hat{\tau} = \varepsilon$ and $\hat{\alpha} = \alpha$ if $\alpha \neq \tau$.

Definition 8 (Weak bisimilarity). Weak bisimilarity between two labelled transition systems S_1, S_2 is a relation $\approx \subseteq Q_1 \times Q_2$ such that whenever $q_1 \approx q_2$

- If $q_1 \xrightarrow{\alpha} q'_1$, then $q_2 \xrightarrow{\hat{\alpha}} q'_2$ such that $q'_1 \approx q'_2$;
- If $q_2 \xrightarrow{\alpha} q'_2$, then $q_1 \xrightarrow{\hat{\alpha}} q'_1$ such that $q'_1 \approx q'_2$.

We call S_1 and S_2 as a whole weakly bisimilar, denoted $S_1 \approx S_2$, if there exists a weak bisimilarity relation between S_1 and S_2 such that $\iota_1 \approx \iota_2$.

2.5 Semantics-Preserving Model Transformation

Our objective is to compare the LTSs of graphs of languages \mathcal{A} and \mathcal{B} . In Section 3 we will define a (relational) model transformation $MT \subseteq \mathcal{G}_{\mathcal{A}}^{st} \times \mathcal{G}_{\mathcal{B}}^{st}$ translating \mathcal{A} -graphs to \mathcal{B} -graphs. We aim at proving this model transformation to be *semantics preserving*, in the sense that the LTSs of source and target models are always weakly bisimilar.

However, there is an obvious problem: the LTSs of \mathcal{A} - and \mathcal{B} -graphs do not have the same labels, in fact $dom(\mathcal{R}_{\mathcal{A}}) \cap dom(\mathcal{R}_{\mathcal{B}}) = \emptyset$. Nevertheless, there is a clear intuition which rules correspond to each other: on the one hand the two initialisation rules, and on the other hand the rules `movePC` and `createO`. The reason for taking the latter two as corresponding is that both rules decide on where control is moving. The rule `moveT` has no matching counterpart in the \mathcal{A} -language, it can be seen as an *internal* step of the \mathcal{B} -language, completing a step initiated by `createO`. These observations give rise to the following renaming of the labels (i.e., the rule names) to a common set of names.

$$\begin{aligned} map_{\mathcal{A}} &: \text{initA} \mapsto \text{init}, & \text{movePC} &\mapsto \text{move} \\ map_{\mathcal{B}} &: \text{initB} \mapsto \text{init}, & \text{createO} &\mapsto \text{move}, & \text{moveT} &\mapsto \tau \end{aligned}$$

We call such a mapping $map: dom(\mathcal{R}) \rightarrow \text{Sym}$ (for a given rule system \mathcal{R}) *non-trivial* if it does not map every rule name to τ .

Definition 9 (Preservation of semantics). *Given two (graph) languages $\mathcal{G}_{\mathcal{A}}^{st}, \mathcal{G}_{\mathcal{B}}^{st}$, a model transformation $MT \subseteq \mathcal{G}_{\mathcal{A}}^{st} \times \mathcal{G}_{\mathcal{B}}^{st}$ is semantics-preserving if there are non-trivial mapping functions $map_{\mathcal{A}}: dom(\mathcal{R}_{\mathcal{A}}) \rightarrow \text{Sym}$, $map_{\mathcal{B}}: dom(\mathcal{R}_{\mathcal{B}}) \rightarrow \text{Sym}$ such that for all $G_{\mathcal{A}} \in \mathcal{G}_{\mathcal{A}}^{st}$, $G_{\mathcal{B}} \in \mathcal{G}_{\mathcal{B}}^{st}$ with $MT(G_{\mathcal{A}}, G_{\mathcal{B}})$*

$$map_{\mathcal{A}}(S(G_{\mathcal{A}})) \approx map_{\mathcal{B}}(S(G_{\mathcal{B}})) .$$

3 Model Transformation

Our model transformation needs to translate \mathcal{A} -models into \mathcal{B} -models. We will actually present two definitions of the transformation, both tailored towards the specific proof technique used for showing semantics preservation.

3.1 Triple Graph Grammars

Our first transformation uses triple graph grammars (TGGs). TGG rules [24,12] typically capture transformations between models of *different* types. Triple graphs can be separated into three subgraphs, typed over their own type graphs. Two of these subgraphs evolve simultaneously while the third keeps correspondences between them. For our example, we have the two type graphs $T_{\mathcal{A}}^{rt}$ and $T_{\mathcal{B}}^{rt}$ which — for forming a type graph for TGGs — are conjoined and augmented with one new correspondence G-node (the glue); see Fig. 4, resulting in a combined type graph $T_{\mathcal{A}\mathcal{B}}^{rt}$.

Normally, for a transformation, the source model is given in the beginning and is then gradually transformed. TGG rules however build two models simultaneously, matching each part of the source model to the target one. This allows to keep correspondences between transformed elements and to prove certain properties of the corresponding graphs. The TGG rules for the \mathcal{A} to \mathcal{B} transformation are given in Fig. 5.

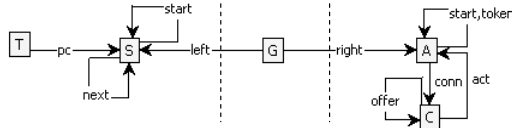


Fig. 4. Type graph T_{AB}^{rt} for TGG graph rules

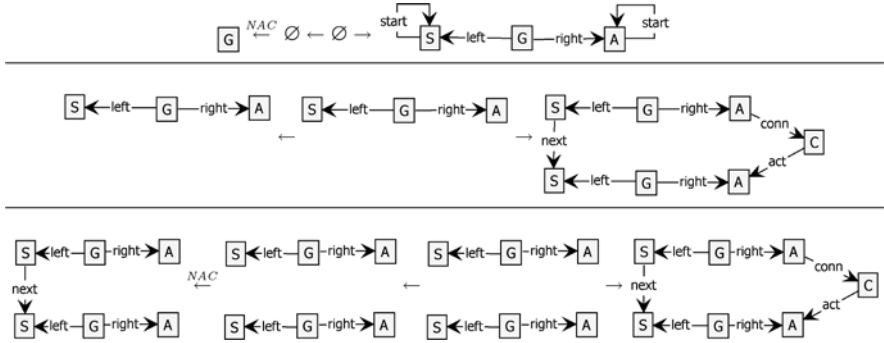


Fig. 5. TGG transformation rules

These rules incrementally build combined \mathcal{A} and \mathcal{B} -graphs. Initially, only the upper rule in Fig. 5 can be applied; it constructs one S- and one A-node connected via one correspondence G-node. The middle rule creates further S-, A- and C-nodes together with their correspondences; the lower rule simultaneously generates next-edges between S-nodes and connections via C-nodes between *corresponding* A-nodes. Let \mathcal{G}_{AB}^{rt} denote the set of graphs obtained by applying the three TGG rules on an empty start graph. To obtain the actual translation, restrict \mathcal{G}_{AB}^{rt} to the type graphs of \mathcal{A} and \mathcal{B} . Using the definition of type restriction as given in Section 2, the model transformation MT thus works as follows: Given an \mathcal{A} -graph G_A and a \mathcal{B} -graph G_B , we have $MT(G_A, G_B)$ exactly if there is some $G_{AB} \in \mathcal{G}_{AB}^{rt}$ such that $G_A = \pi_{T_A^{st}}(G_{AB})$ and $G_B = \pi_{T_B^{st}}(G_{AB})$.

3.2 In-Situ Transformation

Instead of building two models simultaneously, in-situ transformations destroy the source model while building the target model. This has the disadvantage of leading to “mixed” states, with components of both the source and the target model. This necessitates additional operational rules (see Section 5). On the other hand, in-situ transformations describe a clear evolution process. This is better suited as a basis for proof strategy 2, which relies on a congruence result for bisimilarity: the basic idea is that replacing a part of the model does not affect behavioural equivalence of the entire model.

We will now present the in-situ transformation rules, which are shown in Fig. 6. The first rule relabels nodes by replacing the label S by the label A. The second rule replaces a next-edge by a connection via a C-node. The third rule replaces the program

¹ Remember that labels are represented by loops on an unlabelled node.

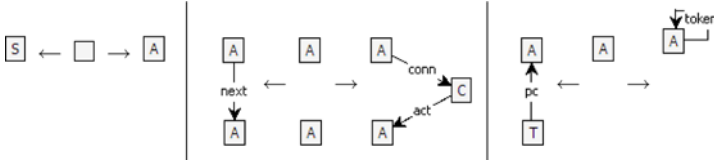


Fig. 6. In-situ transformation rules from language \mathcal{A} to language \mathcal{B}

counter by a token and allows the transformation of run-time models. We have reached a model in language \mathcal{B} as soon as no further rule applications are possible. We define that $MT(G_A, G_B)$ iff G_A is transformed into G_B via the rules in Fig. 6.

3.3 Comparison

In this section we argue that both strategies define the same model transformation. Assume that a graph G_A is transformed into a graph G_B via the TGG transformation of Section 3.1. This means that G_A and G_B are constructed simultaneously by the TGG grammar and arise as projections of a graph G_{AB} . Then we can apply the in-situ rules of Fig. 6 to G_A , obtaining the corresponding items of G_B .

The other direction is slightly more complicated. Assume that we are given a graph G_A of language \mathcal{A} . Then, with the TGG rules, we generate a graph G_{AB} which projects (via $\pi_{T_A^{\text{st}}}$) to G_A . We can then show, by induction on the length of this generating sequence and by using the fact that the transformation rules are confluent, that the graph $\pi_{T_B^{\text{st}}}(G_{AB})$ obtained in this way coincides with G_B , the graph generated by applying the in-situ transformation rules as long as possible.

4 Proof Strategy 1

In this section, we present our first approach to proving semantic preservation of the model transformation on all source models (for more details see the extended version [10]). This proof strategy uses the correspondences generated by the TGG rules, despite the fact that the semantic rules are applied on the individual models, based on the following two observations.

First observation: Both for \mathcal{A} and \mathcal{B} -models, the operational rules keep the syntactic, static structure of a model, except for start-edges: all S-nodes and next-edges, and all A, C-nodes and conn, act-edges stay the same.

To formulate structural correspondences, we introduce the following notation. For an S-node v_S and an A-node v_A , we write $\text{corr}(v_S, v_A)$ if there is a G-node v_G and a left-edge from v_G to v_S and a right-edge from v_G to v_A . For an edge e labelled label going from a node v to v' , we simply write $\text{label}(v, v')$. We also use these as predicates. The first result shows that correspondences between S and A-nodes are unique. Here, $\exists!$ stands for “there exists exactly one”.

Proposition 10. *Let $G \in \mathcal{G}_{AB}^{\text{rt}}$, v_S an S-node and v_A an A-node in G . Then the following two properties hold: (A) $\exists! v$ of type A such that $\text{corr}(v_S, v)$, and (B) $\exists! v$ of type S such that $\text{corr}(v, v_A)$.*

A number of further results show that (1) corresponding nodes either both or none have start-edges, and (2) next-edges between S-nodes will generate connections via C-nodes between corresponding A-nodes and vice versa.

Second observation: Correspondences between nodes in \mathcal{A} -models and \mathcal{B} -models are kept during application of semantic rules. Predicate *corr* as well as Prop. 10 and properties (1) and (2) can thus also be applied to separate \mathcal{A} and \mathcal{B} -graphs.

Theorem 11. *Let G_A^0, G_B^0 be an \mathcal{A} - and a \mathcal{B} -graph such that $MT(G_A^0, G_B^0)$. Then*

$$map_A(S(G_A^0)) \approx map_B(S(G_B^0))$$

For the proof, we need to construct a weak bisimulation relation \mathcal{R} (defining \approx) between the states of the first and the second LTS:

- $$\mathcal{R} = \{ (G_A, G_B) \in \mathcal{G}_A^{rt} \times \mathcal{G}_B^{rt} \mid \exists G_{AB} \in \mathcal{G}_{AB}^{rt} \}$$
- (1) $\pi_{T_A^{st} \setminus start}(G_A) = \pi_{T_A^{st} \setminus start}(G_{AB}) \wedge \pi_{T_B^{st} \setminus start}(G_B) = \pi_{T_B^{st} \setminus start}(G_{AB})$,
 - (2) \forall S-nodes v_S in G_A , A-nodes v_A in G_B s.t. $corr(v_S, v_A)$:
 $start(v_S)$ iff $start(v_A)$,
 - (3) \forall S-nodes v_S in G_A , A-nodes v_A in G_B s.t. $corr(v_S, v_A)$: $\exists v_T$ s.t. $pc(v_T, v_S)$
 iff (i) $token(v_A) \wedge \forall v_C$ s.t. $conn(v_A, v_C) : \neg offer(v_C)$ or
 (ii) $\neg token(v_A) \wedge \exists v_C, v'_A : token(v'_A) \wedge offer(v_C) \wedge conn(v'_A, v_C) \wedge act(v_C, v_A)$,
 - (4) $\exists v_T, v_S : pc(v_T, v_S) \iff \neg \exists v'_S : start(v'_S) \wedge \exists v_A : token(v_A) \iff \neg \exists v'_A : start(v'_A) \wedge \neg \exists v_A : start(v_A) \implies \exists ! v'_A : token(v'_A) \wedge \forall v_C : offer(v_C) \implies \exists v_A : token(v_A) \wedge conn(v_A, v_C) \wedge \neg \exists v_S : start(v_S) \implies \exists ! v'_S \text{ s.t. } \exists v_T : pc(v_T, v'_S) \}$

It contains all pairs of \mathcal{A} and \mathcal{B} -graphs which (1) in their static structure (except for start) still follow the structure generated by the TGG rules, (2) have start-edges only on corresponding nodes, (3) exhibit run-time properties only on corresponding nodes, and (4) obey certain well-formedness criteria for run-time elements.

Fig. 7 further illustrates condition (3). We have two possibilities for run-time elements in matching states: either the pc-edge is on an S-node and the token is on the corresponding A-node and no further offers exist (left), or the pc-edge is on a node for which the corresponding A-node has no token yet, but an offer has already been created and is ready to move the token to the A-node by means of the invisible step moveT

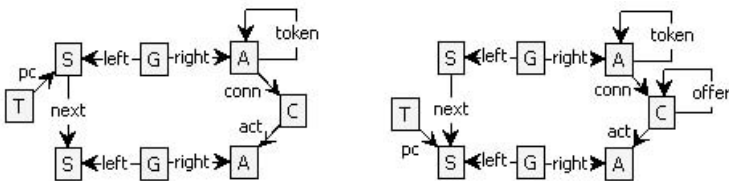


Fig. 7. Illustration of condition (3): Left (i), right (ii)

(right). We show that the relation \mathcal{R} is a weak bisimulation by proving that the states of transition systems can mimic each others moves. Due to space limitations we cannot give the full proof here, which can instead be found in the extended version [10].

5 Proof Strategy 2

5.1 The Borrowed Context Technique

In the following we will describe a different proof strategy, based on the borrowed context technique [4,21], which refines a labelled transition system (or even unlabelled reaction rules) in such a way that the resulting bisimilarity is a congruence [14]. Weak bisimilarity as in Def. 8 is usually not a congruence. By a congruence we mean a relation over graphs that is preserved by contextualization, i.e., by gluing with a given environment graph over a specified interface. This is a mild generalization of standard graph rewriting in that we consider “open” graphs, equipped with a suitable interface.

The basic idea behind the borrowed context technique is to describe the possible interactions with the environment. In addition to existing labels, we add the following information to a transition: what is the (minimal) context that a graph with interface needs to evolve? More concretely we have transitions of the form

$$(J \rightarrow G) \quad \alpha, (J \rightarrow F \leftarrow K), \mathcal{N} \quad (K \rightarrow H)$$

where the components have the following meaning: $(J \rightarrow G)$ is the original graph with interface J (given by an injective morphism from J to G) which evolves into a graph H with interface K . The label is now composed of three entities: the original label $\alpha = \text{map}(r)$ stemming from the operational rule r (as detailed in Section 2.5) and furthermore two injective morphisms $(J \rightarrow F \leftarrow K)$ detailing what is borrowed from the environment. The graph F represents the additional graph structure, whereas J, K are its inner and the outer interface. Finally we provide a set \mathcal{N} of negative borrowed contexts, describing negative constraints on the environment (see also [21]). We are using a saturated and weak version of bisimulation (see the extended version [10]).

5.2 Using Borrowed Contexts for Verification of Model Transformation

For in-situ model transformation within the same language, applications of the borrowed context technique are straightforward: show for every transformation rule that the left-hand and right-hand sides L, R with interface I are bisimilar with respect to the operational rules. Then the source model must be bisimilar to the target model by the congruence result. This idea has been exploited in [22] for showing behaviour preservation of refactorings.

However, in order to apply the idea above in our situation it is necessary to have an operational semantics also for “mixed” (or hybrid) models which incorporate components of both the source and the target model. Hence below we introduce such a mixed operational semantics, which has to satisfy the following conditions: (i) the mixed rules are *not* applicable to a pure source or target model; (ii) it is possible to show (borrowed context) bisimilarity of left-hand and right-hand sides of all transformation rules. Finally, observe that our final aim is to show bisimilarity of closed graphs, i.e., of graphs with empty interface. It can be shown that if all left-hand sides are connected, the notion of bisimilarity induced by borrowed contexts coincides with the standard one.

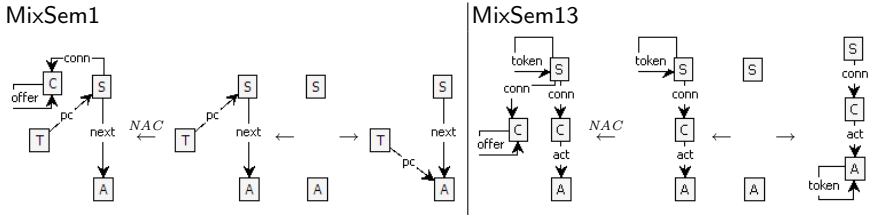


Fig. 8. Some rules of the operational semantics of mixed models

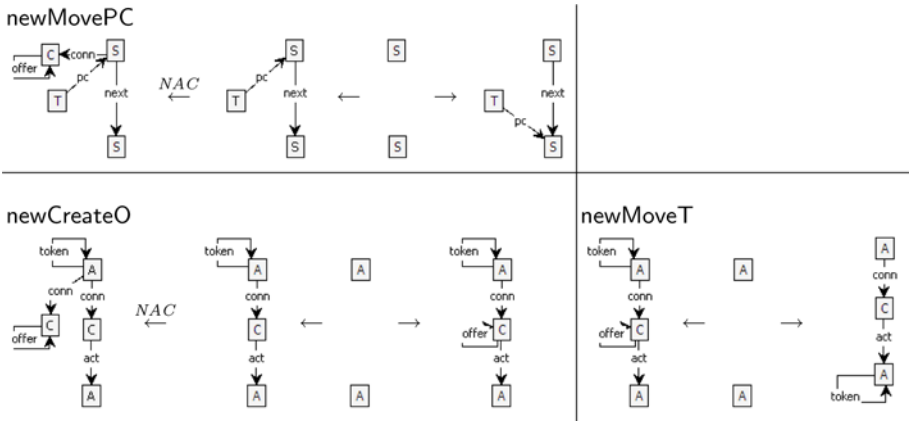


Fig. 9. Modified operational rules for the source and target languages

5.3 Rules of the Mixed Semantics

There are sixteen additional rules for the mixed semantics. Seven of them handle the behaviour of pc-edges at A-nodes, seven the semantics of the token-edge at an S-node and two of them are mixed counterparts to the initialization rules. Fig. 8 shows two examples of mixed rules, the rest are provided in [10]. Here we work with a single function *map* (see Section 2.5), both rules in Fig. 8 are mapped to move.

Furthermore, we modify some of the operational rules of Fig. 3: first, we equip several rules, also of the source semantics (language \mathcal{A}) with NACs (without changing the operational behaviour). Second, we restrict to a minimal interface by deleting and recreating the connections (see Fig. 9). Due to the layout of the graphs, this does not modify the semantics. Both modifications are needed to make the proof work and the latter modification is also very convenient since it allows us to derive fewer labels.

5.4 The In-Situ Transformation Preserves Weak Bisimilarity

Theorem 12. *The left-hand sides and right-hand sides of the three in-situ transformation rules in Fig. 6 are weakly bisimilar, with respect to the borrowed contexts technique, under the rules of the mixed semantics.*

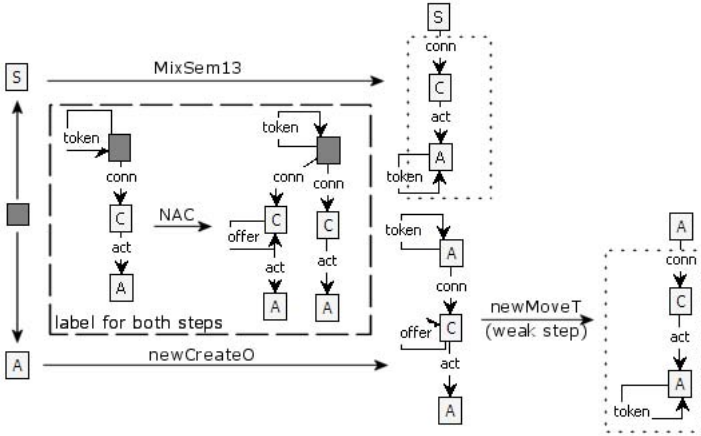


Fig. 10. Example of a label derivation using the borrowed context technique

Since weak bisimilarity is a congruence [10] and borrowed context bisimilarity coincides with standard bisimilarity (see Def. 8) on source and target models, this implies that $map(S(G_A)) \approx map(S(G_B))$ whenever $MT(G_A, G_B)$.

We give some intuition on the label derivation process by discussing one example, which needs the handling of weak moves and NACs (see Fig. 10).

In the labelled transition system, the graph consisting only of an S-node makes a move (with rule MixSem13) with the label shown in the (big) dashed box, i.e., it borrows a token, a C-node and an A-node. Spelling out the transition labels more concretely we have $\alpha = move$, F is the graph in the dashed box on the left (where the grey node represents both interfaces J, K) and the only NAC in \mathcal{N} is given on the right. The corresponding graph (the A-node) can answer this step with the same label, by making a step with rule newCreateO plus a weak step (τ) with rule newMoveT. After this second step, using an up-to-context proof technique, the same context (see dotted boxes) can be removed from both graphs, leaving the original pair of graphs already in the relation.

On the other hand, the answer to the newCreateO-step is with rule MixSem13. So the pair of graphs reached after one step has to be in the bisimulation as well and we have to check that they can mimic each others moves.

The entire bisimulation relation only contains five pairs, three are the in-situ transformation rules of Fig. 6 and two additional ones are needed. However, it is necessary to derive a large number of labels to prove that it is a bisimulation.

6 Discussion and Evaluation

Providing proof techniques for showing that full semantics preservation of a set of model transformation rules, for arbitrary source models, is a very difficult problem, on which there has been little work so far. The difficulty of the problem stems from three aspects: first, we need to show bisimilarity in transition systems based on graphs, a topic that has only recently started to receive attention; second, we do not only have to

prove bisimilarity for a given pair of start graphs, but for an infinite set of pairs of source and corresponding target models; and third, we want to address this on the level of a reusable proof *technique*, and not just a single proof for a given model transformation.

We feel that so far little progress has been made in tackling the inherent underlying difficulty. Hence it is our strong feeling that it is first necessary to consider case studies of modest size to clearly outline and evaluate possible solutions.

The case study that was chosen for this paper might seem small, but it already incorporates several non-trivial aspects: a heterogeneous setup (different source and target languages), negative application conditions and the need for weak bisimilarity, since one step in the source model has to be matched by two steps in the target model.

Our two proof strategies reflect two major possibilities: either to work out a direct proof manually – which could be verified with a theorem prover – or to use a semi-automatic method based on bisimulation proof theory.

Direct approach. The direct bisimulation proof based on triple graph grammars uses little additional theory and can be carried out by resorting to standard proof methodology. Because of that it is more flexible than the borrowed context technique and can deal with the rules of the operational semantics without modification.

Borrowed contexts. Here we extended the borrowed context technique to work with weak bisimilarity, which is a novel contribution. The technique seems to be easier to mechanize than the direct proof: the label derivation process can be done fully automatically and, at least in the case where a finite bisimulation up-to context exists, there are possibilities to find it via an algorithm as suggested in [9]. We also have some initial ideas for automatically generating the mixed semantics (by applying the transformation rules to the left-hand sides of the operational rules).

Summary. We were able to make both proofs work with a reasonable effort, but further work is necessary in order to make the approach scale. We conclude that additional techniques, in particular mechanisation, will be needed to address realistic languages such as the ones in [5]. However, we do see a lot of unused potential in exploiting bisimulation theory and congruence results as was done in the second proof strategy. In the future it will also be interesting to study refactoring cases, rather than transformations between distinct modelling languages: they promise to be easier, because they involve only a single operational semantics. Furthermore we have to consider whether weak bisimilarity is the appropriate behavioural equivalence in all instances.

Related work. The work closest to ours in its objective of showing semantic preservation for a transformation between models of different types is [6]. They present a mechanised proof of semantics preservation (wrt. some version of bisimilarity — the paper does not contain an explicit definition) for a transformation of automata to PLC-code, based on TGG rules. This proof faced some problems since it was not trivial to present graph transformation within Isabelle/HOL.

Although there is extensive work on the verification of model transformations, to our knowledge there are only few attempts to show that transformations will always transform source models into behaviourally equivalent target models. For instance, [11] discusses several proof techniques (bisimulation, model-checking), but does not really explain how they could be exploited to prove full semantics preservation.

As opposed to general model transformation, there has been more work on showing correctness of refactorings. The methods presented in [26,20,17,17] address behaviour preservation in model refactoring, but are in general limited to checking a certain number of models. The employment of a congruence result is also proposed in [3] which uses the process algebra CSP as a semantic domain. The techniques used in [15,23] mainly treat state-based models, using set theory and predicate logic to show equivalences. In [2] it is shown how to exploit confluence results for graph transformation systems in order to show correctness of refactorings. A number of approaches also focus on preserving specific aspects instead of the full semantics (see [16]).

Instead of generally proving correctness of a transformation, a number of approaches, also in the area of compiler validation, carry out run-time checks of equivalence between a given source and generated target model [17,18,13].

References

1. Barbosa, P.E.S., Ramalho, F., de Figueiredo, J.C.A., dos, S., Junior, A.D., Costa, A., Gomes, L.: Checking semantics equivalence of MDA transformations in concurrent systems. *The Journal of Universal Computer Science* 11, 2196–2224 (2009)
2. Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: A case study with BPEL. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 183–199. Springer, Heidelberg (2007)
3. Biztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
4. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. *MSCS* 16(6), 1133–1163 (2006)
5. Engels, G., Kleppe, A., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 94–109. Springer, Heidelberg (2008)
6. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards verified model transformations. In: Workshop on Model Development, Validation and Verification, pp. 78–93 (2006)
7. Gorp, P.V., Stenten, H., Mens, T., Demeyer, S.: Towards automating source-consistent UML refactorings. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 144–158. Springer, Heidelberg (2003)
8. Hausmann, J.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, University of Paderborn (2005)
9. Hirschhoff, D.: On the benefits of using the up-to techniques for bisimulation verification. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 285–299. Springer, Heidelberg (1999)
10. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Full semantics preservation in model transformation – a comparison of proof techniques. Technical Report TR-CTIT-10-09, CTIT, University of Twente (2010)
11. Kastenber, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
12. Königs, A.: Model transformation with triple graph grammars. In: Workshop on Model Transformations in Practice (2005)

13. Küster, J., Gschwind, T., Zimmermann, O.: Incremental development of model transformation chains using automated testing. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 733–747. Springer, Heidelberg (2009)
14. Leifer, J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)
15. McComb, T., Smith, G.: Architectural Design in Object-Z. In: ASWEC 2004, pp. 77–86. IEEE, Los Alamitos (2004)
16. Mens, T., Tourwé, T.: A survey of software refactoring. *IEEE Trans. Software Eng.* 30(2), 126–139 (2004)
17. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: GT-VMT 2006. ENTCS, vol. 211, pp. 185–194 (2006)
18. Necla, G.: Translation validation for an optimizing compiler. In: PLDI 2000. SIGPlan Notices, vol. 35, pp. 83–95. ACM, New York (2000)
19. Object Management Group: OMG Unified Modeling Language (OMG UML) – Superstructure, Version 2.2 (2009), <http://www.omg.org/docs/formal/09-02-02.pdf>
20. Pérez, J., Crespo, Y.: Exploring a method to detect behaviour-preserving evolution using graph transformation. In: Third International ERCIM Workshop on Software Evolution, pp. 114–122 (2007)
21. Rangel, G., König, B., Ehrig, H.: Deriving bisimulation congruences in the presence of negative application conditions. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 413–427. Springer, Heidelberg (2008)
22. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 242–256. Springer, Heidelberg (2008)
23. Ruhroth, T., Wehrheim, H.: Refactoring object-oriented specifications with data and processes. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 236–251. Springer, Heidelberg (2007)
24. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
25. van Glabbeek, R.: The linear time - branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
26. van Kempen, M., Chaudron, M., Kourie, D., Boake, A.: Towards proving preservation of behaviour of refactoring of UML models. In: SAICSIT 2005, pp. 252–259 (2005)

Specification and Verification of Model Transformations Using UML-RSDS*

Kevin Lano and Shekoufeh Kolahdouz-Rahimi

Dept. of Computer Science, King's College London

Abstract. In this paper we describe techniques for the specification and verification of model transformations using a combination of UML and formal methods. The use of UML 2 notations to specify model transformations facilitates the integration of model transformations with other software development processes. Extracts from three large case studies of the specification of model transformations are given, to demonstrate the practical application of the approach.

1 Introduction

Model transformations are mappings of one or more software engineering models (*source* models) into one or more *target* models. The models considered may be graphically constructed using languages such as the Unified Modelling Language (UML) [14], or can be textual notations such as programming languages or formal specification languages.

The concepts of Model-driven Architecture (MDA) [11] and Model-driven Development (MDD) use model transformations as a central element, principally to transform high-level models (such as Platform-Independent Models, PIMs) towards more implementation-oriented models (Platform-Specific Models, PSMs), but also to improve the quality of models at a particular level of abstraction.

We consider that the following properties are particularly important for model transformations:

Syntactic correctness. Can it be shown that a transformation maps correct models of the source language into correct models of the target language?

Definedness. Is the transformation applicable to every model of the source language?

Determinacy/Uniqueness. Does the specification define a unique target model from a given source model?

Rule completeness. Can the intended effect of a transformation rule can be unambiguously deduced from its explicit specification?

Language-level semantic correctness. Is there an interpretation χ from the source language $L1$ to the target language $L2$ such that, for any $M1$ and $M2$ which are models of $L1$ and $L2$ and are related by the transformation:

$$M1 \models \varphi \Rightarrow M2 \models \chi(\varphi)$$

for each sentence φ of $L1$, where $M \models \varphi$ denotes that φ is true in M ?

* Work carried out within the HoRTMoDA EPSRC project.

Confluence. Is the effect of the transformation independent of any alternative orders of application of transformation rules which are allowed by the specification?

In Section 2 we introduce the UML-RSDS specification approach for model transformations, and compare it with other approaches. Section 3 describes particular verification techniques for such specifications. In Section 4 we illustrate the use of UML-RSDS on the UML to relational database schema transformation, in Section 5 we give extracts from the specification of a transformation-based slicing tool. Section 6 describes a transformation from the UML 1.4 activity diagram notation to that of UML 2.2.

2 Specification Techniques for Model Transformations

A large number of formalisms have been proposed for the definition of model transformations: *declarative* approaches such as graph grammars [11] or the Relations notation of QVT [16], *hybrid* approaches such as ATL [3] and Epsilon [6], and *imperative* approaches such as Kermeta [5].

Ideally, any specification language for model transformations should support validation, modularity, verification, and the implementation of transformations. Modularity is the key property which supports the other three properties. Transformations described as single large monolithic relations cannot be easily understood, analysed or implemented. Instead, if transformations can be decomposed into appropriate smaller units, these parts can be (in principle) more easily analysed and implemented, and the analysis and implementation of the complete transformation can be composed from those of its parts.

UML-RSDS (UML Reactive System Development Support) is a subset of UML with a precise axiomatic semantics [9], [10] (Chapter 6). A UML-RSDS toolset supports the specification and analysis of systems in this subset, and the generation of executable code from specifications. UML-RSDS can be used to define model transformations in two alternative ways: (i) declaratively and abstractly using *constraints*, which express implicitly how two (or more) models are related, and what changes to one model need to be made (to preserve the truth of the constraints) when a change in another model takes place, or (ii) by using operations of metamodel classes to explicitly define how a target model is produced from a source model.

For example, the well-known tree to graph transformation (Figure 1) can be specified by two global constraints¹:

$$(C1) : t : Tree \text{ implies } \exists n : Node \cdot n.name = t.name$$

$$(C2) : t : Tree \text{ and } t.parent \neq t \text{ and} \\ n : Node \text{ and } n.name = t.name \text{ and} \\ n1 : Node \text{ and } n1.name = t.parent.name \text{ implies} \\ \exists e : Edge \cdot e.source = n \text{ and } e.target = n1$$

¹ $\exists x : C \cdot P$ abbreviates the OCL formula $C.allInstances() \rightarrow exists(x | P)$.



Fig. 1. Tree to graph transformation metamodels

This model expresses that there is a mapping between the tree objects in the source model and the node objects in the target model, and that there is an edge object in the target model for each non-trivial relationship from a tree node to its parent. The *identity* constraint means that tree nodes must have unique names, and likewise for graph nodes.

The constraints correspond directly to the informal requirements for the transformation. In the UML-RSDS approach, such constraints are implemented by identifying for each operation *op* of the system if *op* can potentially invalidate a constraint, either by making the antecedent true or the succedent false [8]. For operations that can affect the constraint the tool generates additional code for *op* which ensures the constraint is preserved.

In this case, a creation operation *createTree* will have additional code generated which creates a *Node* element for the new *Tree* object, and the operation *setparent* will have additional code to create a new *Edge* element if the parent of a tree node is set to be different to itself (and if matching graph nodes for the parent and tree node already exist).

Code generation directly from such high-level constraints may produce highly inefficient code, however. Instead, the constraints can be refined by defining explicit transformation rules as operations, specified by precondition and postcondition predicates, defining how particular elements of the source model are mapped to the target model. These operations are usually placed in new classes, external to the source or target metamodels, but referring to these metamodels.

In this example, the mapping from trees to graph nodes could be expressed by two explicit operations, using the OCL notation of UML:

```
mapTreeToNode(t : Tree)
```

```
post:
```

```
∃ n : Node · n.name = t.name
```

```
mapTreeToEdge(t : Tree)
```

```
pre: t ≠ t.parent and
```

```
t.name : Node.name and t.parent.name : Node.name
```

```
post:
```

```
∃ e : Edge · e.source = Node[t.name] and
e.target = Node[t.parent.name]
```

The notation $Node[x]$ refers to the node object with primary key (in this case name) value equal to x , it is implemented in the UML-RSDS tools by maintaining a map from the key values to nodes. In OCL it would be expressed as

$$Node.allInstances() \rightarrow select(name = x) \rightarrow any()$$

Likewise, $Node.name$ abbreviates

$$Node.allInstances() \rightarrow collect(name)$$

The explicit transformation rules generally permit more efficient implementation than the purely constraint-based specifications. They can be related to the global declarative form by showing, using reasoning in the axiomatic semantics (Chapter 6 of [10]) of UML-RSDS, that they do establish the constraints:

$$t : Tree \Rightarrow [mapTreeToNode(t)](\exists n : Node \cdot n.name = t.name)$$

and hence

$$[for\ t : Tree\ do\ mapTreeToNode(t)](\forall t : Tree \cdot \exists n : Node \cdot n.name = t.name)$$

because the individual applications of $mapTreeToNode(t)$ are independent and non-interfering. $[stat]P$ is the weakest precondition of predicate P with respect to statement $stat$ (Chapter 6 of [10]).

In the explicit specification approach individual transformation rules are specified as operations using pre/post pairs in the OCL notation. The precondition of the rule identifies when it is applicable, and to which elements of the source model. The postcondition identifies what changes to elements and connections should be made in the target model.

Rules are grouped into *rulesets*, which are UML classes. The attributes of a ruleset are common data used by its contained rules (operations). All rules in a ruleset can be applied in the same phase of a transformation. A ruleset has an algorithm or *application policy* which controls the order and conditions of application of its rules: this can be specified as a UML activity, state machine or other UML behaviour formalism, as the *classifierBehavior* of the ruleset class. In this paper we will use an abstract program notation which is a specific concrete syntax for a subset of UML structured activities. Since the ruleset algorithm is a UML *Behavior*, it can itself be specified by a *BehavioralFeature*, and by pre and post condition constraints. Ruleset verification checks that the composition of rules defined by the algorithm satisfies these constraints, using inferences based on the structure of the algorithm. For example, if $P \Rightarrow [r1]Q$ and $Q \Rightarrow [r2]R$ for rules $r1$ and $r2$, then $P \Rightarrow [r1; r2]R$ for the sequential combination of $r1$ and $r2$.

For the tree to graph transformation, the ruleset activity is defined by:

```
for t : Tree do mapTreeToNode(t) ;
for t : Tree do mapTreeToEdge(t)
```

This also defines the rule application order for the complete transformation.

Individual rules and rulesets can be reused for different transformations, and rulesets can be validated and verified independently of other parts of the transformation. At the same time, only standard UML and OCL notations are used to define transformations, improving reuse of transformations and the integration of transformations with other UML tools.

In terms of the properties discussed in the introduction, UML-RSDS supports the proof of syntactic correctness, semantic correctness, definedness, determinacy and confluence. Completeness checks upon individual rules are also supported. Syntactic correctness can be checked by translating rulesets into B machines and rules into operations of these machines. The proof of internal correctness of the machine will demonstrate that applications of the rules establish the properties of the target language, which are expressed as invariants of the machine. Semantic correctness can be shown by the use of inference rules for []. Definedness is shown by checking that rules are only invoked at points in a transformation where their precondition is true, and that expressions are always defined at the point where they are evaluated. For unbounded loops and recursions, proof of termination using variants is necessary. Determinacy is shown by analysing operation postconditions to check that they are unambiguous, and by establishing confluence for unordered loops.

UML-RSDS has the following advantages compared to other model transformation approaches:

- Standard UML and OCL is used, so that developers do not need to learn a new notation to specify model transformations.
- Transformations can be analysed by other UML tools, such as OCL checkers [18].
- The abstract specification level, using constraints, is inherently bidirectional [20], supporting transformation in both directions between two metamodels.
- Since model transformation specifications are themselves UML models, transformations can be applied to themselves, supporting reflection, a capability absent from most model transformation approaches [4].
- The notation has a formal semantics which supports verification.

UML-RSDS is not restricted to single-target, single-source transformations; mappings involving any number of metamodels can be defined.

3 Verification Techniques for Model Transformations

For UML-RSDS specifications of model transformations, syntactic correctness and language-level semantic correctness of individual rules can be verified by translating the specifications to the B formalism, following the set-theoretic definition of UML semantics given in [10].

For the tree to graph example, the transformation metamodels and operation *mapTreeToNode* can be formalised in B as:

```

MACHINE Tree2Graph
SEES String_TYPE
SETS Tree_OBJ; Node_OBJ
VARIABLES trees, nodes, name, nname
INVARIANT
  trees <: Tree_OBJ & nodes <: Node_OBJ &
  name : trees >-> String & nname : nodes >-> String
INITIALISATION
  trees, nodes, name, nname := {}, {}, {}, {}
OPERATIONS
  mapTreeToNode(t) =
    PRE t : trees
    THEN
      IF #n.(n : nodes & nname(n) = name(t))
      THEN skip
      ELSE
        ANY nx WHERE nx : Node_OBJ - nodes
        THEN
          nodes := nodes \ { nx } ||
          nname(nx) := name(t)
        END
      END
    END
  END
END

```

This translation is performed automatically by the UML-RSDS tool. Internal consistency proof of the machine in B then demonstrates syntactic correctness of the *mapTreeToNode* rule: that it maintains the identity constraint of names in the target model. More precisely, it shows that if the source model satisfies the source language constraints, and the existing target model satisfies the target language constraints, then applying the *mapTreeToNode* operation with a true precondition results in a target model which also satisfies the target language constraints.

The same procedure can be used to verify language-level semantic correctness: that source-language properties remain true, in interpreted form, in the transformed model. An example would be the property that a tree has no cycles under the *parent* relationship: this translates into an acyclicity property of the resulting graph.

Checks on definedness, uniqueness and rule completeness can be carried out by syntactic analysis of the rulesets and individual transformation rules. Definedness of a transformation τ is ensured if the termination condition $trm(S_\tau)$ of the B statement S_τ corresponding to τ is equivalent to *true* [7]. That is, $[S_\tau]true$ is *true*. In turn, this is ensured if the precondition of each operation is guaranteed to be true each time it is invoked within τ , if unbounded loops and recursions can be proved to terminate, and expression evaluations are well-defined. Uniqueness is ensured if all operations have deterministic postconditions, and if all unordered iterations are confluent. In the case of rule completeness, the checks include that for every object x created or modified in the target model by the rule, all data

features of x are explicitly set by the rule, unless their values can be deduced from default initialisations of the target language, or from derivation constraints from explicitly set features.

Confluence of a specification is the most complex property to establish. We can however define rules which show that bounded unordered loops are confluent in specific cases. A *for* $x : s$ *do acts* loop is a form of loop activity in UML structured activities. In the general case an execution of the loop consists of a collection of executions of $acts[v/x]$, one for each element v of s at the start of the loop. These executions may occur in any order and may be concurrent.

We assume that the loop body is a single operation call with its parameter ranging over s .

The inference rule: from

$$v : s \Rightarrow [acts(v)]P(v)$$

derive

$$[for\ v : s\ do\ acts(v)](\forall v : s@pre \cdot P(v))$$

is valid for such loops, provided that one execution of *acts* does not affect another: the precondition of each $acts(v)$ has the same value at the start of $acts(v)$ as at the start of the loop, and if $acts(v)$ establishes $P(v)$ at its termination, $P(v)$ remains true at the end of the loop.

Technically, we can say that the $acts(v)$ are *strongly confluent* with respect to P , permitting concurrent execution within the iteration statement *stat*: *for* $v : s$ *do* $acts(v)$, if, for $i \in \mathbb{N}_1$:

1. $P(v) \circ \downarrow(acts(v), j) \equiv P(v) \circ \downarrow(stat, i)$ for each $v \in s \circ \uparrow(stat, i)$ and $j \in Occ_{stat, i}(acts(v))$, the occurrences of $acts(v)$ within the occurrence i of *stat*.
2. $Pre_{acts(v)} \circ \uparrow(acts(v), j) \equiv Pre_{acts(v)} \circ \uparrow(stat, i)$ for each $v \in s \circ \uparrow(stat, i)$ and $j \in Occ_{stat, i}(acts(v))$.

$P \circ t$ means P holds at time t , $e \circ t$ is the value of e at time t , and $\uparrow(S, i)$ and $\downarrow(S, i)$ are the start and end times of the i -th occurrence of S [9].

Together these properties show complete non-interference between the separate invocation instances within the *for* loop, with regard to the truth of P on elements of s .

For the tree to graph transformation, both iterations over *Tree* satisfy strong confluence with regard to the respective postconditions of their iterated operations, which allows us to deduce that the constraints of this transformation are achieved by the *Tree2Graph* activity.

4 Case Study: UML to Relational Database Schemas

We have used UML-RSDS to specify the well-known transformation from UML class diagrams to relational database schemas. Our specification differs from existing transformations for this mapping, because we avoid the use of recursion

between rules. Instead a sequential ordering of separate transformation steps is used.

A simple transformation rule in this transformation is the introduction of a primary key, to a class which does not have one, using the metamodel of Figure 2:

introducePrimaryKey(*c* : *UMLClass*)

pre:

“persistent” : *c.stereotypes.name* and
c.ownedAttribute.stereotypes.name → *excludes*(“identity”) and
c.feature.name → *excludes*(*c.name* + “Id”)

post:

∃ *a* : *Property* ·
a.name = *c.name* + “Id” and
a : *c.ownedAttribute* and
a.type = *IntegerType* and
∃ *s* : *Stereotype* ·
s.name = “identity” and
a.stereotypes = *Set*{ *s* })

e@pre refers to the value of *e* at the start of the operation.

This transformation rule is defined in a metaclass, *TransformationRules*, in the metamodel. This metaclass represents the ruleset to which *introducePrimaryKey* (*c* : *UMLClass*) belongs. A behaviour can be attached to this class, to identify how the rule should be applied, and, in the case of several rules, in which order they should be applied.

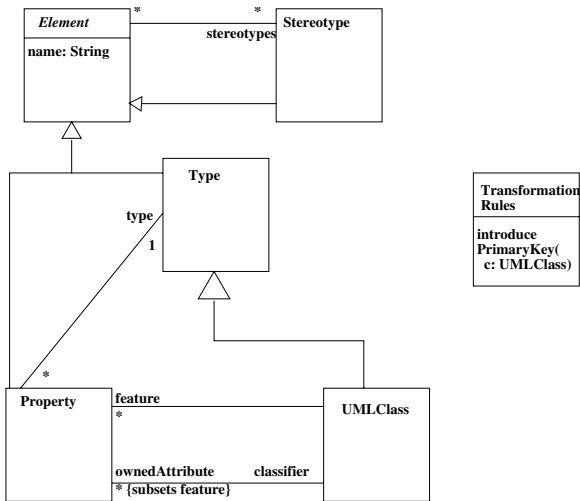


Fig. 2. Metamodel for primary key transformation

In this example, the rule should be applied by iterating it over all classes in the source model, in an arbitrary order:

```

introducePrimaryKeys()
for c : UMLClass
do
  if “persistent” : c.stereotypes.name and
    c.ownedAttribute.stereotypes.name → excludes(“identity”) and
    c.feature.name → excludes(c.name + “Id”)
  then
    introducePrimaryKey(c)

```

If inheritance has been removed from the model, then the separate iterations of *introducePrimaryKey* are independent and non-interfering, provided they do not overlap in their executions, so it can be deduced that the ruleset satisfies an overall pre-post specification defining the introduction of primary keys to each persistent class.

This ruleset is one step within the model transformation which maps a UML class diagram to a relational database schema (Figure 3).

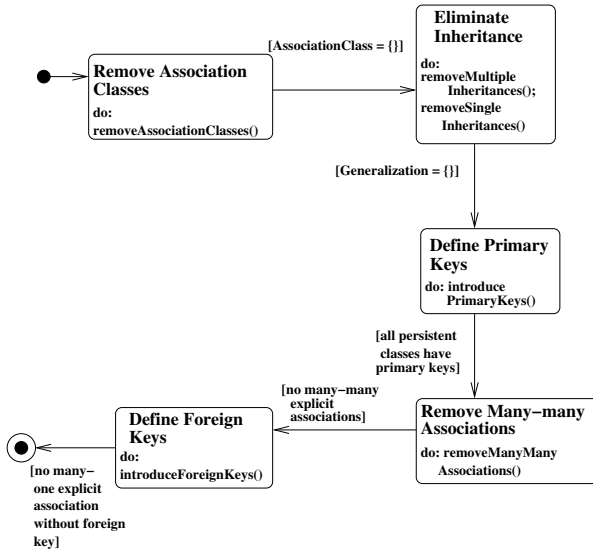


Fig. 3. Transformation algorithm

The correctness of a transformation with multiple phases can be demonstrated from the correctness of the individual phases. Syntactic correctness follows if there is a series of intermediate languages $\mathcal{L}_{I1}, \dots, \mathcal{L}_{In}$ such that each ruleset τ_i transforms from \mathcal{L}_{Ii} to \mathcal{L}_{Ii+1} , and so that the starting language for the algorithm combining these rulesets is \mathcal{L}_1 , and the final language is \mathcal{L}_2 .

In the example of Figure 3 the intermediate languages are subsets of the UML class diagram metamodel, which are successively reduced to smaller languages (without the metaclasses *AssociationClass*, *Generalization*, and with restricted

forms of association, etc) until a metamodel isomorphic to that of the relational data model can be used in the final step.

The semantic correctness of this algorithm can be shown by defining suitable state invariants. For example, the entire transformation preserves the property that no class has two different features with the same name, because each individual ruleset preserves this property. Only ruleset specifications are needed for this high-level verification, not the detailed specifications of individual rules.

The transformation specifications can be used directly as input to the code generation tool of the UML-RSDS toolset, which automatically synthesises Java code representing the metamodel (in a similar manner to Kermeta code) and code implementing the transformation rules and any algorithm for their combination.

For the introduce primary key rule, this code is, in part:

```
public void introducePrimaryKey(UMLClass c)
{ if (!(Element.getAllname(
    c.getstereotypes()).contains("persistent") &&
    !(Element.getAllname(
        Element.getAllstereotypes(
            c.getownedAttribute()).
            contains("identity"))))
    { return; }
    Property a = new Property();
    Controller.inst().addProperty(a);
    Controller.inst().addownedAttribute(c,a);
    Stereotype s = new Stereotype();
    Controller.inst().addStereotype(s);
    Controller.inst().setname(s,"identity");
    Controller.inst().setstereotypes(a,
        (new SystemTypes.Set()).add(s).getElements());
}
```

The implicit effects of the changes to *ownedAttribute* are explicitly coded in the definition of *addownedAttribute* in the generated code.

In contrast to other specifications of the UML to relational database transformation, our approach uses sequential composition of transformation steps, instead of recursive invocation of rules. This approach has the advantage of simplifying verification, and reducing dependencies between rules. The individual transformation steps can be reused in different contexts, independently of this transformation. New steps can also be added (for example, to remove qualified associations) without the need to modify the details of existing steps. This set of transformations has been incorporated into the UML-RSDS toolset.

5 Case Study: State Machine Slicing

Model transformations can be used to carry out the slicing of UML state machines. We have specified a toolset for state machine slicing, using model transformations defined in UML-RSDS. In this section we will give extracts from the

specification to illustrate the use of UML-RSDS for a substantial application of model transformations. In the formulation of Harman and Danicic [2] a slice is considered as a transformed version S of an artifact C which has a lower value of some complexity measure, but an equivalent semantics with respect to the sliced data:

$$S <_{syn} C \wedge S =_{sem} C$$

We use the following criteria for slicing a state machine M : $S <_{syn} M$ if S is syntactically smaller than M . $S =_{sem} M$ if for all input sequences e of events, starting from S and M in their initial states, and with the same initial values for their common variables, the state s of interest is reached by S as a result of the input sequence whenever it is reached by M as a result of the same sequence, and then the value of the variables V of interest in the state s of interest are the same in the two models.

This definition means that an analyser can deduce properties about M from properties of S , for properties which concern the values of V in s over all paths to s . In order to achieve independence of semantic variations of UML state machines, we can define *strict semantic equality* $S =_{sem}^{str} M$ which only requires the behaviour of S and M to be the same for input sequences e which always trigger explicit transitions in M at every step.

We will consider transformations which are valid under one or both of these semantics (if a transformation preserves $S =_{sem} M$ then also it preserves $S =_{sem}^{str} M$).

The following transformations are used to slice state machines:

- *removeStates*: Remove states (and their incoming and outgoing transitions) which cannot be reached from the initial state, or which have no outgoing path to the selected state of the slice.
- *sliceTransitions*: Slice transition actions to remove assignments which cannot affect the value of the variables of interest in the selected state.
- *deleteTransitions*: Delete transitions with a *false* guard.
- *mergeTransitions*: Merge two transitions which have the same sources, targets and actions. The guard of the resulting transition is the disjunction of the original guards.
- *replaceVariablesByConstants*: Replace a feature v by a constant value e throughout a state machine, if v is initialised to e on the initial transition of the state machine, and is never subsequently modified.
- *mergeStates*: Merge a group K of states into a single state k if the states are connected only by actionless transitions and all transitions which exit K are triggered by events distinct from any of the events that trigger internal transitions of K .

For example, transitions can be merged if their sources, targets and actions are the same: $tr1 : s1 \rightarrow_{op(x)[G1]/acts} s2$ and $tr2 : s1 \rightarrow_{op(x)[G2]/acts} s2$ can be replaced by:

$$tr : s1 \rightarrow_{op(x)[G1 \text{ or } G2]/acts} s2$$

Likewise, state merging can be used to reduce a set of states to a single state: A group K of states can be merged into a single state k if:

1. All transitions between the states of K have no actions.
2. All transitions which exit the group K are triggered by events distinct from any of the events that trigger internal transitions of K . If two transitions that exit K have the same trigger and different targets or actions, they must have disjoint guard conditions.
3. Each event α causing exit from K cannot occur on states within K which are not the explicit source of a transition triggered by α .

This transformation rule is only valid for the $S =_{sem}^{str} M$ semantics, an alternative state merging transformation is valid for the $S =_{sem} M$ semantics.

If the initial state is in a group g which is merged to a single state p , then p is initial in the new state machine.

Semantic correctness of the slicing transformation can be proved by analysis of the individual steps. For example, for the property

$$InitialState.allInstances() \rightarrow size() = 1$$

that there is a unique initial state, only the rulesets *removeStates* and *mergeStates* can fail to preserve this property, and so it is sufficient to establish their correctness with respect to the property, in order to verify the correctness of the entire algorithm.

6 Case Study: Model Migration of Activity Diagrams from UML 1.4 to UML 2.2

This transformation was one of the case studies for the 2010 transformation tool competition [19]. It involves the transformation of models of the UML 1.4 activity diagram language [12] into models of the UML 2.2 activity diagram language [15].

In UML 1.4 the language of activity diagrams was a variant of the state machine language. However in UML 2.2, a separate language is defined. The structure of these two languages are quite similar, so the transformation can be specified in a direct manner based on the structure of the source language.

Figure 4 shows a screen shot of the UML-RSDS system with the transformation source metamodel on the left, and the target metamodel on the right.

We assume that there is only one composite state in the source model, the *top* state of the model, an OR-composite state (Page 2-158 of the UML 1.4 superstructure specification [13]). Then *top.subvertex* in the source model is interpreted by *node* in the target model.

The transformation is mainly a direct mapping from source language meta-classes and features to corresponding target language meta-classes and features. The only parts requiring significant logic in the transformation are (i) the mapping of different kinds of pseudostate to different kinds of activity nodes, and (ii)

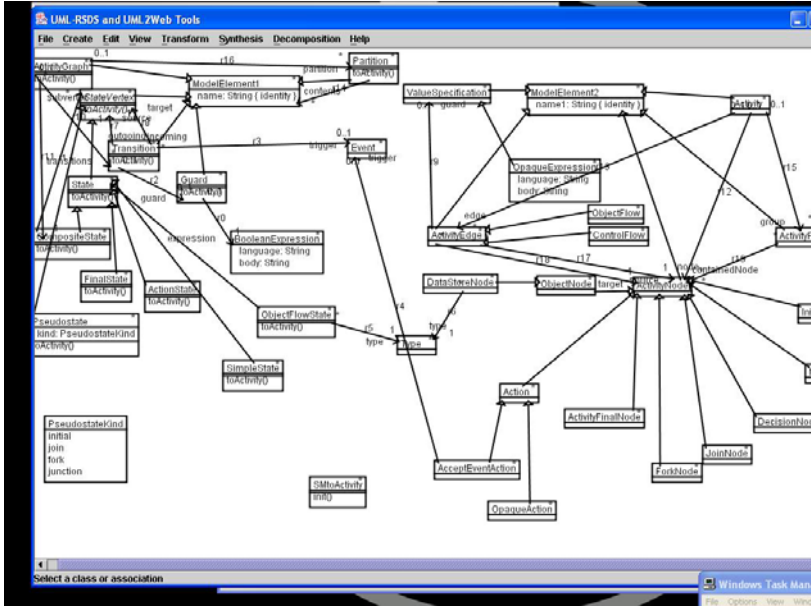


Fig. 4. Model Migration case study metamodels

the mapping of transitions to control or object flows, depending on what state vertices they connect, and of transitions with triggers to control nodes which receive the trigger event or signal.

The transformation can be formalised by a set of constraints, which define how the source and target models are related. For example, the correspondence of final states and activity final nodes could be defined by a constraint $C1$:

$$f : FinalState \text{ implies } \exists n : ActivityFinalNode \cdot n.name = f.name$$

Likewise for the other source language metaclasses and features. We assume that *name* is a primary key (identity) attribute for the source model elements. An artificial element id attribute could be introduced instead if *name* is not an identity attribute.

Because of the similarity in structure between the source and target metamodels, the transformation can be specified by rules placed in the source language metaclasses, one per metaclass, to define how that entity should be mapped into the target model. This makes the transformation easy to understand and modify, and also facilitates the use of inheritance.

For states, we have six operations, one for each concrete subclass of *StateVertex*. For example, in *ObjectFlowState*:

toActivity()

post:

$$\exists n : DataStoreNode \cdot n.name = name \text{ and } n.type = type$$

It is assumed that UML 1.4 types can be mapped without change into UML 2.2 types. Similar operations are defined in *FinalState* and *ActionState*.

In *Pseudostate* we define:

toActivity()

post:

(*kind* = *initial* implies $\exists n : \text{InitialNode} \cdot n.\text{name} = \text{name}$) and
 (*kind* = *join* implies $\exists n : \text{JoinNode} \cdot n.\text{name} = \text{name}$) and
 (*kind* = *fork* implies $\exists n : \text{ForkNode} \cdot n.\text{name} = \text{name}$) and
 (*kind* = *junction* and *incoming.size* = 1 implies
 $\exists n : \text{DecisionNode} \cdot n.\text{name} = \text{name}$) and
 (*kind* = *junction* and *incoming.size* > 1 implies
 $\exists n : \text{MergeNode} \cdot n.\text{name} = \text{name}$)

A junction state is mapped to a decision node if it has one incoming transition, otherwise to a merge node.

A simple state is assumed to be used in an activity diagram in order to wait for an event to occur (since action states cannot have triggers on their outgoing transitions, page 3-159 of [13]). These states are therefore mapped to *AcceptEventAction* instances:

toActivity()

post:

outgoing.size = 1 and *outgoing.trigger.size* = 1 implies
 $\exists n : \text{AcceptEventAction} \cdot n.\text{name} = \text{name}$ and
n.trigger = *outgoing.trigger*

in *SimpleState*.

Transitions are mapped to particular activity edges:

toActivity()

pre:

source.name : *ActivityNode.name* and
target.name : *ActivityNode.name*

post:

(*source* : *ObjectFlowState* or *target* : *ObjectFlowState* implies
 $\exists f : \text{ObjectFlow} \cdot f.\text{name} = \text{name}$ and
f.source = *ActivityNode[source.name]* and
f.target = *ActivityNode[target.name]* and
f.guard = *OpaqueExpression[guard.name]*) and
 (*source* /: *ObjectFlowState* and *target* /: *ObjectFlowState* implies
 $\exists f : \text{ControlFlow} \cdot f.\text{name} = \text{name}$ and
f.source = *ActivityNode[source.name]* and
f.target = *ActivityNode[target.name]* and
f.guard = *OpaqueExpression[guard.name]*)

A transition is mapped to an object flow if it has source or target in *ObjectFlowState*, otherwise to a control flow.

The notation *Classname*[*pkset*] can be used to obtain a set of objects of *Classname*, from a set *pkset* of primary key values. For example *OpaqueExpression*[*guard.name*] abbreviates

OpaqueExpression.allInstances() → *select(oe | oe.name : guard.name)*

The partitions of the source model are mapped to activity partitions of the target model, with corresponding contents. Finally, activity graphs are mapped to activities:

```

toActivity()
post:
  ∃ a : Activity · a.name = name and
    a.node = ActivityNode[top.subvertex.name] and
    a.group = ActivityPartition[partition.name] and
    a.edge = ActivityEdge[transitions.name]

```

The overall algorithm is specified as the activity of the transformation meta-class *SMtoActivity*:

```

init() ;
CompositeState.toActivity() ;
Guard.toActivity() ;
Transition.toActivity() ;
Partition.toActivity() ;
ActivityGraph.toActivity()

```

In general, the mapping of subordinate parts of an element must be performed before the mapping of the element itself.

7 Comparison with Other Approaches

Model transformation specification approaches are divided between those which are declarative in style, and often implicit in their execution, and those that are imperative and explicit. ATL [3], VIATRA [17] and QVT-R [16] are in the first category, whilst Kermeta [5] is in the second. VIATRA and UML-RSDS combine explicit and implicit aspects, providing control expressions to define specific execution orders of transformation steps. Kermeta and UML-RSDS allow the implicit specification of changes to inverse association ends: these changes are deduced from the explicit modifications of one end. UML-RSDS also deduces changes to association ends which generalise the modified end. Iterations over unordered collections may occur in any order (the *for* loop in UML-RSDS, *forall* in VIATRA and *each* iteration in Kermeta). ATL and QVT-R avoid explicit definition of the order of execution of rules: the implementation of the rules is determined by the transformation tools and not by the specifier. Implicit specifications have the advantage that they are usually more concise, and that they require the specifier only to specify the essential parts of a transformation. However, explicit specifications allow the specifier to have greater control over the details of the transformation. Control over the order of execution of rules is often particularly important to enable efficient implementation of the transformation, so it is a desirable feature of a model transformation language that it supports the explicit definition of rule ordering. If implicit ordering is used, then there

should be tool support to check that specifications are confluent, that is, different possible execution orders for rules do not change the meaning of the specification. ATL provides a runtime check to detect possible ambiguities of this kind.

References

1. Ehrig, H., Engels, G., Rozenberg, H.-J. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2. World Scientific Press, Singapore (1999)
2. Harman, M., Binkley, D., Danicic, S.: Amorphous Program Slicing. *Journal of Systems and Software* 68(1), 45–69 (2003)
3. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
4. Jouault, F., Kurtev, I.: On the interoperability of model-to-model transformation languages. *Science of Computer Programming* 68, 114–137 (2007)
5. Kermeta (2010), <http://www.kermeta.org>
6. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) *ICMT 2008*. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
7. Lano, K.: *The B Language and Method*. Springer, Heidelberg (1996)
8. Lano, K.: Constraint-Driven Development. *Information and Software Technology* 50, 406–423 (2008)
9. Lano, K.: A Compositional Semantics of UML-RSDS. *SoSyM* 8(1), 85–116 (2009)
10. Lano, K. (ed.): *UML 2 Semantics and Applications*. Wiley, Chichester (2009)
11. OMG, Model-Driven Architecture (2004), <http://www.omg.org/mda/>
12. OMG, UML Specification, version 1.4 (2001), <http://www.omg.org/spec/UML/1.4/>
13. OMG, UML Superstructure Specification, version 1.4. OMG document 01-09-67 (2001), <http://www.omg.org/spec/UML/1.4/>
14. OMG, UML superstructure, version 2.1.1. OMG document formal/2007-02-03
15. OMG, UML Specification, version 2.2 (2007), <http://www.omg.org/spec/UML/2.2>
16. OMG, Query/View/Transformation Specification, ptc/05-11-01 (2005)
17. OptXware, *The Viatra-I Model Transformation Framework Users Guide* (2010)
18. Richters, M.: *A UML-based Specification Environment* (2001), <http://www.db.informatik.uni-bremen.de/projects/USE>
19. Rose, L., Kolovos, D., Paige, R., Polack, F.: *Model Migration Case for TTC 2010*, Dept. of Computer Science, University of York (2010)
20. Stevens, P.: Bidirectional model transformations in QVT. *SoSyM* 9(1) (2010)

Multiformalism and Transformation Inheritance for Dependability Analysis of Critical Systems

Stefano Marrone¹, Camilla Papa², and Valeria Vittorini²

¹ Seconda Università di Napoli, Dipartimento di Matematica
via Vivaldi, 43, 81100 - Caserta, Italy
`stefano.marrone@unina2.it`

² Università di Napoli “Federico II”, Dipartimento di Informatica e Sistemistica
Via Claudio 21, 80125 Napoli, Italy
`{camilla.papa,valeria.vittorini}@unina.it`

Abstract. Multiformalism approaches and automatic model generation are challenging issues in the context of the analysis of critical systems for which formal verification and validation are mandatory. Reusable model transformations may reduce the skill level required in formal modeling, time and cost of the analysis process, and they may support the integration among different formal languages. This paper investigates how the relationship existing between different classes of formal languages may be exploited to define new model transformations by extending existing definitions. Specifically, the inheritance relationship is considered with the ultimate goal of achieving formalisms integration also by developing proper reusable model transformations. This idea is applied to the integration between Repairable Fault Trees and Generalized Stochastic Petri Nets, where the inheritance relationship between Fault Trees and Repairable Fault Trees is the basis to define inheritable model transformations. The described techniques are demonstrated on the availability model of a modern railway controller.

Keywords: Multiformalism, Model Transformation, Language Inheritance, Model Composition, System Availability.

1 Introduction

In order to assess the compliance of critical (computer-based) systems with the international safety and dependability standards, the usage of formal methods is sometimes strictly required. Modeling languages such as Fault Trees (FT) [15] and Generalized Stochastic Petri Nets (GSPN) [18] have been widely used, for example, to evaluate the occurrence probability of unsafe events and perform risk analysis. Formal methods are one of the most advocated techniques in this context but their effective usage in industrial settings is limited by time and cost efforts in developing *complex models of complex systems*: the need of proper skills, the lack of agile modeling methodologies, the difficulty related to model validation, the shortage of really user friendly modeling and analysis tools make their use hard.

Moreover, when complexity and heterogeneity of systems and applications increase, a single formalism is not more able to capture all important concerns. To cope with heterogeneity and address different views and aspects of such systems, models must be expressed by using different formal languages: this requires a considerable effort in formalisms integration that is one of the research objectives of the *multi-formalism* approaches.

Within this scope, automatic generation of models according to Model Driven Engineering (MDE) techniques and the definition of strategies and tools enabling model reuse may be a way to achieve a more agile approach to multiformal model development. The work described in this paper is part of the wider ongoing research project OsMoSys (Object-based multi-formaliSm MOdeling of SYStems) [23], [12], in which a new thread has been recently opened to explore the synergies between model transformations and formal languages. Specifically, we are investigating the application of MDE to the automatic generation of dependability models and the role that model transformations may play in presence of hierarchies of modeling languages. The contribution of this paper is twofold: to demonstrate that model transformations reuse may be achieved by exploiting inheritance relationship between classes of formal languages and to provide a practical engineering approach to the design and the development model transformations within hierarchies of formal languages.

The concepts and the techniques described in this paper are applied to the integration of Repairable Fault Trees (RFT) [19] and GSPN, where the inheritance relationship between FT and RFT is the basis to define an derived model transformation. The case described in the paper suggests that two advantages can be gained from this approach: 1) both model and transformation reuse will be improved, 2) extensible model transformations in formal language hierarchies contribute to the definition of a more usable modeling methodology.

The effectiveness of the proposed techniques is shown by applying them to a RFT model of the Radio Block Centre (RBC), the vital core of the European Railway Traffic Management System/European Train Control System (i.e., the reference standard of the new European railway signalling and control systems [22]).

The paper is organized as follows. Section 2 introduces languages hierarchies and discusses how model transformations may act in presence of such hierarchies. In Section 3 a FT-GSPN transformation is realized and used to derive the RFT-GSPN transformation according to the discussion presented in Section 2; Section 3 also contains a brief description of the RFT formalism. Section 4 illustrates the application of proposed techniques to the RBC case study. In Section 5 a review of related works is provided. Finally, Section 6 contains some closing remarks and directions for further works.

2 Transformations in Hierarchies

Several approaches have been proposed for multi-formalism modeling: a brief review of these solutions can be found in [10]. Multi-formalism techniques are

very appealing in modeling complex systems since they allow to build complex models by integrating or composing sub-models specified by different formal languages. Since each formalism is usually coupled with efficient solution techniques, proper approaches and tools must be adopted to solve a multi-formalism model. The OsMoSys methodology (which is focused on component based model development and multi-formalism) defines a conceptual framework based on object orientation concepts and meta-modeling in which the modeling languages are introduced by means of meta-classes called *formalisms*. Table 1 shows the four layers of meta-modeling on which the OsMoSys approach is based [23], [12].

Table 1. The OsMoSys modeling stack

Level	OsMoSys layer	Description
M3	Meta-formalisms	Languages to define formalisms.
M2	Model Meta-classes	Formalisms to build models
M1	Model Classes	Model specifications
M0	Model Objects	Model instances

In the OsMoSys framework new formalisms can be easily defined by inheritance from existing ones. Fig. 1 shows a situation in which a hierarchy expresses such relationship among languages. In this example both Fault Trees and Petri Nets can be defined in terms of graph concepts. Hence, they both inherit from a Graph formalism used to define the basic elements of all graph-based languages, for example Node and Edge.

Within the Petri Nets hierarchy in Fig. 1, the Place and Transition elements of the Petri Net formalism extend the Node element of the Graph formalism,

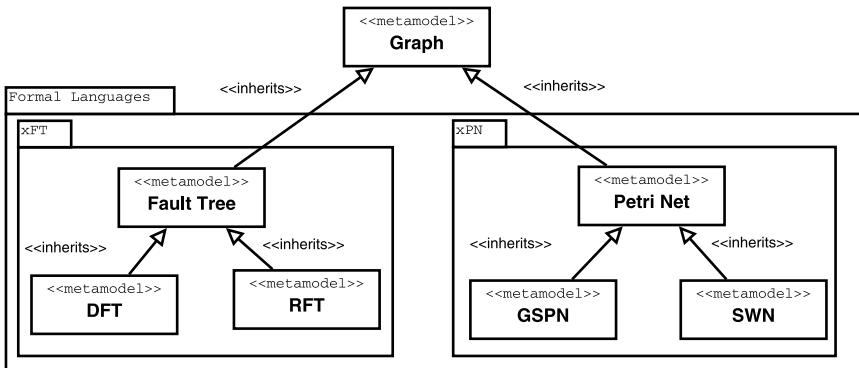


Fig. 1. An example of formalism hierarchy

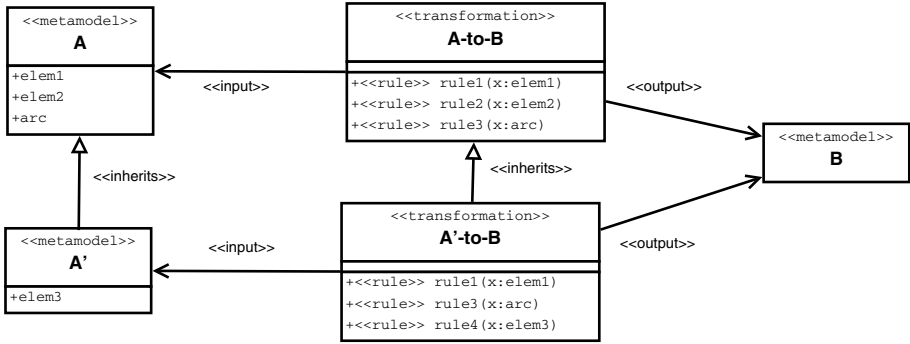


Fig. 2. Inheritance in transformations

and the Arc element inherits from the Edge element. The inherited elements add to the parent elements some properties (e.g. marking for Place, weight for Arc, etc.). Similarly, the GSPN formalism is obtained by extending Petri Nets with temporal specification. Timed and Immediate Transition elements extend the Transition element of Petri Nets by associating a firing delay to timed transitions and a null-delay to immediate transitions.

Now we can better state the problem at the basis of this work. With reference to Fig. 2, $A \leftarrow A'$ is a hierarchy, where the formalism A' inherits from A . As an example, A' extends A by adding the element $elem3$. Let us suppose that a model transformation $A - to - B$ is defined from the source formalism A to a target formalism B and it is available for reuse. We want to investigate the existence of the $A' - to - B$ transformation and the relationship between $A - to - B$ and $A' - to - B$ induced by the inheritance relation existing between A and A' . In the following we will refer to rule-based transformation languages, according to the taxonomy introduced in [17]. This means that a transformation consists of a set of rules triggered by elements belonging to the source language and producing elements in the target language.

The exact meaning of inheritance between formalisms should be defined in order to exploit the relationship between model transformations in formalisms hierarchies. At the state of our work, we restrict our considerations to the following cases:

1. *addition*: the derived formalism (A') extends the parent formalism (A) by adding a new element (e.g. $elem3$);
2. *re-definition*:
 - a. *new context*: an element defined by the parent formalism A is inherited without changes but it may be used within a new context in the derived formalism A' ;
 - b. *old context*: an element defined by the parent formalism A assumes a new meaning within the derived formalism A' , but the context in which it is used does not change;

3. *full reuse*: an element defined by the parent formalism A is inherited by A' without modifications.

With respect to the cases above, the model transformation $A' - to - B$ may be built by “inheriting” from $A - to - B$ as follows:

1. *addition*: a new rule must be defined that maps the new source element onto the target one. In Fig. 2 this is the case of $rule4(x : elem3)$.
2. *re-definition*:
 - a. *new context*: the subset of rules in $A - to - B$ transformation which is related to the translation of inherited elements is extended and/or partially redefined in $A' - to - B$. In Fig. 2 this could be the case of the *arc* element. $rule3(x : arc)$ must be redefined so that it triggers if an *arc* must connect *elem1* to *elem3*.
 - b. *old context*: the subset of rules in $A - to - B$ transformation which is related to the translation of inherited elements must be redefined. This is the case of *elem1* in Fig. 2, where $rule1(x : elem1)$ is redefined in $A' - to - B$;
3. *full reuse*: the subset of rules in $A - to - B$ transformation which is related to the translation of inherited elements is re-used. This means that such rules are inherited from $A - to - B$ to $A' - to - B$. This is the case of $rule2(x : elem2)$ in Fig. 2.

In Section 3 the cases 2.a and 3 will be applied. In order to better understand the real application of the case 2.b., let us consider an example of language inheritance where a derived formalism extends an element defined in the parent formalism by adding some properties. In this case, a derived rule for such element must define the proper mapping only for new properties.

Two main *mechanisms* have been defined in the literature for transformation composition, that here we propose to use for defining transformation inheritance: *superimposition* and *rule overriding*. Superimposition allows for overlaying several transformation definitions and executing them as they were a single transformation [13]. In other word superimposition builds a new transformation by union of the sets of rules of all involved transformations. From our point of view, superimposition makes possible that a transformation inherits all the rules of one or more superimposed transformations. Within superimposition mechanism, rule overriding allows to substitute an existing rule by a new one with the same name. Of course we use rule overriding to change rule implementations in derived transformations without changing their matching elements.

3 The Repairable Fault Trees Transformation

In this Section the concepts introduced above are applied to define the $RFT - to - GSPN$ transformation by inheritance from a $FT - to - GSPN$ transformation. Although both the transformations have been already defined from a theoretical point of view in [7] and [9] respectively, the contribution here given

is deeply different. Indeed, the pure theoretical definition of the transformations themselves is not the goal of this paper. This paper addresses the formalisms integration perspective within the multi-formalism modeling, the methodological approach to model transformations inheritance and reuse based on the formal language relationship, the definition of a practical method to the concrete realization of the model transformations based on widely used MDE techniques and tools.

Hence, this Section starts with a brief introduction to RFT, then it illustrates the rules set of the FT-to-GSPN transformation, and then continues by defining and illustrating the steps needed to provide a concrete method to inherit and implement the RFT-to-GSPN transformation.

3.1 Repairable Fault Trees

The RFT formalism was introduced to ease the modeler's approach to complex repair policy modeling and evaluation [19] as a result of the application of the OsMoSys multi-formalism multi-solution methodology [23]. Repairable Fault Trees preserve the modeling simplicity of FTs and allow to exploit the expressive power of Petri Nets by implementing where possible an efficient divide-et-impera solving process [19]. At the state, RFTs allow to model a series of complex maintenance policies and extend the well known FT formalism by adding a new element, called Repair Box (RB), that is able to take into account:

- which fault condition will start a repair action (trigger event);
- a repair policy, including the repair algorithm, the repair timing and priority, and the number of repair facilities;
- the set of components in the system that are actually repairable by the RB.

Graphically, a RFT model is a simple FT with the addition of the RBs. The FT is obtained exactly as for usual FT models, then RBs are added to implement repair actions. A RB is connected to the tree by arcs linking the trigger event to the RB and the RB to all the Basic Events in the FT (that are the elementary events, at the bottom of the tree) on which the repair operates. The RFT model of a system can be obtained in two steps. First, the FT of the system is built by inspection of its structure; then the chosen repair policies are applied to the model by evaluating which conditions will trigger the repair and on which sub-tree the repair will be applied.

The RB node encapsulates a GSPN model of the repair action. Under proper hypotheses, a RFT model is solved by translating the parts affected by a repair into equivalent GSPN sub-models, in order to (efficiently) evaluate the steady-state probability of the related subsystems failures [19] in presence of repair policies. Then, those parts of the tree are collapsed into Basic Events whose occurrence probabilities are given by the solution of the GSPNs sub-nets. After that, the overall probability of the top event may be evaluated by means of the usual combinatorial techniques.

3.2 FT-to-GSPN Model Transformation

The *FT-to-GSPN* transformation consists of seven matched rules that map the ones described in [7] into the declarative language we used to implement the model transformation. In the following we refer to the ATLAS Transformation Language (ATL) [13] that provides the composition mechanisms we described in Section 2. ATL is one of the most used and supported transformation language by academic and industrial model driven community. Its success is due to its simplicity and also to its participation to the Eclipse Modeling Projects.

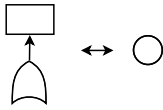


Fig. 3. Rule 1

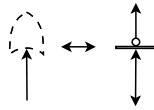


Fig. 4. Rule 2

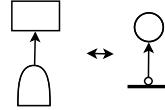


Fig. 5. Rule 3

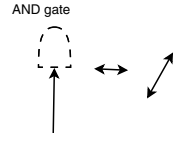


Fig. 6. Rule 4

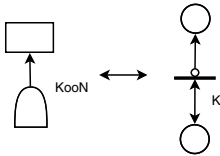


Fig. 7. Rule 5

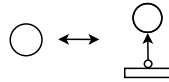


Fig. 8. Rule 6

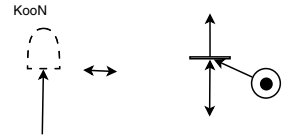


Fig. 9. Rule 7

The seven matched rules are illustrated in the figures from Fig. 3 to Fig. 9. A dotted line means that those elements just describe the context in which the rule operates, but they are not translated. *rule1* translates the output event of an OR gate (Fig. 3). *rule2* translates an input arc of an OR gate (Fig. 4). *rule3* translates the output event of an AND gate (Fig. 5). *rule4* translates an input arc of an AND gate (Fig. 6). *rule5* translates the output event of an K-out-of-N (KooN) gate (Fig. 7). *rule6* translates a basic event (Fig. 8) and finally *rule7* translates an input arc of an KooN gate (Fig. 9).

3.3 Inheriting RFT-to-GSPN

According to our model driven approach based on the formalisms inheritance, the first step requires that the formalisms meta-models involved as source and target languages of the transformations are considered. Fig. 11 shows the meta-model of the GSPN language which is the target language, and Fig. 10 shows the RFT meta-model and how RFT \ll inherits \gg from FT.

The set of elements of the RFT formalism consists of all the elements defined by the FT formalism plus the RB node that models a repair action. Hence this is the case 1 described in Section 2. When fault propagation causes an event which

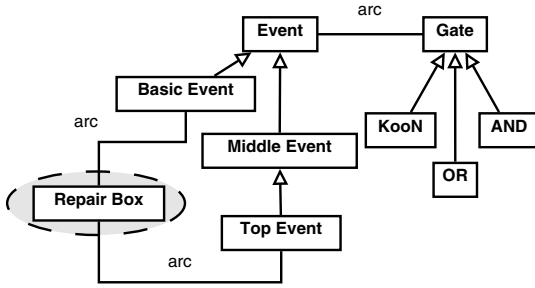


Fig. 10. Metamodel of Fault Tree language

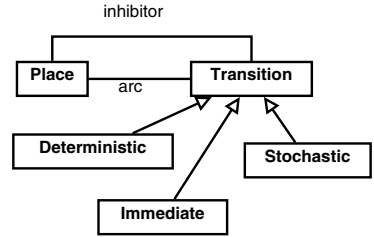


Fig. 11. Metamodel of GSPN language

is connected to a RB to happen (trigger event), the events that caused the fault get cleared (repaired) according to a defined policy. As a consequence, the FT arc element is inherited by RFT but it is used in two new contexts (Section 2 case 2.a):

- to connect a RB to a middle event (i.e. the trigger, representing a failure occurring at a sub-tree level and enabling the repair);
- to connect a RB to a basic event to be repaired.

This would mean that two rules must be defined that take into account the translation of triggering and repair arcs. The first one generates the target GSPN subnet depicted in Fig. 12 that is needed to model the copy of a token from a place to another; this token enables the activation of the GSPN subnet representing the repair action. The second rule has an intrinsic high complexity. It must generate the GSPN target subnets in charge of modeling the restoration of the initial marking after the repair has been completed. This aim is achieved by two kinds of subactions: one that clears a marked place (clearing pattern Fig. 14) and one that marks an empty place (filling pattern in Fig. 13). Some of the places of these GSPN patterns are tagged with two labels (*endLabel* and *triggerLabel*) in order to merge with the translated RFT with the GSPN model of the repair policy as described at the end of this Section.

The seven rules from *FT – to – GSPN* transformation are inherited and fully reused since they are related to the FT elements neither modified nor reinterpreted by the RFT formalism. In conclusions Fig. 15 summarizes the *RFT – to – GSPN* transformation rules. On the left part of the figure a generic

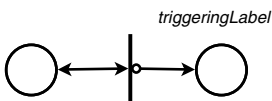


Fig. 12. Triggering pattern

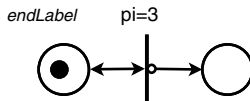


Fig. 13. Filling pattern

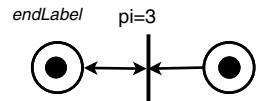


Fig. 14. Clearing pattern

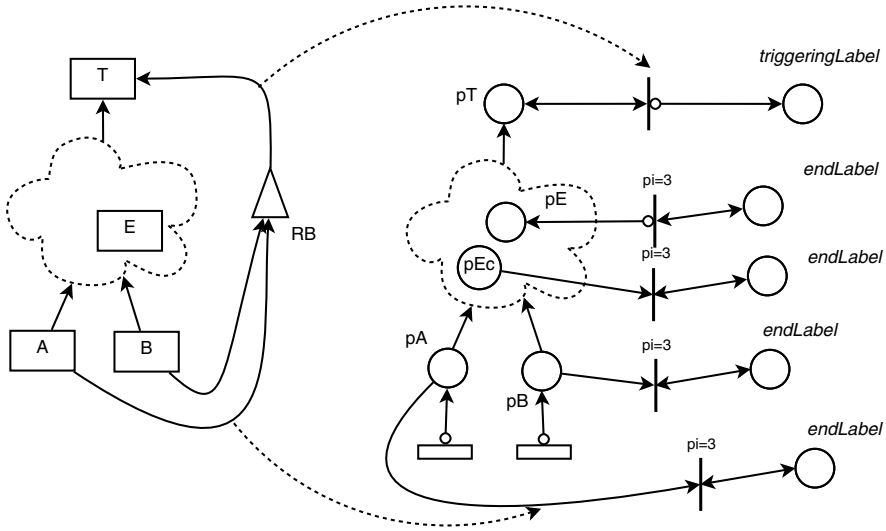


Fig. 15. RFT-to-GSPN

RFT model is depicted where a RB is connected by a trigger arc to the T event and by two repair arcs to the Basic Events A and B.

Let us suppose that the E event is in the path from A and B to T. According to the defined rules, a triggering pattern is generated from the trigger arc: one place of the triggering pattern is connected to the place pT representing the event T. The repair arcs connecting the Basic Events A and B are translated by the clearing patterns connected to the places pA and pB . For the intermediate events proper clearing and/or filling patterns are generated according to the type of the gate in input to the event. From the previous description it should be clear that the *RFT – to – GSPN* transformation is more complex than *FT – to – GSPN* due to the complexity of the single repair action and of the global repair strategy of a RFT. The rule that matches a trigger arc is quite simple, but the rule that matches the repair arc requires recursive calls in order to explore an entire subtree when the subsystem must be repaired. The RFT-to-GSPN transformation does not explicitly address the translation of the repair policies embedded into the RBs that are already expressed by means of a GSPN models. Hence, in our approach a GSPN model template is considered available for each repair policy. In order to have the entire solvable GSPN net, the GSPN resulting from the application of the RFT-to-GSPN transformation must be integrated with the GSPN model of the repair policy.

This integration is accomplished by superposing the places that has been tagged during the translation (*triggerLabel* and *endLabel*).

3.4 Model-to-Text Transformation

In order to allow the analysis of the generated GSPN models we must translate them into the format of a some existing solution tool. This is done by means

of a Model-to-Text (M2T) transformation. A full description of M2T techniques and the main supporting tools is out of the scope of this paper: for this purpose we use ATL whose queries can also be exploited in order to obtain a string as output. As solving tool we have chosen GreatSPN [8] due to its assessed architecture, great efficiency in GSPN analyses. The Algebra tool can be used in order to integrate the GSPN models. Notwithstanding, the use of Algebra it is not mandatory since it is possible to define a proper M2M transformation to merge two GSPN nets.

4 The Radio Block Centre Case Study

In [11] the RBC case-study was analyzed in order to investigate the impact of different repair policies on the system availability. To that aim, a rather complex formal model was built (by hand) using the RFT formalism. We will show how a GSPN model can be obtained from this RFT by applying the transformations defined in Section 3. RBC is an ERTMS/ETCS subsystem¹ and controls the movements of the trains traveling across supervised track area. The availability of a RBC is critical as there is no way for the signalling system to work without its contribution. In case of a RBC failure, all the trains under its supervision are compelled to brake and proceed in a degraded mode. This would lead to the *Immobilising Failure*, the most critical among the ERTMS/ETCS safe failures.

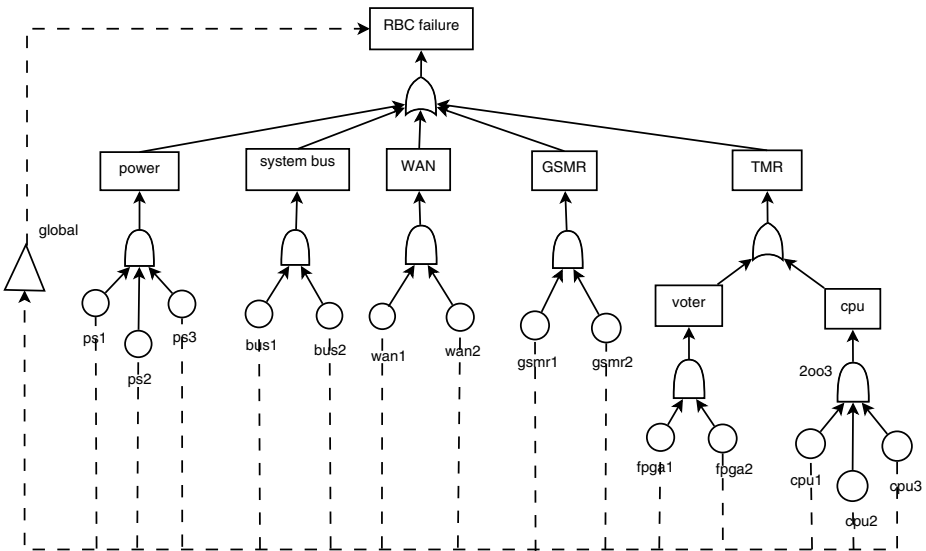
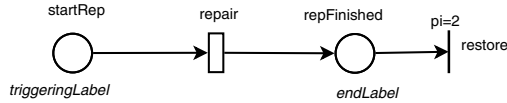


Fig. 16. The RBC model

¹ ERTMS/ETCS is the signalling system at the basis of several worldwide high speed rail projects.

Table 2. MTBFs of the RBC components

Unit	MTBF[h]
CPU board	$1.35 * 10^6$
Bus	$2.25 * 10^5$
FPGA	$3.33 * 10^8$
Power Supply	$5.5 * 10^4$
GSM-R card	$1.752 * 10^5$
Network card	$4 * 10^5$

**Fig. 17.** GSPN template

Due to RBC the criticality and its positioning in technical rooms, maintenance takes an important role for the non-functional requirements fulfillment: proper repairing policies have to be defined and evaluated by means of (formal) models.

Thus, a RFT model has been developed starting from the RBC reference architecture. The underlying FT was built according to the usual Fault Tree Analysis techniques. Then Repair Boxes can be applied to the RBC in order to evaluate the effect of maintenance policies on the overall system availability. In order to better explain the translation steps, let us consider just a single repair box.

According to Fig. 16, when a failure occurs the *global* RB is triggered and one or more Basic Events connected to global may be repaired. The reference values for the MTBFs (in hours) of system components have been chosen from the data-sheets of commercial devices (see 11) and are reported in Table 2.

Here we suppose that the Repair Box encapsulates the GSPN model described in Fig. 17. This GSPN net is a simple Global Repair Time (GRT) repair policy: only one parameter, representing the global time needed to repair the whole set of basic events connected with the RB, is given; when the repair time elapses, then all basic events connected with the RB are immediately repaired. We choose the MTTR of 0.5 hours that is, according to 11, the time for system repair that is the inverse of the rate of *repair* stochastic transition.

Fig. 18 describes the transformations from the RFT to GSPN model of the system. For sake of simplicity we focus our attention only on a part of the RFT (including the GSMR module). The application of the *RFT – to – GSPN* transformation translates the model according to superimposition mechanism: that is it generates the Petri Nets contained in the *ft structure* box using the set of seven rules inherited from *FT – to – GSPN*; then the parts of the GSPN described in the boxes *trigger arc* and *repair arc* are generated by translating the triggering and repair arcs according to its own rules. The whole GSPN model is obtained by merging the GRT policy model and the RBC model by place superposition over the matching labels (*superposition area* ovals).

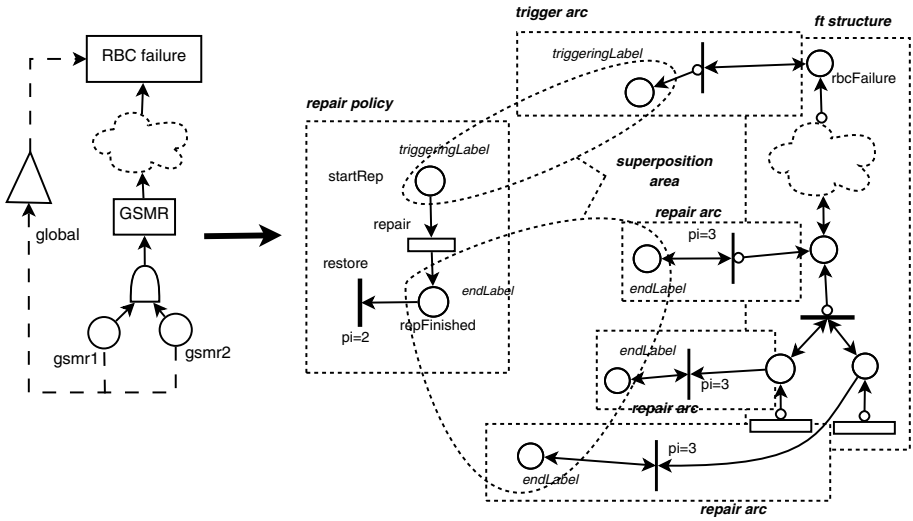


Fig. 18. GSPN model of RBC

The generated GSPN is now ready to be analyzed by means of specific tools. The probability of occurrence of immobilising failure (and so the RFT top event occurrence) can be computed by evaluating the mean number of tokens into the GSPN corresponding place (*rbcFailure*). According to data previously reported, the probability of unavailability is $1.63 \cdot 10^{-3}$; this value is too high due to the fact that only a single repair box has been considered for containing the description of example in a limited space.

5 Related Work

Language inheritance and formalism hierarchies are wide research fields. In [21] a very deep analysis of inheritance mechanisms has been done with particular reference to programming languages, while in [4] the focus of the analysis shifts towards models and metamodels. More systematic studies [17] confirm that compositionality and reuse are hot topics in this research field. First approaches in transformation composition are focused on reusing single rules [14] or patterns [3]. Another interesting approach is represented by VIATRA2 [2] that can be easily applied also in the context of embedded and critical systems. As it comprehends a graph pattern matching based language, reuse is almost applied at rule level: in order to make the transformational processes usable in real world applications, it is necessary to increase the level of reuse allowing entire transformations to be used as extension point for the others. A recent research proposes an alternative compositional mechanism that fulfills the previous requirement: in [13], the authors define the module superimposition as a simple mechanism for coarse-grain reuse. In [1] this mechanism has been applied to model composition in software product lines.

The integration of formal methods and techniques into MDE based development processes is still an open research issue. A recent research work [6] proposes an extension of MARTE for dependability and modeling (MARTE-DAM). In the past, research efforts have focused on the generation of formal models from UML [16], [5] or from AADL [20].

6 Conclusions

In this paper we have presented a study on deriving (and more broadly engineering) model-to-model transformations from existing ones, under the hypothesis of inheritance relationship between meta-models. Specifically, such mechanism has been applied in formalism integration within the context of the availability analysis of critical systems.

The implementation of a model transformation from FT to GSPN is provided and a RFT to GSPN transformation is inherited from it, where the RFTs language (Repairable Fault Trees) is an extension of FTs. As other factorization and modularization mechanisms described in the literature, the proposed approach allows to apply a divide-et-impera principle that is very important when coping with a transformation that presents several layers of complexity.

The primary concern of any future work is a deeper insight into the formal language hierarchies and the role that they play in model transformations. A first development in this line will result in the definition of integration mechanisms between xFT languages (Fault Trees and their extensions, Dynamic Fault Trees, Non Deterministic Repairable Fault Trees, Parametric Fault Trees, ...) and xPN languages (Petri Nets different classes, Timed Petri Nets, Stochastic Petri Nets, Stochastic Well Formed Petri Nets,...). Further efforts will be spent on the application of transformational and compositional approaches to other dependability aspects as safety and security.

References

1. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 4–19. Springer, Heidelberg (2009)
2. Balogh, A., Varró, D.: Advanced model transformation language constructs in the VIATRA2 framework. In: 21st Annual ACM Symposium on Applied Computing, pp. 1280–1287. ACM, New York (2006)
3. Balogh, A., Varró, D.: Pattern composition in graph transformation rules. In: European Workshop on Composition of Model Transformations (2006)
4. Barbero, M., Jouault, F., Gray, J., Bézivin, J.: A Practical Approach to Model Extension. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 32–42. Springer, Heidelberg (2007)
5. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable petri net models. In: Proceedings of the 3rd International Workshop on Software and Performance, pp. 35–45. ACM, New York (2002)
6. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Software and Systems Modeling (2009)

7. Bobbio, A., Franceschinis, G., Gaeta, R., Portinale, L.: Parametric Fault Tree for the Dependability Analysis of Redundant Systems and Its High-Level Petri Net Semantics. *IEEE Transaction on Software Engineering* 29(3), 270–287 (2009)
8. Chiola, G., Franceschinis, G., Gaeta, R., Ribaud, M.: GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation* 24(1), 47–68 (1995)
9. Codetta-Raiteri, D.: Extended Fault Trees Analysis supported by Stochastic Petri Nets. Ph.D. Thesis. Univ. di Torino (2005)
10. Di Lorenzo, G., Flammini, F., Iacono, M., Marrone, S., Moscato, F., Vittorini, V.: The software architecture of the OsMoSys Multisolution Framework. In: Proc. of 2nd International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS). ACM, New York (2007)
11. Flammini, F., Mazzocca, M., Iacono, M., Marrone, S.: Using Repairable Fault Trees for the Evaluation of Design Choices for Critical Repairable Systems. In: Proc. of High Assurance System Engineering, pp. 163–172. IEEE Computer Society, Washington (2005)
12. Franceschinis, G., Gribaudo, M., Iacono, M., Marrone, S., Moscato, F., Vittorini, V.: Interfaces and Binding in Component Based Development of Formal Models. In: Proc. of 4th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS). ACM, New York (2009)
13. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. *Software and Systems Modeling* (2009)
14. Kurtev, I., van den Berg, K., Jouault, F.: Rule-based modularization in model transformation languages illustrated with ATL. *Sci. Comput. Program.* 68(3), 111–127 (2007)
15. Lee, W.S., Grosh, D.L., Tillman, F.A., Lie, C.H.: Fault Tree Analysis, Methods and Applications-A Review. *IEEE Trans. Reliability* 34, 194–203 (1985)
16. Majzik, I., Pataricza, A., Bondavalli, A.: Stochastic dependability analysis of system architecture based on UML models. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) *Architecting Dependable Systems*. LNCS, vol. 2677, pp. 219–244. Springer, Heidelberg (2003)
17. Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformations. In: Proc. Dagstuhl Seminar on Language Engineering for Model-Driven Software Development (2005)
18. Murata, T.: Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE* 77(4), 541–580 (1989)
19. Raiteri, D.C., Franceschinis, G., Iacono, M., Vittorini, V.: Repairable fault tree for automatic evaluation of repair policies. In: Proc. of the Performance and Dependability Symposium. IEEE Computer Society, Washington (2004)
20. Rugina, A.E., Kanoun, K., Kaâniche, K.: A System Dependability Modeling Framework Using AADL and GSPNs. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) *Architecting Dependable Systems IV*. LNCS, vol. 4615, pp. 14–38. Springer, Heidelberg (2007)
21. Taivalsaari, A.: On the notion of inheritance. *ACM Computing Surveys* 28(3), 438–479 (1996)
22. UIC: ERTMS/ETCS class1 System Requirements Specification, SUBSET-026, issue 2.2.2 (2002)
23. Vittorini, V., Iacono, M., Mazzocca, N., Franceschinis, G.: The OsMoSys approach to multiformalism modeling of systems. *Journal of Software and Systems Modeling* 3(1), 68–81 (2004)

Translating Pi-Calculus into LOTOS NT

Radu Mateescu¹ and Gwen Salaün^{1,2}

¹ INRIA Grenoble – Rhône-Alpes / VASY project-team / LIG, Inovallée
655, av. de l'Europe, Montbonnot, F-38334 Saint Ismier, France

² Grenoble INP, 46, av. Félix Viallet, F-38031 Grenoble, France
{Radu.Mateescu,Gwen.Salaun}@inria.fr

Abstract. Process calculi supporting mobile communication, such as the π -calculus, are often seen as an evolution of classical value-passing calculi, in which communication between processes takes place along a fixed network of static channels. In this paper, we attempt to bring these calculi closer by proposing a translation from the finite control fragment of the π -calculus to LOTOS NT, a value-passing concurrent language with classical process algebra flavour. Our translation is succinct in the size of the π -calculus specification and preserves the semantics of the language by ensuring a one-to-one correspondence between the states and transitions of the labeled transition systems corresponding to the input π -calculus and the output LOTOS NT specifications. We automated this translation by means of the PIC2LNT tool, which makes it possible to analyze π -calculus specifications using all the state-of-the-art simulation and verification functionalities provided by the CADP toolbox.

1 Introduction

Process calculi (or algebras) are abstract specification languages used to model concurrent systems. These formalisms have been widely studied and used for the specification of real-world systems in many different application areas such as telecommunication protocols, hardware design, or embedded systems. One of the most famous calculi is the π -calculus [20] proposed by Milner, Parrow, and Walker about twenty years ago. The π -calculus is an extension of CCS [18] with mobile communication, and is equipped with an operational semantics defined in terms of labeled transition systems (LTSS). Although a lot of theoretical results have been achieved on this language (see [24] for a survey), only a few verification techniques and tools, such as the Mobility Workbench (MWB) [28] or JACK [8], are operational for analyzing π -calculus specifications automatically.

In this paper, we attempt to provide similar analysis features for π -calculus specifications by reusing the verification technology already available for classical (*i.e.*, without mobility) value-passing process algebras. Contrary to existing analysis tools for the π -calculus, which rely on specific algorithms and intermediate models, such as HD-automata [8], our approach is based on a novel translation from π -calculus to a classical process algebra. We focus here on the finite control fragment of the π -calculus and adopt as target language LOTOS NT [4], a recent enhancement of LOTOS [16]. In LOTOS NT, the abstract data type part was

abandoned in favor of constructive data type definitions and pattern-matching, and the behavior process algebraic part was replaced by an imperative-like language with a user-friendly syntax. To the best of our knowledge, this is the first π -calculus translation that uses a classical process algebra as target language.

Most of the π -calculus constructs can be translated quite straightforwardly into LOTOS NT thanks to its good level of expressiveness. Nevertheless, we faced some subtle difficulties in order to have a translation as succinct as possible and preserving the LTS semantics, *i.e.*, mapping each transition of a π -calculus agent to a transition of the resulting LOTOS NT term. Obviously, one of the main problems was to emulate mobile communication in a language that offers only communication on static channels; we overcame this issue by heavily exploiting the data types and synchronization features of LOTOS NT. Our translation is fully automated by the PIC2LNT tool we have implemented. Since LOTOS NT is one of the input languages of the CADP [12] verification toolbox, all the state-of-the-art verification features of CADP can be used on the LOTOS NT specifications generated from the π -calculus ones.

The outline of the paper is as follows. In Section 2, we introduce both specification languages, namely the π -calculus and LOTOS NT. Section 3 presents the translation rules and shows the semantics preservation. Section 4 describes the PIC2LNT translator and Section 5 illustrates the overall approach on the specification and verification of a simple Web services case study. Finally, Section 6 compares our proposal with related approaches, and Section 7 draws up some conclusions and lines for future work.

2 Pi-Calculus and LOTOS NT

We briefly present below the syntax and semantics of π -calculus and of the LOTOS NT fragment that serves as target language for the translation.

Pi-Calculus. We consider the original version of π -calculus [20] equipped with the early operational semantics defined in [21]. For simplicity of the presentation, we focus on the monadic π -calculus, although the translation to LOTOS NT given in Section 3 can be straightforwardly extended to handle the polyadic version of the calculus [19]. The syntax and semantics of the π -calculus are shown in Figure 1. Channel names (denoted by a, \dots, z) belong to an infinite countable set of names \mathcal{N} . Agents (denoted by P) are built from inaction (0), action prefix (\cdot), parallel composition (\parallel), choice ($+$), channel creation (ν), guard ($[\]$), and instantiation ($A(\dots)$). The occurrences of y in $x(y).P$ and $(\nu y)P$ are bound and the other occurrences of channel names are free. The set of free (resp. bound) names of an agent P is denoted by $fn(P)$ (resp. $bn(P)$), and the set of names of P is defined as $n(P) = fn(P) \cup bn(P)$. Each agent identifier A has an arity $r(A) \geq 0$ and must be defined by an equation $A(x_1, \dots, x_{r(A)}) \stackrel{\text{def}}{=} P$. The parameter names $x_1, \dots, x_{r(A)}$ must be pairwise distinct and $fn(P) \subseteq \{x_1, \dots, x_{r(A)}\}$.

The actions (denoted by α) that an agent can perform are of four kinds: internal action (τ), free output ($\bar{x}y$), bound output ($\bar{x}(z)$), and free input (xy). The

$P ::= 0 \mid \tau.P \mid \overline{xy}.P \mid x(y).P \mid P_1 \mid P_2 \mid P_1 + P_2$ $\mid (\nu x)P \mid [x = y]P \mid [x \neq y]P \mid A(x_1, \dots, x_{r(A)})$		$\alpha ::= \tau \mid \overline{xy} \mid \overline{x}(z) \mid xy$			
TAU	$\tau.P \xrightarrow{\tau} P$	OUT	$\overline{xy}.P \xrightarrow{\overline{xy}} P$	IN	$x(y).P \xrightarrow{xz} P\{z/y\}$
SUM	$\frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 + P_2 \xrightarrow{\alpha} P'_1}$	PAR	$\frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 \mid P_2 \xrightarrow{\alpha} P'_1 \mid P_2}$ if $bn(\alpha) \cap fn(P_2) = \emptyset$		
COM	$\frac{P_1 \xrightarrow{\overline{xy}} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1 \mid P_2 \xrightarrow{\tau} P'_1 \mid P'_2}$	CLOSE	$\frac{P_1 \xrightarrow{\overline{xy}} P'_1 \quad P_2 \xrightarrow{xy} P'_2}{P_1 \mid P_2 \xrightarrow{\tau} (\nu y)(P'_1 \mid P'_2)}$ if $y \notin fn(P_2)$		
RES	$\frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}$ if $x \notin n(\alpha)$	OPEN	$\frac{P \xrightarrow{\overline{xy}} P'}{(\nu y)P \xrightarrow{\overline{x}(z)} P'\{z/y\}}$ if $x \neq y, z \notin fn((\nu y)P')$		
MATCH	$\frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$	MISMATCH	$\frac{P \xrightarrow{\alpha} P'}{[x \neq y]P \xrightarrow{\alpha} P'}$ if $x \neq y$		
IDE $\frac{P\{y_1/x_1, \dots, y_{r(A)}/x_{r(A)}\} \xrightarrow{\alpha} P'}{A(y_1, \dots, y_{r(A)}) \xrightarrow{\alpha} P'}$ if $A(x_1, \dots, x_{r(A)}) \stackrel{\text{def}}{=} P$					

Fig. 1. Syntax and early operational semantics of π -calculus

same notations $fn(\alpha)$, $bn(\alpha)$, and $n(\alpha)$ are used for actions, the only bound occurrence being z in the bound output $\overline{x}(z)$. Intuitively, bound names in actions correspond to references of places in the executing agent where substitutions must be performed; bound outputs are used to represent scope extrusions (rules OPEN and CLOSE). The early operational semantics is given in terms of rules enabling to infer the transitions, labeled by actions, that an agent can perform. The rules associated to binary operators (\mid and $+$) have also symmetric forms, omitted here for conciseness. Given a substitution $\sigma : \mathcal{N} \rightarrow \mathcal{N}'$, we denote by $P\sigma$ the agent P in which each free name x has been replaced by $\sigma(x)$, possibly with changes of the bound variables to avoid captures. The substitution $\{y_1/x_1, \dots, y_n/x_n\}$ maps each x_i into y_i for $i \in [1, n]$ and keeps all the other names unchanged. In the sequel, we will consider only π -calculus agents that satisfy the *finite control* property [6], *i.e.*, do not contain recursive calls of agent identifiers through the parallel composition operator.

LOTOS NT fragment. LOTOS NT [4] is a simplified variant of the E-LOTOS standard [15] that attempts to combine the best features of imperative programming languages and value-passing process algebras. LOTOS NT has a user-friendly syntax and a formal operational semantics defined in terms of labeled transition systems (LTSS). LOTOS NT is supported by the LNT.OPEN tool of CADP [12], which allows the on-the-fly exploration of the LTSS corresponding to LOTOS NT specifications. As target of our translation, we use only a small fragment of LOTOS NT, whose syntax and semantics are given in Figure 2.

LOTOS NT terms (denoted by B) are built from actions, choice (“**select**”), conditional (“**if**”), sequential composition (“;”), hiding (“**hide**”), and parallel composition (“**par**”). Communication is carried out by rendezvous on gates G with bidirectional transmission of multiple values (for simplicity, in Fig. 2 we considered actions with only two values being sent in both directions).

$B ::= \mathbf{stop} \mid \mathbf{null} \mid G(!E, ?x) \mathbf{where} E' \mid B_1; B_2 \mid \mathbf{if} E \mathbf{then} B \mathbf{end} \mathbf{if}$ $\mid \mathbf{var} x:T \mathbf{in} x := E; B \mathbf{end} \mathbf{var} \mid \mathbf{hide} G \mathbf{in} B \mathbf{end} \mathbf{hide}$ $\mid \mathbf{select} [\mathbf{var} x_1:T_1, \dots, x_n:T_n \mathbf{in}] B_1 [] \dots [] B_n \mathbf{end} \mathbf{select}$ $\mid \mathbf{par} G \mathbf{in} B_1 \parallel \dots \parallel B_n \mathbf{end} \mathbf{par} \mid P[g_1, \dots, g_m](E_1, \dots, E_n)$	
LNT-NULL $\mathbf{null} \xrightarrow{\delta} \mathbf{stop}$	LNT-ACT $\frac{v' \in \mathit{type}(x) \wedge \llbracket E' \{v'/x\} \rrbracket = \mathbf{true}}{G(!E, ?x) \mathbf{where} E'; B \xrightarrow{G \{!E\} !v'} B \{v'/x\}}$
LNT-SEQ-1 $\frac{B_1 \xrightarrow{\beta} B'_1}{B_1; B_2 \xrightarrow{\beta} B'_1; B_2}$	LNT-SEQ-2 $\frac{B_1 \xrightarrow{\delta} B'_1 \quad B_2 \xrightarrow{\beta} B'_2}{B_1; B_2 \xrightarrow{\beta} B'_2}$
LNT-IF $\frac{\llbracket E \rrbracket = \mathbf{true} \quad B \xrightarrow{\beta} B'}{\mathbf{if} E \mathbf{then} B \mathbf{end} \mathbf{if} \xrightarrow{\beta} B'}$	LNT-VAR $\frac{B \{ \llbracket E \rrbracket / x \} \xrightarrow{\beta} B'}{\mathbf{var} x:T \mathbf{in} x := E; B \mathbf{end} \mathbf{var} \xrightarrow{\beta} B'}$
LNT-HID-1 $\frac{B \xrightarrow{\beta} B' \quad \mathit{gate}(\beta) \neq G}{\mathbf{hide} G \mathbf{in} B \mathbf{end} \mathbf{hide} \xrightarrow{\beta} \mathbf{hide} G \mathbf{in} B' \mathbf{end} \mathbf{hide}}$	LNT-HID-2 $\frac{B \xrightarrow{\beta} B' \quad \mathit{gate}(\beta) = G}{\mathbf{hide} G \mathbf{in} B \mathbf{end} \mathbf{hide} \xrightarrow{i} \mathbf{hide} G \mathbf{in} B' \mathbf{end} \mathbf{hide}}$
LNT-SEL $\frac{i \in [1, n] \quad B_i \xrightarrow{\beta} B'_i}{\mathbf{select} [\mathbf{var} x_1:T_1, \dots, x_n:T_n \mathbf{in}] B_1 [] \dots [] B_n \mathbf{end} \mathbf{select} \xrightarrow{\beta} B'_i}$	LNT-PAR $\frac{i \in [1, n] \quad B_i \xrightarrow{\beta} B'_i \quad \mathit{gate}(\beta) \neq G}{\mathbf{par} G \mathbf{in} B_1 \parallel \dots \parallel B_n \mathbf{end} \mathbf{par} \xrightarrow{\beta} \mathbf{par} G \mathbf{in} B_1 \parallel \dots \parallel B'_i \parallel \dots \parallel B_n \mathbf{end} \mathbf{par}}$
LNT-COM $\frac{I \subseteq [1, n] \quad \forall i \in I. B_i \xrightarrow{\beta} B'_i \quad \mathit{gate}(\beta) = G \quad j \in I}{\mathbf{par} G \mathbf{in} B_1 \parallel \dots \parallel B_n \mathbf{end} \mathbf{par} \xrightarrow{\beta} \mathbf{par} G \mathbf{in} B_1 \parallel \dots \parallel B'_j \parallel \dots \parallel B_n \mathbf{end} \mathbf{par}}$	LNT-IDE $\frac{B \{g_1/G_1, \dots, g_m/G_m\} \{ \llbracket E_1 \rrbracket / x_1, \dots, \llbracket E_n \rrbracket / x_n \} \xrightarrow{\beta} B'}{P[g_1, \dots, g_m](E_1, \dots, E_n) \xrightarrow{\beta} B'}$
where process $P[G_1, \dots, G_m](x_1:T_1, \dots, x_n:T_n)$ is B end process	

Fig. 2. Syntax and early operational semantics of the LOTOS NT fragment

Synchronizations may also contain optional guards (“**where**”) expressing boolean conditions on received values. The gate on which an action β takes place is denoted by $\mathit{gate}(\beta)$. The special action δ is used for defining the semantics of sequential composition. An action $G(\dots)$ can occur in isolation, in which case it is considered to be equivalent to $G(\dots); \mathbf{null}$. The internal action is denoted by the special gate i , which cannot be used for synchronization. The parallel composition operator allows multiway rendezvous on the same gate. As in LOTOS [16], processes are parameterized by gates and data variables.

The reader familiar with LOTOS may notice that the LOTOS NT fragment considered is not far from LOTOS itself, which could also serve as the target language for the translation. However, as it will become clear in Section 3, LOTOS NT presents at least two advantages *w.r.t.* LOTOS for translating π -calculus agents: (a) the symmetric sequential composition operator “;” of LOTOS NT makes it possible to group together the behaviour following the branches of a “**select**” statement, thus enabling a succinct translation of nested action prefixes occurring in π -calculus agents, and (b) as opposed to the sequential composition operator “ \gg ”

of LOTOS, the semantics of the “;” operator of LOTOS NT does not create spurious internal actions in the LTS, making possible to achieve a one-to-one correspondence between the transitions of a π -calculus agent and those of the LOTOS NT term resulting after translation.

3 Translation from Pi-Calculus to LOTOS NT

The translation presented below maps each π -calculus agent P to a LOTOS NT behaviour term $t(P, \overline{G}, k)$, where \overline{G} is the set of LOTOS NT gates on which the term communicates with its environment and $k \geq 1$ is a natural number identifying the corresponding concurrent activity (*i.e.*, the operand of the immediately enclosing parallel composition operator, if any, which contains the term). Two classes of channels are distinguished: *public* channels correspond to the free channel names occurring in P , whereas *private* channels correspond to channel names bound by ν operators occurring in P . The set \overline{G} includes two predefined gates G_{pub} and G_{priv} , which serve to model the non-synchronized communications on public and private channels, respectively.

Channel names. Since the communication in LOTOS NT takes place along static gates, we cannot use directly these gates to represent mobile communication. Instead, we represent π -calculus channel names as values of a LOTOS NT data type `Chan`, and we model channel mobility between π -calculus agents by communicating values of this type along gates between the corresponding LOTOS NT processes. The example below shows the definition of the LOTOS NT type `Chan` for the π -calculus agent $(\nu x)(\overline{a}b.\overline{c}x.0)$.

<pre> type Chan is a, b, c, x (id:Nat) with "==" , "!=" end type function new_id () : Nat is !external null end function </pre>	<pre> function is_public (ch:Chan) : Bool is case ch in a b c -> return true any -> return false end case end function </pre>
---	--

The `Chan` type is equipped with the comparison operators “==” and “!=”. It provides a constant constructor for each public channel and a constructor parameterized by a natural number `id` for each private channel. The predicate `is_public` characterizes public channels. To create new `Chan` values when a ν operator is evaluated, we use a function `new_id` defined externally in \mathbb{C} , which returns a new natural number at each invocation.

Inaction and action prefix. The null π -calculus agent 0 is naturally translated into the **stop** LOTOS NT operator, which does not perform any action. The prefix operator is translated using the choice operator “**select**” and the sequential composition operator “;” as shown below. In order to capture all potential interactions that may become possible during execution due to mobility of channels, the communication on a channel x is modeled by a nondeterministic choice on all the gates connecting the current LOTOS NT term to its environment.

Binary synchronizations between the current term and its environment are enforced by emitting the value x of type Chan corresponding to x and the identifier k of the current term, which acts as sender (resp. receiver) for output (resp. input) actions. The semantics of LOTOS NT parallel composition (used to translate the $|$ operator, see below) ensures that only the terms corresponding to different concurrent activities (having different identifiers k) and sharing the same value x can communicate in an unidirectional manner by transmitting a value y of type Chan on some gate. Variables s and r act as placeholders for the identifiers of the sender and receiver terms, respectively.

$t(\bar{x}y.P, \{G_1, \dots, G_n, G_{pub}, G_{priv}\}, k) =$ select var r:Nat in $G_1 (!x, !y, !k, ?r) \square \dots$ $G_n (!x, !y, !k, ?r) \square$ $G_{pub} (!x, !y, !true)$ where $\text{is_public}(x) \square$ $G_{priv} (!x, !y, !true)$ where $\text{not}(\text{is_public}(x))$ end select ; $t(P, \{G_1, \dots, G_n, G_{pub}, G_{priv}\}, k)$	$t(x(y).P, \{G_1, \dots, G_n, G_{pub}, G_{priv}\}, k) =$ select var s:Nat, y:Chan in $G_1 (!x, ?y, ?s, !k) \square \dots$ $G_n (!x, ?y, ?s, !k) \square$ $G_{pub} (!x, ?y, !false)$ where $\text{is_public}(x) \square$ $G_{priv} (!x, ?y, !false)$ where $\text{not}(\text{is_public}(x))$ end select ; $t(P, \{G_1, \dots, G_n, G_{pub}, G_{priv}\}, k)$
---	---

When x denotes a public (resp. private) channel, an action on gate G_{pub} (resp. G_{priv}) is added in order to model the possibly non-synchronized execution of send/receive actions, which comprise the emission of a true/false boolean value in order to differentiate them in the LTS of the resulting LOTOS NT term. Note that the translation makes no difference between free and bound output, these actions being distinguished by the value y of type Chan being sent, which can be either public or private.

Sum, match, and mismatch. The sum operator is naturally translated by using the choice operator of LOTOS NT. The match and mismatch operators are translated in terms of the conditional operator.

$t(P_1 + P_2, \bar{G}, k) = \text{select } t(P_1, \bar{G}, k) \square t(P_2, \bar{G}, k) \text{ end select}$ $t([x = y]P, \bar{G}, k) = \text{if } x == y \text{ then } t(P, \bar{G}, k) \text{ end if}$ $t([x \neq y]P, \bar{G}, k) = \text{if } x != y \text{ then } t(P, \bar{G}, k) \text{ end if}$

Note that in the translation of the operands P_1 , P_2 , and P the set of gates \bar{G} and the identifier k of the current term do not change, since the sum operator is sequential (*i.e.*, it does not create new concurrent activities).

Parallel composition. The parallel composition operator is translated using the “**par**” operator of LOTOS NT. A fresh gate G_{new} is introduced to model the communication between the two LOTOS NT terms resulting from the translation of the operands P_1 and P_2 . Since the parallel operator creates two concurrent activities, two new distinct identifiers $2k$ and $2k + 1$ are assigned to the corresponding LOTOS NT terms. Given that G_{new} is added to the sets of gates connecting the two terms to their environment, every send/receive communication carried out by $P_1|P_2$ will also be executed by the whole LOTOS NT term. Indeed,

according to the translation of the action prefix (see above), all input/output operations of P_1 and P_2 will also occur in the two LOTOS NT terms as actions along G_{new} , and the “**par**” operator will enforce their proper synchronization¹. All synchronizations on G_{new} are renamed into the internal action i using the “**hide**” operator to reflect the semantics of the π -calculus communication.

$$\begin{aligned}
 t(P_1|P_2, \overline{G}, k) = & \mathbf{hide} \ G_{new} \ \mathbf{in} \ \mathbf{par} \ G_{new} \ \mathbf{in} \\
 & t(P_1, \overline{G} \cup \{G_{new}\}, 2k) \ || \ t(P_2, \overline{G} \cup \{G_{new}\}, 2k + 1) \\
 & \mathbf{end} \ \mathbf{par} \ \mathbf{end} \ \mathbf{hide}
 \end{aligned}$$

The scheme for assigning concurrent activity identifiers yields a contiguous numbering if the direct nestings of parallel operators in the π -calculus agents are arranged to form balanced binary trees. Given that the parallel operator is associative, this can be easily obtained by an adequate insertion of parentheses, *e.g.*, $((P_1|P_2)|(P_3|P_4))$ instead of $((P_1|P_2)|P_3)|P_4$, which would be the default parsing of the agent in absence of parentheses.

Channel creation. The channel creation operator (νx) is translated by creating a new private value of type **Chan** and storing it in a variable x , which can be subsequently used by the LOTOS NT term.

$$t((\nu x)P, \overline{G}, k) = \mathbf{var} \ x:\mathbf{Chan} \ \mathbf{in} \ x := x(\mathbf{new_id}()); t(P, \overline{G}, k) \ \mathbf{end} \ \mathbf{var}$$

This translation rule does not directly forbid the LOTOS NT term to perform an emission along the channel x , in the sense that some action $\overline{x}a$ present in P (whose execution is forbidden by the rule **RES** in Fig. 11) will be translated as an action “ $G(!x, !a, !k, ?r)$ ” on some gate $G \in \overline{G}$. Such emissions are forbidden indirectly by the way in which action prefix and parallel composition are translated (see above). Indeed, for the synchronization on G to take place, the environment must propose on G an action containing the same fresh value x . This is impossible unless x has been previously sent by the current LOTOS NT term to the environment, by an emission corresponding to a bound output previously executed by the agent. Thus, scope extrusions are modeled by the communication of fresh values of type **Chan**, which can be subsequently used by different LOTOS NT terms for communication.

Agent definition and instantiation. Agent definitions A are mapped to LOTOS NT process definitions as shown below. In addition to the channel names $x_1, \dots, x_{r(A)}$ (represented as values of type **Chan**), the process is parameterized by the gate set \overline{G} and the identifier k , which capture the context of the call.

$$\begin{aligned}
 t(A(x_1, \dots, x_{r(A)})) & \stackrel{\text{def}}{=} P, \overline{G}, k = \mathbf{process} \ A_d[\overline{G}](x_1, \dots, x_{r(A)}:\mathbf{Chan}, k:\mathbf{Nat}) \ \mathbf{is} \\
 & t(P, \overline{G}, k) \\
 & \mathbf{end} \ \mathbf{process} \\
 t(A(y_1, \dots, y_{r(A)}), \overline{G}, k) & = A_d[\overline{G}](y_1, \dots, y_{r(A)}, k)
 \end{aligned}$$

¹ The translation of an agent $P_1|...|P_n$ requires $n-1$ gates, one for each $|$ operator. The number of gates could be reduced to 1 by using the generalized parallel composition operator (not yet fully implemented in LOTOS NT) proposed in [13], which can model binary synchronization between n processes.

Since an agent identifier may be invoked at several places in the π -calculus agent under translation, and LOTOS NT processes have a fixed number of gate parameters, we chose to produce one LOTOS NT process definition A_d for each occurrence of the agent identifier A in a context where $|\overline{G}| = d$. This increases the size of the LOTOS NT specification by only a logarithmic factor *w.r.t.* the size of the input π -calculus specification (see the discussion on complexity below). Finally, the restriction to finite control agents ensures that all (direct or transitive) recursive invocations of the agent identifier A inside its body P will occur inside the same concurrent activity, and therefore all the corresponding calls to process A_d will have the same context \overline{G} and k . This enables to translate each π -calculus agent definition into a finite number of LOTOS NT processes.

Pi-calculus specification. The π -calculus agent occurring at the top-level of a specification is translated in a context consisting of the gates G_{pub} and G_{priv} (which model the communications on public and private channels, respectively) and a concurrent activity with identifier 1.

$$\pi 2nt(P) = \mathbf{par} \ G_{priv} \ \mathbf{in} \ t(P, \{G_{pub}, G_{priv}\}, 1) \ || \ \mathbf{stop} \ \mathbf{end} \ \mathbf{par}$$

The translation of action prefix and channel creation operator (see above) produces extra synchronizations on gate G_{priv} , which must be forbidden in order to reflect that the environment of the π -calculus agent is not aware of the private channels of P . This is done by a synchronization on G_{priv} with “**stop**”, the simplest environment not aware of private channels, added at the top-level of the resulting LOTOS NT term.

Correctness and complexity of the translation. While devising the translation, we sought to preserve the behaviour by ensuring a one-to-one correspondence between the transitions performed by the π -calculus agent and those performed by the LOTOS NT term. The actions α of the agent are related to LOTOS NT actions by the function $h(\alpha, G, k)$, where G is a gate name and $k \geq 1$:

α	$h(\alpha, G, k)$
τ	i
$\overline{x}y$	$G !x !y !k ?r:\text{Nat}$
$\overline{x}(z)$	$G !x !z !k ?r:\text{Nat} \ \text{where} \ \neg\text{is_public}(z)$
xy	$G !x ?y:\text{Chan} ?s:\text{Nat} !k$

We also define the set $C_k \stackrel{\text{def}}{=} \{2^l \cdot k + r \mid l \geq 0 \wedge r < 2^l\}$, which represents the set of concurrent activity identifiers generated as children of activity k . The following proposition states the correctness of the translation.

Proposition 1 (Behaviour preservation). *Let P be a π -calculus agent, \overline{G} a set of LOTOS NT gates, and $k \geq 1$. Then, for every action α and agent P' :*

$$P \xrightarrow{\alpha} P' \quad \text{iff} \quad \exists k' \in C_k . \forall G \in \overline{G} . t(P, \overline{G}, k) \xrightarrow{h(\alpha, G, k')} t(P', \overline{G}, k).$$

The proof of this proposition (omitted here due to space limitations) is by induction on the depth of the derivation leading to the transition $P \xrightarrow{\alpha} P'$. When

translating a top-level π -calculus agent P in the context given by $\{G_{pub}, G_{priv}\}$ and activity identifier 1 (see the rule for π -calculus specification), Proposition 1 and the semantics of the “**par**” operator ensure that each internal transition of P is mapped to one internal transition of the LOTOS NT term and each communication on a public channel corresponds to an action on gate G_{pub} . On the other hand, since the synchronizations on G_{priv} are blocked, there are no communications between P and its environment along the private channels of P . Thus, every action performed by a top-level π -calculus agent is mapped into a single action (either **i**, or an action on G_{pub}) of the resulting LOTOS NT term.

In order to estimate the complexity of the translation, we calculate the size of the output LOTOS NT term *w.r.t.* the size $|P|$ of the input π -calculus agent P . The size is defined as the number of operators contained in the LOTOS NT term and in the π -calculus agent, respectively. The definition of type **Chan** has a size linear *w.r.t.* $|P|$ and each translation rule given above invokes the translation t only once for each operand of P . The only sources of increase in size are the translation rules for action prefixes and for agent definitions. In the worst case, the former rule expands each action of P into $|\overline{G}|_{max}$ actions, and the latter duplicates the definition of an agent as many times as $|\overline{G}|_{max}$, the maximum size of the set \overline{G} . Assuming that all nestings of parallel operators inside P are arranged to form balanced binary trees, $|\overline{G}|_{max}$ is bounded by $O(\log |P|)$, which makes the size of the whole LOTOS NT term proportional to $O(|P| \cdot \log |P|)$.

4 Tool Support: Pic2Lnt

We developed an automatic translator tool from π -calculus to LOTOS NT, named PIC2LNT, implemented using the SYNTAX+TRAIAN compiler construction technology [11]. It consists of about 900 lines of SYNTAX code, 2,300 lines of LOTOS NT code, and 500 lines of C code. Although the π -calculus version used in Sections 2 and 3 to illustrate the translation is monadic, the PIC2LNT translator implements a polyadic version of the π -calculus, by exploiting the fact that LOTOS NT allows the communication of multiple values on the same gate. The concrete syntax of polyadic π -calculus accepted by PIC2LNT subsumes the syntax implemented in MWB, with the restriction to finite control agents. Figure 3 gives an overview of the tool chain that makes possible the verification of π -calculus specifications using the CADP toolbox [12].

We applied PIC2LNT on a benchmark of π -calculus specifications, which includes most of the examples provided with MWB (except those with self-recursion along the parallel operator), as well as unitary tests that we wrote ourselves. Our benchmark currently contains 160 files, which consist of about 2,000 lines of π -calculus and were translated in about 23,000 lines of LOTOS NT. This expansion in size is caused partly by the complexity of the translation (estimated at the end of Section 3) and partly by the fact that LOTOS NT is more verbose than the π -calculus (*e.g.*, LOTOS NT requires more keywords, gates have to be declared explicitly and passed as parameters to each process call, variables must be declared before usage, etc.).

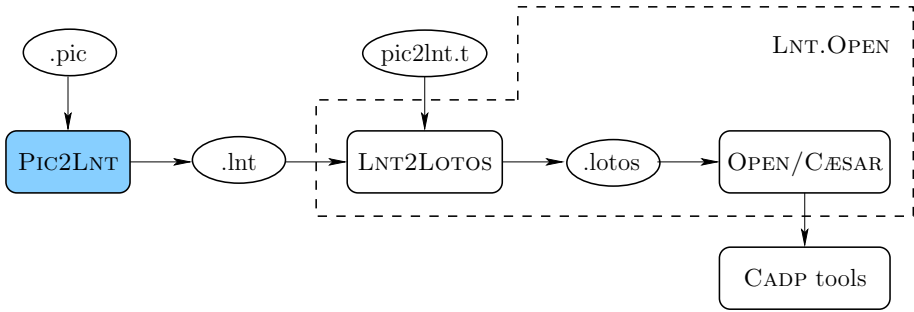


Fig. 3. Overview of the translation and verification process

Once a π -calculus specification is translated into LOTOS NT, the LNT.OPEN tool connects, by means of an intermediate translation into LOTOS (the `pic2lnt.t` file contains the external C definition of the function `new_id()`), the resulting specification to the OPEN/CÆSAR environment [10], which gives access to all the state-of-the-art on-the-fly verification tools of CADP. In particular, one can use the EVALUATOR 4.0 model checker [17] to verify temporal properties specified in MCL (an extension of alternation-free μ -calculus with regular expressions, data-based constructs, and fairness operators) involving channel names and/or data values present on transition labels. The counterexamples provided by the model checker are translated back into the π -calculus format by using the label renaming features provided by CADP.

5 Case Study: A Dispatcher Web Service

With the recent advent of Web services, the π -calculus has found a new application area. Many works have focused on the application of the π -calculus for modelling Web services and their composition, see *e.g.*, [7,14,25]. As far as implementation languages are concerned, BPEL is an XML-based executable language for implementing Web services orchestrations, and its specification [5] includes some dynamic communication primitives, namely endpoint references. As written in the BPEL specification: “An endpoint reference makes it possible in BPEL to dynamically select a provider for a particular type of service and to invoke their operations”.

In this section, we present an example of Web service (a dispatcher) involving dynamicity in the system architecture. This service receives requests from some clients, and depending on the product searched by the client, forwards this request to the adequate server. The server receives this request from the dispatcher as well as a private channel (x), and uses this new channel to interact directly with the client. First, it sends to him/her some information about the product (*e.g.*, price, availability, etc.) and next it receives the client’s final decision (*purchase* or *refuse*). The client stops as soon as (s)he accepts the purchase. We show

below the π -calculus specification of a system composed of one client and three servers selling different products (identified by a, b, c). In the specification, we use four private channels ($req, a, b,$ and c) and three public channels ($request, purchase,$ and $refuse$). Polyadic emissions are enclosed between angle brackets.

$$\begin{aligned}
 Main &= (\nu req, a, b, c)(Client(req, a, b, c) \mid Dispatcher(req) \mid \\
 &\quad Server(a) \mid Server(b) \mid Server(c)) \\
 Client(req, a, b, c) &= (\nu x)(\overline{request\ a}.\overline{req}(a, x).ClientAux(req, a, a, b, c, x)) + \\
 &\quad (\nu x)(\overline{request\ b}.\overline{req}(b, x).ClientAux(req, b, a, b, c, x)) + \\
 &\quad (\nu x)(\overline{request\ c}.\overline{req}(c, x).ClientAux(req, c, a, b, c, x)) \\
 ClientAux(req, k, a, b, c, x) &= x(\overline{info}).(\overline{x\ purchase}.\overline{purchase\ k}.0 + \\
 &\quad \overline{x\ refuse}.\overline{refuse\ k}.Client(req, a, b, c)) \\
 Dispatcher(req) &= req(k, x).\overline{k\ x}.Dispatcher(req) \\
 Server(k) &= k(x).\overline{x\ info}.x(decision).Server(k)
 \end{aligned}$$

Figure 4(a) shows the LTS generated using the LNT.OPEN tool from the LOTOS NT specification (an excerpt of which can be found in Appendix A) produced by the PIC2LNT translator. The transition labels have been renamed using CADP to keep only the relevant information about channels. The system starts emitting a request for one of the three possible products (state 0). Then, the dispatcher interacts with the concerned server, and this server with the client. This corresponds to sequences of τ transitions in the LTS because private interactions result in hidden transitions according to the π -calculus semantics. At that point of the execution, if the client decides to purchase the product (states 13, 15, 17), the system terminates (state 19). If the client decides to refuse the purchase (states 14, 16, 18), state 20 is reached where the client can submit another request.

Next, we illustrate how the EVALUATOR 4.0 model checker [17] of CADP can be used for analyzing this simple example. This tool accepts as input MCL formulas

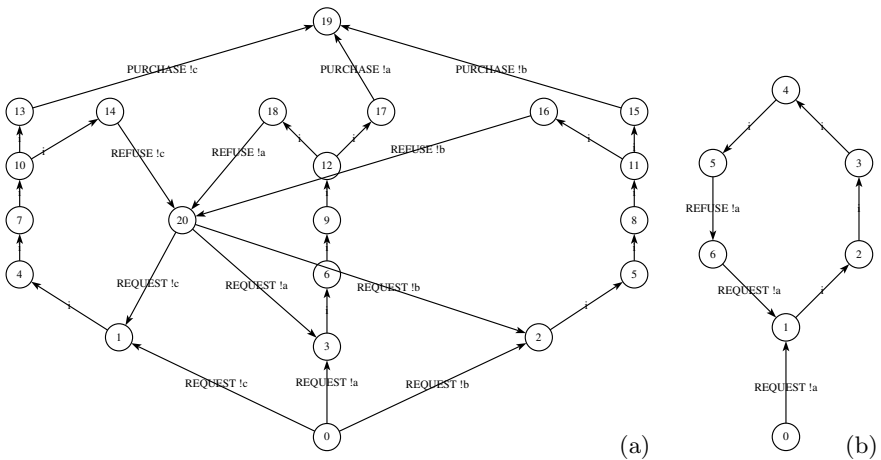


Fig. 4. (a) LTS of the dispatcher specification. (b) Lasso-shaped counterexample.

expressing properties on actions but also on data parameters. In particular, for LTS models generated from π -calculus specifications, MCL makes it possible to specify properties about channels appearing either as subject or object of a communication. For example, the MCL formula below expresses that each request submitted by the client is eventually answered positively:

$$[true^*.\{request ?x:String\}] \mu X.((true)true \wedge [\neg\{purchase !x\}]X)$$

The first box modality matches all transition sequences starting at the initial state of the LTS and ending with a *request* action. The channel value (encoded as a character string) communicated via *request* is captured in the x variable, which is reused later on in the minimal fixed point formula expressing the inevitable reachability of a *purchase* action involving x . This formula fails on the LTS in Figure 4(a) because a client can indefinitely refuse a product, as illustrated in Figure 4(b) by the lasso-shaped counterexample exhibited by EVALUATOR 4.0.

The other MCL formula below states that no positive or negative answer can be delivered for product a without a corresponding request being issued:

$$[(\neg\{request !"a"\})^*.\{purchase !"a"\} \vee \{refuse !"a"\}]false$$

This box modality forbids the existence of a sequence consisting of (0 or more) actions different from a *request* for a , followed by a *purchase* or a *refuse* action concerning a . Using EVALUATOR 4.0, we checked that this property holds on the LTS shown in Figure 4(a).

6 Related Work

During the past two decades, various approaches were followed for analyzing π -calculus specifications automatically. One of the first analysis tools specifically dedicated to the π -calculus was the Mobility Workbench (MWB) [28], developed in the 90s for manipulating and analyzing mobile concurrent systems. The main features of MWB are checking open bisimulation equivalences [26] and modal μ -calculus formulas using a sequent-calculus based model checker.

Other works considered automata-based representations of finite control π -calculus agents, with the goal of reusing the equivalence checkers and model checkers available for automata. Several decidability results about the strong and weak equivalence of π -calculus agents under certain assumptions on name spaces were presented in [6]. In a similar line of work, [21] introduce an *irredundant unfolding* notion that enables to check efficiently bisimilarity of finitary agents using an ordinary partition refinement algorithm, and also to minimize single agents. This line of work was continued in [8] by associating ordinary finite state automata to equivalent π -calculus agents using HD-automata as intermediate representation. This enabled to reuse the automata-based verification environment JACK for analyzing mobile processes, both by means of equivalence checking (using the MAUTO tool) and by model checking formulas specified in π -logic (a variant of modal μ -calculus dedicated to the π -calculus), by translating them into ACTL and applying the AMC model checker of the JACK environment.

A logical encoding of the operational semantics of the π -calculus into MMC processes was proposed in [30]. MMC is a model checker for mobile systems which builds on XMC, a model checker for Milner's value passing CCS implemented using the XSB tabled logic-programming engine. This connection allows the specification of correctness properties in an expressive subset of the π -logic and their verification using MMC. A probabilistic / stochastic version of the π -calculus was considered in [23], where an automated procedure was proposed for generating first the corresponding symbolic transition graphs, and second Markov decision processes or continuous-time Markov chains. These models can be used as input of existing probabilistic model checkers such as PRISM, where properties are typically specified using the temporal logics PCTL and CSL.

Compared to these works, we chose to follow here a different approach, by translating a π -calculus specification into an equivalent description in a high-level language equipped with tools for the generation and manipulation of the underlying transition system. This translation-based approach was subject to several proposals concerning various languages. In [2], the authors show how to map the π -calculus into the MONSTR graph rewriting language. This work, which was not targeted to verification purposes, illustrated the convenience of representing an evolving network of communicating agents in a graph manipulation formalism, but also pointed out the heavy cost in practice of faithfully implementing the communication primitive of mobile process calculi.

Another way of analyzing π -calculus specifications, proposed independently in [29,27], consists in translating them in PROMELA and verifying LTL formulas using the SPIN model checker. As regards channel mobility, PROMELA is suitable as a target language because it allows channel names to be communicated between processes as ordinary data values. The rule-based translation of [29] was implemented in the PI2PROMELA tool and successfully applied to model the Bluetooth service discovery protocol. In [27] a different translation is proposed (only for the monadic π -calculus) and implemented as an add-on to the MWB tool. The verification of LTL properties on the generated PROMELA code requires the manual specification of an environment whose role is to close the PROMELA description and to define the variables needed in the LTL formulas. Therefore, as observed in [27], the verification approach based on translation to PROMELA cannot be completely automated. Moreover, no attempt is made in [29,27] to justify that the translations proposed preserve the π -calculus semantics. Indeed, PROMELA is not equipped with an LTS semantics and the underlying state space model is suited mainly for LTL model checking in the state-based setting, whereas the π -calculus semantics relies on LTSS and bisimulation relations.

Finally, another group of works aimed at modelling the mobility in the LOTOS specification language. A first method for modelling dynamic communication structures by encoding link names as data values was proposed in [9], together with a sufficient condition on the communication structures (binary group communication) guaranteeing that the modelling is possible. This method is illustrated in [9] by specifying the handover procedure implemented in a mobile telecommunication network. In [22], the authors introduce M-LOTOS, a mobile

extension of LOTOS based on the π -calculus, which preserves the other LOTOS specification styles. An operational semantics of M-LOTOS and a notion of early bisimulation are defined, and the usage of the language is illustrated by several examples, including an ODP trader.

Our translation from π -calculus to LOTOS NT makes the state-of-the-art verification tools of CADP directly available for analyzing π -calculus specifications: μ -calculus model checking with EVALUATOR [17], equivalence checking (branching, weak, etc.) with BISIMULATOR [3], compositional and distributed verification, rapid prototyping, etc.

7 Conclusion and Future Work

We have presented a translation from the finite control fragment of the π -calculus to LOTOS NT, which is one of the input languages of the CADP toolbox. Consequently, this translation makes it possible to use all the state-of-the-art verification tools of CADP to analyze π -calculus specifications. The translation is completely automated by the PIC2LNT tool and validated on many examples. The restriction to the finite control fragment of the π -calculus, first considered in [21,6], does not hamper the practical usability of the language: even if the number of π -calculus agents must be statically known, the mobility of communication channels can be fully exploited.

We plan to continue our work by extending the π -calculus with data-handling features, with the goal of widening its possible application domains. This can be done by extending the language grammar and the translation to support typed variables and data expressions. As language for describing data, a natural candidate would be LOTOS NT itself: indeed, the data types and functions used in the π -calculus specification could be described in LOTOS NT and directly incorporated to the LOTOS NT code produced by translation. This would result in an applied π -calculus, such as the variant of the calculus proposed in [1] for the verification of security properties.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just Fast Keying in the Pi-Calculus. *ACM Trans. Inf. Syst. Secur.* 10(3) (2007)
2. Banach, R., Balazs, J., Papadoupoulos, G.: A Translation of the Pi-Calculus Into MONSTR. *J. UCS* 1(6), 339–398 (1995)
3. Bergamini, D., Descoubes, N., Joubert, C., Mateescu, R.: BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 581–585. Springer, Heidelberg (2005)
4. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.0). In: *INRIA/VASY*, 107 pages (March 2010)
5. OASIS Technical Committee. *Web Services Business Process Execution Language Version 2.0* (2007)
6. Dam, M.: On the Decidability of Process Equivalences for the pi-Calculus. *Theor. Comput. Sci.* 183(2), 215–228 (1997)

7. Deng, S., Wu, Z., Zhou, M., Li, Y., Wu, J.: Modeling Service Compatibility with Pi-Calculus for Choreography. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 26–39. Springer, Heidelberg (2006)
8. Ferrari, G.L., Ferro, G., Gnesi, S., Montanari, U., Pistore, M., Ristori, G.: An Automated Based Verification Environment for Mobile Processes. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 275–289. Springer, Heidelberg (1997)
9. Fredlund, L.-Å., Orava, F.: Modelling Dynamic Communication Structures in LOTOS. In: Proc. of FORTE 1991. IFIP Transactions, vol. C-2, pp. 185–200. North-Holland, Amsterdam (1991)
10. Garavel, H.: OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation and Testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
11. Garavel, H., Lang, F., Mateescu, R.: Compiler Construction using LOTOS NT. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 9–13. Springer, Heidelberg (2002)
12. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
13. Garavel, H., Sighireanu, M.: A Graphical Parallel Composition Operator for Process Algebras. In: Proc. of FORTE/PSTV 1999. IFIP, pp. 185–202. Kluwer Academic Publishers, Dordrecht (October 1999)
14. Lucchi, R., Mazzara, M.: A Pi-Calculus based Semantics for WS-BPEL. *J. Log. Algebr. Program.* 70(1), 96–118 (2007)
15. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization, Genève (September 2001)
16. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, Genève (September 1989)
17. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
18. Milner, R.: *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs (1989)
19. Milner, R.: *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, Cambridge (1999)
20. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes. *Information and Computation* 100(1), 1–77 (1992)
21. Montanari, U., Pistore, M.: Checking Bisimilarity for Finitary Pi-Calculus. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 42–56. Springer, Heidelberg (1995)
22. Najm, E., Stefani, J.-B., Février, A.: Towards a Mobile LOTOS. In: FORTE 1995. IFIP Conference Proceedings, vol. 43, pp. 127–142. Chapman & Hall, Boca Raton (1995)
23. Norman, G., Palamidessi, C., Parker, D., Wu, P.: Model Checking Probabilistic and Stochastic Extensions of the Pi-Calculus. *IEEE Trans. Software Eng.* 35(2), 209–223 (2009)
24. Parrow, J.: An introduction to the pi-calculus. In: *Handbook of Process Algebra*, ch. 8, pp. 479–544. North-Holland, Amsterdam (2001)
25. Puhlmann, F.: Why Do We Actually Need the Pi-Calculus for Business Process Management? In: Proc. of BIS 2006, GI. LNI, vol. 85, pp. 77–89 (2006)

26. Sangiorgi, D.: A Theory of Bisimulation for the pi-Calculus. *Acta Inf.* 33(1), 69–97 (1996)
27. Song, H., Compton, K.J.: Verifying Pi-Calculus Processes by Promela Translation. Technical Report CSE-TR-472-03, University of Michigan, USA (2003)
28. Victor, B., Moller, F.: The Mobility Workbench – A Tool for the π -Calculus. In: Dill, D.L. (ed.) *CAV 1994*. LNCS, vol. 818, pp. 428–440. Springer, Heidelberg (1994)
29. Wu, P.: Interpreting Pi-Calculus with Spin/Promela. In: *Proc. of NCTCS 2003*. Computer Science, vol. 8(Suppl.), pp. 7–9 (2003)
30. Yang, P., Ramakrishnan, C.R., Smolka, S.A.: A Logical Encoding of the Pi-Calculus: Model Checking Mobile Processes using Tabled Resolution. *STTT* 6(1), 38–66 (2004)

A Dispatcher Web Service Translated to LOTOS NT

We show below an excerpt of the LOTOS NT code (processes *MAIN* and *Dispatcher_4* corresponding to the agents *Main* and *Dispatcher*) generated by the PIC2LNT translator from the π -calculus specification of the dispatcher Web service given in Section 5. The names of LOTOS NT processes are indexed by the number of gates in \overline{G} (not counting the G_{pub} and G_{priv} gates).

```

process MAIN [PUBLIC,PRIVATE:any] is
  var req, a, b, c:Chan in
    req:=req(new_id()); a:=a(new_id()); b:=b(new_id()); c:=c(new_id());

    hide G0:any in par G0 in hide G1:any in par G1 in
      hide G2:any in par G2 in hide G3:any in par G3 in
        Client_4 [PUBLIC,PRIVATE,G0,G1,G2,G3] (req,a,b,c,2)
        || Dispatcher_4 [PUBLIC,PRIVATE,G0,G1,G2,G3] (req,6) end par end hide
        || Server_3 [PUBLIC,PRIVATE,G0,G1,G2] (a,14) end par end hide
        || Server_2 [PUBLIC,PRIVATE,G0,G1] (b,30) end par end hide
        || Server_1 [PUBLIC,PRIVATE,G0] (c,31) end par end hide
      end var
    end process

process Dispatcher_4[PUBLIC,PRIVATE,G0,G1,G2,G3:any] (req:Chan,pid:Nat) is
  select var k,x:Chan, s:Nat in
    G0 (!req, ?k, ?x, ?s, !pid) [] G1 (!req, ?k, ?x, ?s, !pid) []
    G2 (!req, ?k, ?x, ?s, !pid) [] G3 (!req, ?k, ?x, ?s, !pid) []
    PUBLIC (!req, ?k, ?x, !false) where is_public(req) []
    PRIVATE (!req, ?k, ?x, !false) where not(is_public(req))
  end select ;
  select var r:Nat in
    G0 (!k, !x, !pid, ?r) [] G1 (!k, !x, !pid, ?r) []
    G2 (!k, !x, !pid, ?r) [] G3 (!k, !x, !pid, ?r) []
    PUBLIC (!k, !x, !true) where is_public(k) []
    PRIVATE (!k, !x, !true) where not(is_public(k))
  end select ; Dispatcher_4 [PUBLIC,PRIVATE,G0,G1,G2,G3] (req,pid)
end process

```

Systematic Translation Rules from ASTD to Event-B

Jérémy Milhau^{1,2}, Marc Frappier¹, Frédéric Gervais², and Régine Laleau²

¹ GRIL, Département Informatique, Université de Sherbrooke,
2500 boulevard université, Sherbrooke J1K 2R1, Québec, Canada
{Jeremy.Milhau,Marc.Frappier}@USherbrooke.ca

² Université Paris-Est, LACL, IUT Sénart Fontainebleau,
Département Informatique, Route Hurtault, 77300 Fontainebleau, France
{Frederic.Gervais,Laleau}@u-pec.fr

Abstract. This article presents a set of translation rules to generate Event-B machines from process-algebra based specification languages such as ASTD. Illustrated by a case study, it details the rules and the process of the translation. The ultimate goal of this systematic translation is to take advantage of Rodin, the Event-B platform to perform proofs, animation and model-checking over the translated specification.

Keywords: Process algebra, Translation rules, Systematic translation, ASTD, Event-B.

1 Introduction

Information Systems (IS) are taking an increasingly important place in today's organizations. As computer programs connected to databases and other systems, they induce increasing costs for their development. Indeed, with the importance of the Internet and their high computer market penetration, IS have become the de-facto standard for managing most of the aspects of a company strategy. In the context of IS, formal methods can help improving the reliability, security and coherence of the system and its specification. The APIS (Automated Production of Information system) project [7] offers a way to specify and generate code, graphical user interfaces, databases, transactions and error messages of such systems, by using a process algebra-based specification language. However, process algebra, despite their formal aspect, are not as easily understandable as semi-formal graphical notations, such as UML [13]. In order to address this issue, a formal notation combining graphical elements and process algebra was introduced: Algebraic State Transition Diagrams (ASTD) [8]. Using ASTD, one can specify the behavior of an IS. The interpreter i ASTD [15] can efficiently execute ASTD specifications. However, there is no tool allowing proof of invariants or property check over an ASTD specification. This paper aims to define systematic translation rules from an ASTD specification to Event-B [2] in order to model check or prove properties using tools of the RODIN platform [3]. Moreover, translation results will allow to bridge other process algebras (like EB³ [6

or CSP [11]) with Event-B as they share a similar semantics with ASTD. Event-B is first introduced to the reader in Section 2. An overview of ASTD and a case study will be then presented. This case study will help readers unfamiliar with ASTD to discover the formalism in Section 3. The Event-B machine resulting from translation rules applied to this case study will be described as well as rules and relevant steps of translation in Section 4. Finally, future work and evolution perspectives will be presented.

2 Event-B Background

Event-B [2] is an evolution of the B method [1] allowing to model discrete systems using a formal mathematical notation. The modeling process usually follows several refinement steps, starting from an abstract model to a more concrete one in the next step. Event-B specifications are built using two elements: *context* and *machine*. A *context* describes the static part of an Event-B specification. It consists of declarations of *constants* and *sets*. *Axioms*, which describe types and properties of constants and sets, are also included in the context. A *machine* is the dynamic part of an Event-B specification. It has a state consisting of several *variables* that are first initialized. Then *events* can be executed to modify the state. An event can be executed if it is enabled, *i.e.* all the conditions prior to its execution hold. These conditions are named *guards*. Among all enabled events, only one is executed. In this case, substitutions, called *actions*, are applied over variables. All actions are applied simultaneously, meaning that an event is atomic. The state resulting from the execution of the event is the new state of the machine, enabling and disabling events. Alongside the execution of events, *invariants* must hold. An invariant is a property of the system written using a first-order predicate on the state variables. In order to ensure that invariants hold, *proofs* are performed over the specification.

3 ASTD Background

ASTD is a graphical notation linked to a formal semantics allowing to specify systems such as IS. An ASTD defines a set of traces of actions accepted by the system. ASTD actions correspond to events in Event-B. Event-B actions and substitutions, as they modify the state of an Event-B machine, can be binded to the change of state in ASTD. The ASTD notation is based on operators from the EB³ [9] method and was introduced as an extension of Harel's Statecharts [10]. An ASTD is built from transitions, denoting action labels and parameters, and states that can be elementary (as in automata) or ASTD themselves. Each ASTD has a type associated to a formal semantics. This type can be automata, sequence, choice, Kleene closure, synchronization over a set of action labels, choice or interleaving quantification, guard and ASTD call. One of ASTD most important features is to allow parametrized instances and quantifications, aspects missing from original Statecharts. An ASTD can also refer to attributes, which are defined as recursive functions on traces accepted by the ASTD, as in the EB³ method.

Such a recursive function compares the last action of the trace and maps each possible action to a value of the attribute it is defining. Computing this value may imply to call the function again on the remaining of the trace.

3.1 ASTD Operators

Several operators, or ASTD types, are used to specify an IS. We detail them in the following paragraphs. Operators will be further illustrated in Section 3.2 with the introduction of a case study.

Automata. In an ASTD specification, one can describe a system using hierarchical states automata with guarded transitions. Each automata state is either elementary or another ASTD of whichever type. Transitions can be on states of the same depth, or go up or down of one level of depth. A transition decorated by a bullet (\bullet) is called a final transition. A final transition is enabled when the source state is final. As in Statecharts, an history state allows the current state of an automata ASTD to be saved before leaving it in order to reuse it later.

Sequence. A sequence is applied to two ASTD. It implies that the left hand side ASTD will be executed and will reach a final state before the right hand side ASTD can start. There is no immediate equivalent of this operator in Harel's Statecharts, but its behavior can be reproduced with guards and final transitions. A sequence ASTD is noted with a double arrow \Rightarrow .

Choice. A choice, noted $|$ allows the execution of only one of its operands, like a choice in regular expressions or in process algebras. The choice of the ASTD to execute is made on the first action executed. After the execution of the first action, the chosen ASTD is kept until it terminates its execution. If both operands of a choice ASTD can execute the first action, then a nondeterministic choice is made between the two ASTD. The behavior of a choice ASTD can be modeled in Statecharts using internal transition from an initial state, in a similar way to automata theory with ϵ transitions.

Kleene Closure. As in regular expressions, a Kleene closure ASTD noted $*$ allows its operand to be executed zero, one or several times. When the state of its operand is final, a new iteration can start. There is no similar operator in Statecharts, but the same behavior can be reproduced with guards and transitions.

Synchronization Over a Set of Action Labels. As the name suggests, this operator allows the definition of a set of actions that both operands must execute at the same time. It is similar to Roscoe's CSP parallel operator \parallel_x . There are some similarities with AND states of Statecharts and synchronization ASTD. A synchronization over the set of actions Δ is noted $[[\Delta]]$. We derive two often used operators from synchronization : interleaving, noted $|||$, is the synchronization over an empty set ; parallel, noted $||$, synchronizes ASTD over the set of common actions of its operands, like Hoare's CSP $||$.

Quantified Interleaving. A quantified interleaving models the behavior of a set of concurrent ASTD. It sets up a quantification set that will define the number of instances that can be executed and a variable that can take a value inside the quantification set. Each instance of the quantification is linked to a single value, two different instances have two different values. This feature lacks in Statecharts, as we have to express distinctly each instance behavior, but was proposed as an extension and named “parametrized-and” state by Harel. A quantified interleaving of variable x over the set T is noted $\| \| x : T$.

Quantified Choice. A quantified choice, noted $| x : T$, lets model that only one instance inside a set will be executed. Once the choice is made, no more instances can be executed. As in quantified synchronization, the instance is linked to one value of a variable in the quantification set. An extension of Statecharts named a similar feature “parametrized-or” state.

Guard. Usually, guards are applied to transitions. With the guard ASTD, one can forbid the execution of an entire ASTD until a condition holds. The predicate of a guard can use variables from quantifications and attributes. A predicate $P(x)$ guarding an ASTD is noted $\implies P(x)$

ASTD call. An ASTD call simply links to other parts of the specification using the name of another ASTD. The same ASTD can be called several times, in different locations of the specification. It allows the designer to reuse ASTD in the same specification and helps synchronize processes. An ASTD call is made by writing the name of the ASTD called and its parameters (if any).

3.2 An ASTD Case Study

In order to present features and expressiveness of the ASTD notation to the reader, Fig. 1 introduces the case study that will be used throughout this paper. This ASTD models an information system designed to manage complaints of customers in a company. In this system, each complaint is issued from a customer relatively to a department. This example is inspired from [19]. The **main** ASTD, whose type is a synchronization over common actions, describes the system as a parallel execution of interleaved customers and departments processes. The IS lifecycle of a given customer is described by the parametrized ASTD **customer** (u), the same applies for the description of the company departments in the ASTD **department** (d). In the initial state, a customer or a department must be created. Then complaints regarding these entities can be issued. This is described using an ASTD call. The final transition means that the event can be executed if the source state is final. In our case, in order to delete a customer or a department, any related complaints must be closed. Finally, the ASTD describing the checking and processing of a complaint c issued by customer c about department d is given by **complaint** (c, d, u). After registering a complaint in the system, it must be evaluated by the company and a questionnaire is sent to the customer in order to detail his/her complaint. The specification

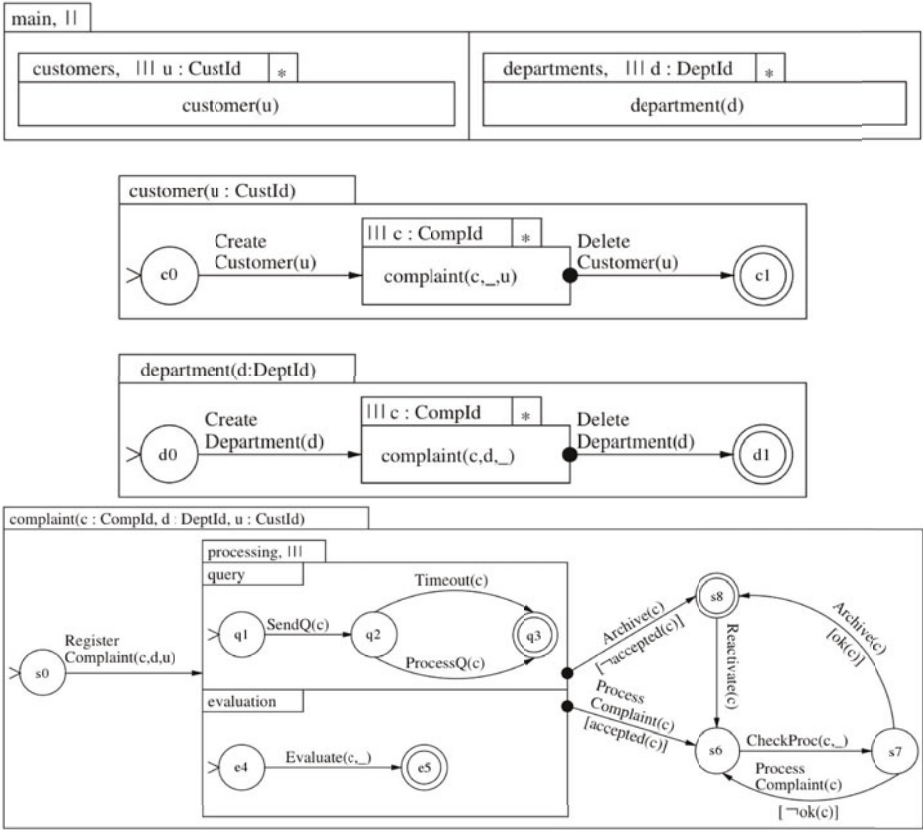


Fig. 1. An ASTD specification describing a complaint management system

takes into account the possibility that the customer does not answer the questionnaire with the $Timeout(c)$ event. Then, if the complaint is accepted and the questionnaire received or timed out, a check is performed. In case of refusal of the complaint, it is archived, but it can be reactivated later. The only final state is state $s8$, meaning that the complaint was archived (solved or not). In this specification, no attribute modification is performed. An ASTD only describes traces and has no consequences on updates to be performed against IS data, such as attributes that are stored in databases. However, an ASTD can access attribute values to use them in guards, as shown in both $Archive(c)$ actions.

3.3 Motivations

ASTD are not the only way to specify IS behavior. The UML-B [18] method introduces a behavior specification in the form of a Statecharts. Using Statecharts, it is easy to describe an ordered sequence of actions whereas using B, it is easier to model interleaving events. A systematic translation of Statecharts into B

machines is proposed by [17]. Compared to Statecharts, ASTD offer additional operators to combine ASTD in sequence, iteration, choice and synchronisation. When a UML-B specification models a system, it can only describe the life-cycle of a single instance of a class whereas ASTD specification models the behavior of all instances of all classes of the system. A new version of UML-B [14] introduces the possibility to refine class and Statecharts as part of the modeling process, and can translate it into Event-B. The UML-B approach can describe the evolution of entity attributes using B substitutions, a feature that ASTD lacks. csp2B [4] provides better proofs (on the B machine) and model checking (on the CSP side) tools than Statecharts but lacks the visual representation of the specification given by UML Statecharts. It is also limited to a subset of CSP specifications, where the quantified interleaving operator must not be nested. ASTD aims to be a compromise in both visual and synchronization aspects. On the other hand, ASTD lacks proofs and model checking allowed by the B side of UML-B and csp2B approaches. In order to answer this issue, a systematic translation of ASTD specifications into Event-B is proposed.

The choice between classical B and Event-B was made at an early stage by comparing tools and momentum of both methods. It appears that community efforts and tool development are currently focused on Event-B. Despite the fact that classical B offers some convenient notation such as IF / THEN statements or operation calls, Event-B appeared as a good compromise for our efforts. Classical B translation rules inspired by Event-B rules might be written.

4 Translation

Translation from ASTD to Event-B is achieved in several steps. Fig. 2 presents the architecture of the translation process. A context derived from ASTD operators introduces constants and sets needed to code their semantics. This context is the same in all translations and is described by Table 1. It codes elements from the semantics of all types of ASTD except automata, and is inspired of mathematical definition of ASTD semantics. Constants, sets and axioms defined in this context may be re-used in other part of the Event-B translation, hence this context is extended by a translation specific context. Automata states are translated into such a specific context since automata states depend on the ASTD specification

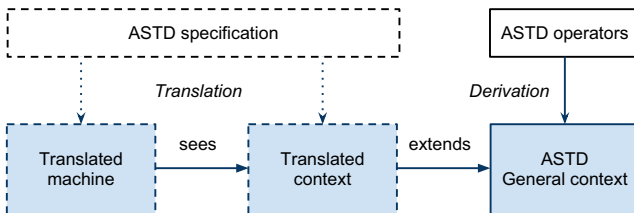


Fig. 2. The architecture resulting from the translation process

Table 1. Event-B representation of ASTD states

ASTD state	State domain	Initial State
choice	$State \in \{ \text{none}, \text{first}, \text{second} \}$	none
sequence	$State \in \{ \text{left}, \text{right} \}$	left
Kleene closure	$State \in \{ \text{neverExecuted}, \text{started} \}$	neverExecuted
synchronization	-	-
quantified choice	$State \in \{ \text{notMade}, \text{made} \} \leftrightarrow \text{QUANTIFICATIONSET}$	notMade $\mapsto 0$
quantified synch	$State \in \text{QUANTIFICATIONSET} \rightarrow \text{STATESET}$	initial for all
guard	$State \in \{ \text{checked}, \text{notChecked} \}$	notChecked
ASTD call	-	-

to translate. For each ASTD, a variable and an invariant corresponding to its type are created. The invariant associates the variable to the set of values it can take, as defined in both contexts. In the following sections, we provide translation rules for each ASTD type, generating appropriate contexts and machines.

4.1 Automata

The first part of automata translation concerns the static part, the context. Several elements are introduced in the context: states, initial states, final states and transition functions.

States. States from automata ASTD are represented as constants and grouped into state sets in order to facilitate later use. Even hierarchical states are represented by a constant.

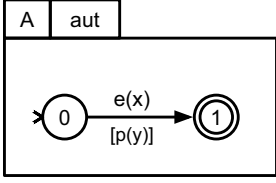
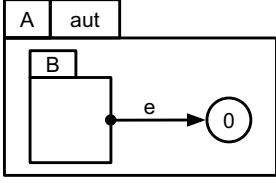
Initial States. Since an ASTD can be reset by the execution of a Kleene closure, initial states are defined as separate constants. They are also useful in the initialisation event of the machine generated in next step of our translation.

Final Predicates. A final predicate is a function taking a state as argument and returning TRUE or FALSE depending if the state is final or not. The number of arguments depends on the type of the ASTD. This predicate is useful in the case of final transitions, sequences or Kleene closures, when transitions are activated if, and only if, a state is final. Hence, a final predicate is written for each ASTD type in the context common to all translations.

Transition Functions. A transition function for each action label is generated. It takes as argument the current state of an automata ASTD and returns the resulting state. Transition functions are deterministic and partial.

The generated context for our case study defines 40 constants, 5 sets and 29 axioms. It is not presented here for the sake of conciseness. Then, for the dynamic part, for each distinct action label in the translated automata ASTD, a single event will be produced. If the action has a guard, a WHEN clause *i.e.* a

Table 2. Automata ASTD to Event-B translation rules

Automata ASTD pattern	Added to the context	Modifications on the machine
	<p>SETS</p> <p>StatesA</p> <p>CONSTANTS</p> <p>s0, s1 initA, isFinalA, TransE</p> <p>AXIOMS</p> <p>ax1 : partition(StatesA, {s0}, {s1}) ax2 : initA = s0 ax3 : isFinalA = {s0 ↦ FALSE, s1 ↦ TRUE} ax4 : TransE = {s0 ↦ s1}</p>	<p>Event $e \hat{=}$</p> <p>any x</p> <p>where</p> <p>g1 : $x \in XSET$ g2 : $P(y)$ g3 : $StateA \in$ $dom(TransE)$</p> <p>then</p> <p>a1 : $StateA :=$ $TransE(StateA)$</p> <p>end</p>
	<p>CONSTANTS</p> <p>isFinalB</p> <p>AXIOMS</p> <p>ax1 : $isFinalB = \dots$ // Depends ... on B type</p>	<p>Event $e \hat{=}$</p> <p>where</p> <p>g1 : $isFinalB(StateB)$ $= TRUE$...</p>

guard, is generated. If the ASTD action has arguments (in the case of quantified variable for instance), an ANY clause is built accordingly and a guard specifying a type for the variable is added. Then a guard testing that the execution of the action is allowed *i.e.* the current state is in the domain of the transition function of the event. The modification of the state is applied by generating a THEN substitution.

Translation rules for automata ASTD are presented in Table 2. When a transition, an initial state or a final state is found, the first rule applies. In the case of a final transition, the second rule then applies. In the second pattern translation, the guard numbered g1 of Table 2 is added to event e that was generated by applying first rule. In our case study, the second rule is applied for the $ProcessComplaint(c)$ action. The guard added in this case is described by guard $grdAutomata$.

$grdAutomata : isFinalProcessing(isFinalQuery(StateQuery(c)) \mapsto isFinalEvaluate(StateEval(c))) = TRUE$

Constants $isFinalX$ and $StateX$ refer to ASTD **X** in Fig. 1. An interleaved state is final if, and only if, both of its operand states are final. For this reason, guard $grdAutomata$ checks if both states of **Query** and **Evaluation** ASTD are final. A pair (x, y) is noted $x \mapsto y$ in Event-B.

The action $CreateCustomer(u)$ is translated into the event described below. $grd1$ describes the set in which the parameter u can take its value. $grd2$ verifies

that a customer is in a state of the domain of transition function $TransCreateCustomer$. $act1$ describes the state update for action $CreateCustomer(u)$: it only modifies the state of customer u according to the transition function $TransCreateCustomer$.

```

Event  $CreateCustomer \hat{=}$ 
  any
     $u$ 
  where
     $grd1 : u \in USERSET$ 
     $grd2 : StateCustomer(u) \in dom(TransCreateCustomer)$ 
  then
     $act1 : StateCustomer(u) := TransCreateCustomer(StateCustomer(u))$ 
  end

```

4.2 Sequence

Because of the number of possibilities to determine whether or not a sequence can switch from **left** state to **right** state, an extra event is introduced. This event is similar to an internal event of the IS and will verify that all the conditions for the switch from left to right side to happen holds and then change the state of the sequence. For example, if an ASTD named **A** is a sequence of ASTD **B** and **C**, the generated event will be called $switchSequenceA$. Then, in order to ensure that the current state allows the execution of every events of ASTD **B** and **C**, a guard is added to each event of **B** and **C** to check if the state of ASTD **A** is *left* or *right* respectively. As for automata, a final predicate must be generated in the context for ASTD **B** state. Translation rule is described in the following table.

Sequence ASTD pattern	Modifications on the machine
	<pre> Event $switchSequenceA \hat{=}$ where $g1 : isFinalB(StateB) = TRUE$ then $a1 : StateA := right$ end </pre>
	<pre> Event $e \hat{=}$ where $g2 : StateA = left$... </pre>
	<pre> Event $f \hat{=}$ where $g3 : StateA = right$... </pre>

4.3 Choice

A choice ASTD can be in three states as described by the general ASTD context: **none** when the choice is not made yet, **first** or **second** depending of the side chosen. The translation rule for a choice ASTD is presented in the following table. If an ASTD named **A** is a choice between ASTD **B** and **C**, then a guard and an

action are added to each event. Events from **B** will receive guard **g1** and action **a1**. A similar transformation of events from ASTD **C** is also needed with guard **g2** and action **a2**.

Choice ASTD pattern	Modifications on the machine
	<pre> Event e ≐ where g1 : StateA = first ∨ StateA = none ... then a1 : StateA := first ... Event f ≐ where g2 : StateA = second ∨ StateA = none ... then a2 : StateA := second ... </pre>

4.4 Kleene Closure

When an iteration of a Kleene closure ASTD is completed, its operand must be reset to initial state. For this reason, an additional event is generated. In the IS, this event is internal and hidden, in the ASTD specification, the semantics off Kleene operator handles the process, but in Event-B the reset must be described. This event will be activated when the its operand is final, and will reinitialize all sub-states in the hierarchy. The following table details the resulting Event-B machine.

Kleene ASTD pattern	Modifications on the machine
	<pre> Event lambdaA ≐ where g1 : isFinalB(StateB) = TRUE then a1 : StateB := initB And all sub states ... end Event e ≐ then a2 : StateA := started ... </pre>

As presented for automata and sequence, a final predicate must be generated in the context for ASTD under the Kleene closure operator.

4.5 Synchronization Over a Set of Action Labels

For actions that are not synchronized, nothing is introduced or modified by the translation of synchronization ASTD. This is the case for interleaving ASTD and action labels not common to both operands of the parallel operator. In the case of a synchronized action, guards from both operand must be put in conjunction, and substitutions applied conjointly.

Synchronization ASTD pattern	Modifications on the machine
	<p>Event $e \hat{=}$</p> <p>where</p> <p>gB : guardsfromBASTD</p> <p>gC : guardsfromCASTD</p> <p>...</p> <p>then</p> <p>$a1$: StateB := ...</p> <p>$a2$: StateC := ...</p> <p>...</p>

In our case study, the only synchronization ASTD is **main**. Common actions of both sides are only actions appearing in the ASTD **complaint** (c, d, u) . For each one of the generated events of **complaint** (c, d, u) , the guards **readyInCustomer** and **readyInDepartment** must hold. cc and dc states correspond to states where the customer and the department respectively are in the complaint quantified interleaving ASTD.

readyInCustomer : $StateCustomer(AssociationCustomer(c)) = cc$

readyInDepartment : $StateDepartment(AssociationDepartment(c)) = dc$

Theses guards check that the customer associated to the complaint c is in the state allowing him to complain *i.e.* created and not deleted, and if the department associated to the complaint c exists in the IS.

4.6 Quantified Interleaving

The quantified interleaving does not introduce additional constraints to events. The following table shows how variables induced by quantified interleaving are handled in events.

Quantified interleaving ASTD pattern	Modifications on the machine
	<p>Event $e \hat{=}$</p> <p>any</p> <p>x</p> <p>where</p> <p>$g1$: $x \in XSET$</p> <p>$g2$: $StateA(x) = \dots$</p> <p>...</p> <p>then</p> <p>$a1$: $StateA(x) := \dots$</p> <p>...</p>

Entities and associations patterns are common in EB³ and ASTD as mentioned in [9]. Such pattern are expressed using interleaving quantifications. In order to code in Event-B the association between several entities, a table variable must register their link. In our case study, we can see that a 1-n association between

a customer and a complaint is created. When a complaint is created, an unique customer u is linked to the complaint c . The same applies to the department associated to the complaint. An Event-B variable is created in order to save the link between a complaint and a customer (respectively a department) and is updated whenever a complaint is registered in the system.

4.7 Quantified Choice

Similarly to the choice operator, the quantified choice implies that for all events using it, a check is performed about whether the choice was made or not. In the case of an action labeled e and taking x as a parameter, where x is the variable of a quantified choice ASTD named \mathbf{A} then the guard $g2$ described in the following table must hold. The substitution $a1$ must also be executed in case this is the first call of an action with this quantified variable. All the events of ASTD \mathbf{A} will be modified to include this guard and substitution.

Quantified choice ASTD pattern	Modifications on the machine
	<pre> Event $e \hat{=}$ any x where $g1 : x \in XSET$ $g2 : StateA = (qNone \mapsto 0) \vee StateA = (qSome \mapsto x)$... then $a1 : StateA := (qSome \mapsto x)$... </pre>

4.8 Guard

There are two cases for guard state: the guard was checked and held when we executed an event ; the guard did not hold, and no event was executed. These cases are handled with guard $g1$ and substitution $a1$ for a guard ASTD named \mathbf{A} guarded with predicate $P(x)$. All the events of ASTD \mathbf{A} will be modified to include this guard and substitution.

Guard ASTD pattern	Modifications on the machine
	<pre> Event $e \hat{=}$ any x where $g1 : StateA = checked \vee (StateA = notChecked \wedge P(x))$... then $a1 : StateA := checked$... </pre>

4.9 Process Call

An ASTD that calls other ASTD does not need any constraint over its actions in Event-B. The translation will be achieved as if the entire called ASTD was substituted for the ASTD call. We do not deal with recursive ASTD calls yet.

When the translation process is completed, we can now access all the tools offered by Rodin to animate, model check and prove elements of the translated ASTD specification.

5 Animation and Model Checking of the Case Study

The final generated system, a context and a machine, translated from our case study represents 270 lines of Event-B, including 40 constants, 5 sets and 29 axioms for the static part and 7 variants, 7 invariants, 17 events (one for initialization, 13 representing ASTD actions and 3 internal events for Kleene Closure induced resets) representing 57 guards and 33 actions for the dynamic part. During the construction of translation rules, animation helped to correct rules, to improve the quality of translation rules and to factor contexts in order to separate static elements from machine. It was chosen to limit the size of quantification sets to three elements each. Only three departments, customers and complaints can be registered inside the system at any time. The screen capture was taken after the execution of 150 events and shows the state of variables of the machine. In order to informally verify the consistency of the Event-B machine with the initial ASTD specification, we generated a set of traces of events executed via the ProB animator. Then, for each trace, we removed the internal events introduced by the translation process such as `lambdaComplaint(c)`. Then we interpreted the initial ASTD specification with *i*ASTD and executed the traces. We could not find a trace of events that could not be interpreted by *i*ASTD. A more formal proof of the consistency of the translation must be performed, but first results are encouraging. Formal proof of translation rules is work in progress, and will be based on simulation.

Regarding the Event-B machine, 86 proof obligations were generated and 62 were automatically proved. The 24 remaining are proved manually and involved functional and set operators that are known for not being proved automatically. The manual proofs raised no specific difficulty. This Event-B specification was model checked for deadlocks and invariant violations using the consistency checking feature of ProB. More than 111 500 nodes were visited and 226 000 transitions activated. No deadlock nor invariant violation were found. More invariant properties might be written in order to be proved. Since ASTD only focuses on event control and not on event effects on the IS, when an event is executed, there is no way to know only by looking at the ASTD specification how IS state will evolve. Hence, no invariant can be generated during the translation. But it could be interesting to express invariants on ASTD as it was done with Statecharts [16]. For instance we could add an invariant to ASTD **Department** saying that whenever transition `DeleteDepartment(d)` is active, no complaint about this department must be registered in the system.

6 Limitations, Conclusion and Future Work

We have presented a set of translation rules allowing generation of Event-B contexts and machines from ASTD specifications. The animation of the resulting

machine using ProB [12] animator helped to find errors and to tweak translation rules. Kleene closure and sequence operators were the most tricky to translate since these operators defines the ordering of events and because they introduce additional events in order to code semantics of ASTD in Event-B. A formal proof of the translation rules will be performed in order to entrust the translation process.

Refinement is one of the most important features of Event-B modelling process. In our approach, this aspect is missing. Indeed, we are translating an ASTD specification modelling a concrete system. Because of that, there is no need to refine the Event-B machine resulting from the translation process. It would have been relevant to introduce refinement in the translation process if a similar notion existed in ASTD, but it is currently not the case. Proof is an important aspect of Event-B that our approach would like to take advantage of. Alongside with formal IS specification, we advocate writing security or functional properties during the modeling process. This way, properties can be checked against the system as soon as it is modeled. Expressing these properties as Event-B invariants and proving invariant preservation in the translated machine is an important step of IS specification validation. Another feature of Event-B we do not use is composition. This may be very useful for the translation of some ASTD operators such as synchronization. It could lead to a more modular approach of translation, in a way similar to ASTD.

It would be interesting to compare the machine resulting of the translation process with a hand-written Event-B specification for the same system. Indeed, we would like to know if the automatic prover can do the same job with the hand-written and the translated machine. This study is work in progress and may result in an evolution of translation rules. Another step that we currently work on is to implement an ASTD modeler as a Rodin plugin. Using benefits from The Eclipse Graphical Modeling Framework (GMF) [5], a graphical editor could be used to build complete IS specifications. One could interpret them using the *i*ASTD [15] interpreter and then translate them to Event-B on the fly in order to perform model checking or proofs. This integrated tool would allow a great flexibility and would combine advantages of process algebra's power of expression, graphical representation's ease of understanding and Event-B's tools for proving, checking and animating.

Acknowledgements. The authors would like to thank the anonymous referees for their insightful comments. This research is financed by ANR (France) as part of the SELKIS project (ANR-08-SEGI-018) and supported by NSERC (Canada).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.R.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)

3. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)
4. Butler, M.: csp2b: A practical approach to combining CSP and b. *Formal Aspects of Computing* 12(3), 182–198 (2000)
5. Eclipse Consortium: Eclipse graphical modeling framework (gmf), <http://www.eclipse.org/modeling/gmf/?project=gmf>
6. Fraikin, B., Frappier, M.: Efficient symbolic computation of process expressions. *Science of Computer Programming* (2009)
7. Fraikin, B., et al.: Synthesizing information systems: the APIS project. In: Rolland, C., Pastor, O., Cavarero, J.L. (eds.) First International Conference on Research Challenges in Information Science (RCIS), Ouarzazate, Morocco, p. 12 (April 2007)
8. Frappier, M., Gervais, F., Laleau, R., Fraikin, B., St-Denis, R.: Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering* 4(3), 285–292 (2008)
9. Frappier, M., St-Denis, R.: EB³: an entity-based black-box specification method for information systems. *Software and System Modeling* 2(2), 134–149 (2003)
10. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of computer programming* 8(3), 231–274 (1987)
11. Hoare, C.A.R.: *CSP—Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs (1985)
12. Leuschel, M., Butler, M.: ProB: A model checker for b. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
13. Rumbaugh, J., Jacobson, I., Booch, G.: *The unified modeling language*. University Video Communications (1996)
14. Said, M.Y., Butler, M., Snook, C.: Language and tool support for class and state machine refinement in UML-B. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 579–595. Springer, Heidelberg (2009)
15. Salabert, K., Milhau, J., et al.: iASTD: un interpréteur pour les ASTD. In: *Atelier Approches Formelles dans l’Assistance au Développement de Logiciels (AFADL 2010)*, Actes AFADL, Poitiers, France, pp. 3–6 (June 9–11, 2010)
16. Sekerinski, E.: Verifying Statecharts with State Invariants. In: 13th IEEE International Conference on Engineering of Complex Computer Systems, pp. 7–14. IEEE, Los Alamitos (2008)
17. Sekerinski, E., Zurob, R.: Translating statecharts to b. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 128–144. Springer, Heidelberg (2002)
18. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15(1), 122 (2006)
19. Van Der Aalst, W.M.P.: The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers* 8(1), 21–66 (1998)

A CSP Approach to Control in Event-B

Steve Schneider¹, Helen Treharne¹, and Heike Wehrheim²

¹ Department of Computing, University of Surrey

² Institut für Informatik, Universität Paderborn

Abstract. Event-B has emerged as one of the dominant state-based formal techniques used for modelling control-intensive applications. Due to the blocking semantics of events, their ordering is controlled by their guards. In this paper we explore how process algebra descriptions can be defined alongside an Event-B model. We will use CSP to provide explicit control flow for an Event-B model and alternatively to provide a way of separating out requirements which are dependent on control flow information. We propose and verify new conditions on combined specifications which establish deadlock freedom. We discuss how combined specifications can be refined and the challenges arising from this. The paper uses Abrial’s Bridge example as the basis of a running example to illustrate the framework.

Keywords: Event-B, CSP, control flow, integration, consistency, deadlock-freedom.

1 Introduction

Event-B [1] is an elegant modelling language which is supported by a notion of *refinement* to enable descriptions of systems to be elaborated during refinement. Event-B has proven to be applicable in a wide range of domains, including distributed algorithms, railway systems and electronic circuits. The basic specification construct is a machine that comprises of a number of events in which control flow is implicit within their guards. Hence, Event-B can be classified as being a *state-based language*: control can only be modelled via state variables and guards on the state. On the other side, there are a number of specification formalisms with language support for *explicitly* specifying control, like statecharts, Petri nets or process algebras, which are however not good at specifying state. This observation has led to introducing *integrated formal methods*, combining state-based formalisms with control-oriented languages. Examples include combinations of (Object-)Z and CSP [17, 7, 11, 20], or closer to the approach presented here, those integrating B with a process algebra [3, 4, 19].

Though Event-B can be and is used for modelling control-intensive applications, it has recently been observed that explicit control specifications alongside Event-B machines are nevertheless beneficial [10]. Control specifications can serve two purposes: on the one hand they can make the control flow in the Event-B machine explicit, and thus enhance readability but also facilitate analysis. On

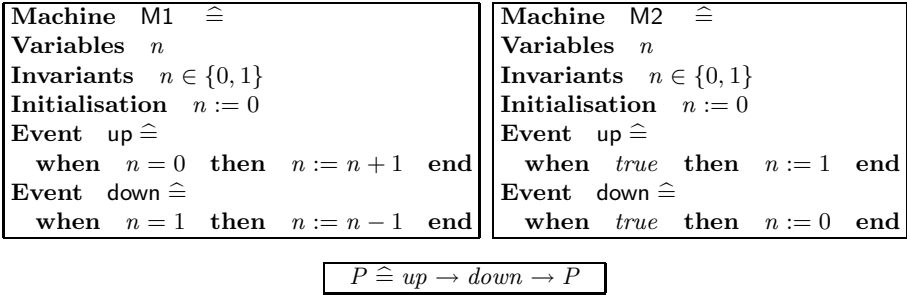


Fig. 1. Event-B machines and control

the other hand, they can be used as a straightforward way of modelling control-oriented requirements, and can thus ease specification. Figure 1 illustrates these two applications. The machine $M1$ on the left hand side regulates its control flow (alternation between **up** and **down** events) via guards on the events. This control flow is made explicit in the Communicating Process Algebra (CSP) [9] process P below. Alternatively, we could have used the machine $M2$ on the right hand side to describe the state, and then let the CSP process P in addition fix the ordering of events. In this simple case the flow of control in $M1$ is obvious and the variable n might be needed anyway. However, more complex applications might necessitate introducing variables solely acting as program counters. This compromises readability of the specification and ease of modelling.

In this paper we will propose a combination of Event-B with the process algebra CSP which serves these two purposes. The paper begins with a motivating example in order to illustrate how CSP descriptions can be defined alongside an Event-B machine. We will then give a failures divergences semantics (the semantics domain of CSP) to the integration via a weakest precondition semantics for Event-B. This follows previous approaches to integrating B with CSP [19] through relating weakest preconditions with CSP’s failures-divergence semantics [13]. The main focus of the paper rests on studying two fundamental issues arising for the combination: how can we determine whether the obtained specifications stay *deadlock free* despite the addition of CSP processes, and how can the central notion of *refinement* used for developing specifications be applied in the combination? The first issue is of particular importance because establishing deadlock-freedom in pure Event-B models is often difficult in practice when the flow of control is not simple, and so it is valuable to investigate approaches which can make that easier. In this paper we introduce proof obligations on Event-B machines which guarantee deadlock-freedom of a combination. For the second part we develop conditions which allow to prove refinement for a combined specification on the Event-B and CSP part in isolation. This gives rise to a compositional refinement framework. Both techniques are illustrated on a running example.

The paper is structured as follows: Section 2 introduces the notation and semantics for CSP and for Event-B; Section 3 introduces the Bridge example

used to illustrate the approach; Section 4 contains the main results for establishing deadlock-freedom; and Section 5 discusses refinement; finally, Section 6 concludes.

2 Notation

We start with a short introduction to the two formalisms, CSP and Event-B.

2.1 CSP

CSP [9] is a process algebra and used to specify control oriented applications. In this paper we will use the following subset of the CSP language:

$$P ::= e \rightarrow P \mid P_1 \square P_2 \mid P_1 \sqcap P_2 \mid P_1 \parallel P_2 \mid S$$

The event e here is drawn from the set of events in process P , and S is a CSP process variable. Events can either be pure CSP events, or correspond to events in the corresponding Event-B machine. Notationally we will use e for simple atomic CSP events not corresponding to Event-B events, whereas op will be used for Event-B event names. In this paper we assume that we have no parameters to channels. Recursive definitions are given as $S \hat{=} P$. In a CSP definition, all process variables used are bound by some recursive definition. External choice, \square , is a choice between alternatives which can be influenced by other components running in parallel, whereas \sqcap is an internal choice taken by the process alone. The prefix operator \rightarrow denotes sequencing. The CSP process P in Figure 1 thus specifies a recursive process alternating between *up* and *down* events.

The form of parallel combination we use is *alphabet* parallel, in which processes are associated with an *alphabet* (usually denoted $\alpha(P)$) which is a set of events. The occurrence of an event in the combination requires the participation of all processes whose alphabet contains that event.

The semantics of CSP (see e.g. [16]) is given in terms of its *traces* (the sequences of events it can execute), its *failures* (the events it might refuse after a trace) and its *divergences* (the traces after which it might engage in internal events only):

$$\begin{aligned} \text{traces}(P) &\subseteq \alpha(P)^* \\ \text{failures}(P) &\subseteq \alpha(P)^* \times 2^{\alpha(P)} \\ \text{divergences}(P) &\subseteq \alpha(P)^* \end{aligned}$$

The process $P \hat{=} up \rightarrow down \rightarrow P$ for instance has the alphabet $\{up, down\}$ and $\text{traces}(P) = \{\langle \rangle, \langle up \rangle, \langle up, down \rangle, \langle up, down, up \rangle, \dots\}$, $\text{failures}(P) = \{\langle \rangle, \langle down \rangle\}$, $\langle up \rangle, \langle up \rangle, \dots\}$ and $\text{divergences}(P) = \emptyset$. These three semantic domains come with three different notions of process refinement in CSP, two of which we are going to consider.

Definition 1. Let P_1, P_2 be CSP processes.

P_2 is a traces refinement of P_1 ($P_1 \sqsubseteq_T P_2$) if $\text{traces}(P_2) \subseteq \text{traces}(P_1)$.

P_2 is a failures refinement of P_1 ($P_1 \sqsubseteq_F P_2$) if $\text{failures}(P_2) \subseteq \text{failures}(P_1)$.

Intuitively, trace refinement is only concerned with *safety*: the refinement may not exhibit more execution traces than the abstract process. Failures on the other hand also treat *liveness*: a pair $(tr, X) \in failures(P)$ specifies events X which may be *refused* to be executed after some trace tr . Failures refinement guarantees that the concrete process may not refuse more events than the abstract one. Further explanation of refinement can be found in [16].

2.2 Event-B

Event-B [11][12] is a state-based specification formalism based on set theory. We cannot describe all of Event-B here, only the basic parts of an Event-B machine, required for this paper. A machine specification usually defines variables (collectively called v), constants c (possibly with axioms $A(c)$, which however do not occur in our examples) and a set of invariants $I(c, v)$ on constants and variables. The core part is the definitions of events, each consisting of a *guard* $G(c, v)$ over constants and variables and a *body* (usually written as an assignment on the variables) which defines a *before-after predicate* $E(c, v, v')$ describing changes of variables upon event execution, in terms of the relationship between the variable values before (v) and after (v'). A machine also has an initialisation T . Proof obligations on events are expressed in terms of *weakest precondition* semantics, where $[S]R$ denotes the weakest precondition for statement S to guarantee to establish postcondition R . A machine will have various proof obligations on it. These include consistency obligations, that events preserve the invariant. They can also include (optional) deadlock-freeness obligations, that at least one event guard is always true.

A machine M_1 is refined by another machine M_2 if there is a *linking invariant* (i.e. a predicate) J on the variables of the two machines, which is established by their initialisations, and which is preserved by all events, in the sense that any event of M_2 can be matched by an event of M_1 to maintain J . This is the standard notion of downwards simulation data refinement (see e.g. [5] for a description). New events can also be introduced as data refinements of *skip* [1]: they need not always be enabled, but their execution should maintain the linking relationship to the same abstract state. Event-B admits a variety of proof obligations depending on what is appropriate for the application. For the purposes of this paper (where we need refinement in Event-B to induce refinement in the CSP semantics), we require the *strong relative deadlock freeness* property S_DLK_E of [12], which states that whenever an event E is enabled in machine M_1 , then either E or a newly introduced event should be enabled in M_2 . We also require the *non-divergence* property WFD_REF , which states that newly introduced events cannot always be enabled. When the standard refinement conditions, together with both these conditions, are met we write $M_1 \sqsubseteq_D M_2$.

The machine M_1 in Figure 1 for instance defines one variable n , specifies this to only take values 0 and 1 (invariant) and defines the two events `up` and `down`. An initialisation section furthermore fixes the initial value for n .

Morgan's CSP semantics for action systems [13] allows traces, failures, and divergences to be defined for Event-B machines in terms of the sequences of

operations that they can and cannot engage in. This gives a way of considering Event-B machines as CSP processes, and treating them within the CSP framework. Note that the notion of *traces* for machines is dual to that presented in [1], where traces are considered as sequences of *states* rather than our treatment of traces as sequences of *events*.

The alphabet $\alpha(M)$ of a machine M is simply its set of *events*.

The traces of a machine M are those sequences of events $tr = \langle a_1, \dots, a_n \rangle$ which are possible for M (after initialisation T): those that do not establish *false*:

$$\text{traces}(M) = \{tr \mid \neg[T;tr]\text{false}\}$$

Here, the weakest precondition on a sequence of events is the weakest precondition of the sequential composition of those events: $[\langle a_1, \dots, a_n \rangle]P$ is given as $[a_1; \dots; a_n]P$.

The failures of a machine M are those pairs (tr, X) for which performing tr followed by refusing X is possible:

$$\text{failures}(M) = \{(tr, X) \mid \neg[T;tr](\bigvee_{op \in X} G_{op}(c, v))\}$$

In other words, it is not always the case that performance of tr is followed by some event from X being enabled.

A sequence of operations tr is a divergence if the sequence of operations is not guaranteed to terminate, i.e. $\neg[T;tr]\text{true}$. Thus

$$\text{divergences}(M) = \{tr \mid \neg[T;tr]\text{true}\}$$

M is divergence-free if $\text{divergences}(M) = \emptyset$.

These definitions provide the link between the weakest precondition semantics of the operations, and the CSP semantics of the B machine.

2.3 Combining CSP and Event-B

Figure 1 has defined the process P that alternates between two events. We do not require both the process P and the machine $M1$ in order to capture the requirement of alternating events. Either description independently would have been clear enough. Nonetheless, it is possible to combine the descriptions of P and $M1$ and we could view P as being an annotation of $M1$. This is exactly the way in which control flow expressions are being used in [10]. The author is using *flows* to provide patterns for the events of an Event-B machine, but does not permit the flows to contain events other than those in the Event-B machine. Being able to provide clear annotation of control flow is one benefit of including control flow expressions within Event-B machines. Another is to be able to relieve an Event-B machine of describing control flow explicitly and handing over that responsibility to a CSP process. Then the purpose of the Event-B machine is to define appropriate updates of the state of the machine.

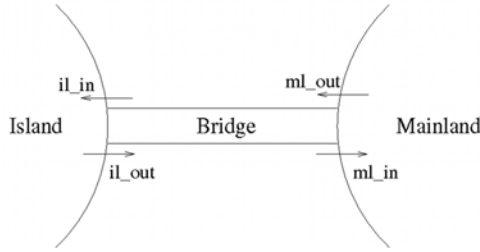


Fig. 2. Single lane bridge between mainland and island

Consider machine $M2$, also defined in Figure 1. The process P combined with $M2$ (i.e. $P \parallel M2$) also provides a definition which specifies alternating events. In this example we have handed over complete control to the CSP. We will illustrate in this paper a mixture of responsibilities in which both the CSP and the Event-B contribute to controlling the ordering of events within a system. It will enable us to see clearly how control flow is also restricted by the state of the system. To this end, we define a joint semantics for the integration in terms of the failures-divergence model of CSP. By giving a CSP semantics to an Event-B machine M , the CSP semantics of $P \parallel M$ follows from the CSP semantics of the parallel operator \parallel .

We will also explore in this paper how consistency conditions can be used to ensure deadlock-freedom of an integrated specification and how refinement can be proven in the integration.

3 Motivating and Running Example

We start with an example, inspired by Abrial’s car-island example of [1], to exemplify the usefulness of control flow specifications in Event-B machines. The specification is of a single lane bridge going from the mainland (ml) to an island (il). The bridge has a capacity of 10 cars (stored in a **constant** CAP). Unlike [1], our island has no limit on the number of cars on it. The specification needs to ensure that cars only travel in one direction on the bridge; variables a and c are used to track the number of cars on the bridge travelling from mainland to island and vice versa.

Figure 2 shows the bridge and four events which are part of the abstract specification: ml_out and ml_in are events moving cars out of and into the mainland, and il_in and il_out are the corresponding events for the island. The abstract Event-B machine $Bridge0$ given in Figure 3 needs to guarantee that cars on the bridge only travel in one direction and that the bridge does not become overloaded. The guards of the events ensure this, e.g., a car may only move from mainland to bridge (ml_out) if the direction island-to-mainland is currently empty, which we can see from the value of variable c , and if the capacity of the bridge is not already reached, which we can see from a . When it enters the bridge in direction island, variable a is incremented.

Machine	Bridge0	$\hat{=}$	
Variables	a, c		
Constants	$CAP = 10$		
Invariants	$a, c \in \mathbb{N}$		
Initialisation	$a := 0, c := 0$		
Event	m _l Out	$\hat{=}$	when $c = 0 \wedge a < CAP$ then $a := a + 1$ end
Event	m _l In	$\hat{=}$	when $c > 0$ then $c := c - 1$ end
Event	i _l Out	$\hat{=}$	when $a = 0 \wedge c < CAP$ then $c := c + 1$ end
Event	i _l In	$\hat{=}$	when $a > 0$ then $a := a - 1$ end

Fig. 3. Abstract bridge model

This constitutes our abstract specification. Here, there is no necessity of explicitly modelling control. The ordering of events depends on the data values of a and c only. This could also be modelled in CSP, but state is not CSP's primary domain.

3.1 Bridge with CSP Control

In a development step, the specification is refined so as to introduce traffic lights which regulate the flow of cars onto the bridge. There are two traffic lights here, one between mainland and bridge (m_l_tl) and the second one between island and bridge (i_l_tl). Each can be either green or red. The single lane use of the bridge should now be achieved by proper switching of traffic light colours. In this setting it becomes obvious that certain orderings among events need to be specified, and CSP provides a natural way of doing so. The first part concerns the behaviour of car drivers: if car drivers would ignore red traffic lights, then the correctness of the system can never be achieved, and so we capture the expectation that drivers will not drive through a red light. The environment assumption about correct driver behaviour is formulated in CSP as *REQ1* and *REQ2*:

$$\begin{array}{ll}
 REQ1 = m_tl_green \rightarrow P & REQ2 = i_tl_green \rightarrow Q \\
 P = m_out \rightarrow P & Q = i_out \rightarrow Q \\
 \square m_tl_red \rightarrow REQ1 & \square i_tl_red \rightarrow REQ2
 \end{array}$$

These two processes specify constraints on cars going past the two traffic lights: m_lOut is only possible when the mainland traffic light is green (*REQ1*) and a similar property needs to hold for i_lOut. *REQ1* specifies a process which first of all executes event m_l_tl_green and then has the choice of allowing m_lOut or carrying on with m_l_tl_red.

A second constraint contains the switching of traffic lights: at most one of them is allowed to be green, which gives a natural ordering on the events. The process *TL1* allows the choice of which light to switch at any stage. Thus we model the choice between turning the island or the mainland light to green.

Machine	<i>Bridge1</i>	$\hat{=}$
Variables	<i>a, c</i>	
Constants	<i>CAP = 10</i>	
Invariants	<i>a, c ∈ ℕ</i>	
Initialisation	<i>a := 0, c := 0</i>	
Event	<i>mL_out</i>	$\hat{=}$ when <i>a < CAP</i> then <i>a := a + 1</i> end
Event	<i>mL_in</i>	$\hat{=}$ when <i>c > 0</i> then <i>c := c - 1</i> end
Event	<i>iL_out</i>	$\hat{=}$ when <i>c < CAP</i> then <i>c := c + 1</i> end
Event	<i>iL_in</i>	$\hat{=}$ when <i>a > 0</i> then <i>a := a - 1</i> end
Event	<i>mL_tl_green</i>	$\hat{=}$ when <i>c = 0</i> then <i>skip</i> end
Event	<i>iL_tl_green</i>	$\hat{=}$ when <i>a = 0</i> then <i>skip</i> end

Fig. 4. The *Bridge1* machine

$$\begin{aligned}
 TL1 &= mL_tl_green \rightarrow mL_tl_red \rightarrow TL1 \\
 \square & iL_tl_green \rightarrow iL_tl_red \rightarrow TL1
 \end{aligned}$$

The data dependent part of the system still remains in an Event-B machine, *Bridge1*, given in Figure 4. Observe that the guards *c = 0* and *a = 0* have been dropped from events *mL_out* and *iL_out* respectively. Responsibility for ensuring this element of the condition that these events are enabled is now within the CSP part of the description, arising from the behaviour of the traffic lights, and the assumptions about correct driver behaviour. The combination of CSP and Event-B allows for a natural and clear separation of data-dependent and control-dependent aspects of a model. The resulting model is:

$$TL1 \parallel REQ1 \parallel REQ2 \parallel Bridge1$$

We will wish to show that this model is internally consistent: that the CSP control description is compatible with the Event-B model, and does not introduce new deadlocks. We will also want to be able to relate this model to the original abstract *Bridge0* model, to demonstrate that it is a refinement. The next sections provide the underlying theory to enable us to consider these issues.

3.2 Event-B Bridge with Control

Figure 5 gives a pure Event-B machine which has the same behaviour as the combined model $TL \parallel REQ1 \parallel REQ2 \parallel Bridge1$, where all the control is managed within the event guards. We introduce a variable for each of the CSP control components: *tl*, *r1*, and *r2*, for *TL1*, *REQ1*, and *REQ2* respectively. Their values correspond to the states of the processes: they are used as guards to enable events, and they are updated when events occur in accordance with the control process description. For example, *tl = reds* is part of the enabling condition for the *tl_green* events, and *tl* is updated according to which light turns green.

Machine	ControlledBridge	$\hat{=}$
Variables	$a, c, tl, r1, r2$	
Sets	$LIGHTS = \{reds, mlgreen, ilgreen\}$	
Constants	$CAP = 10$	
Invariants	$a, c \in \mathbb{N} \wedge r1, r2 \in \{0, 1\} \wedge tl \in LIGHTS$	
Initialisation	$a := 0, c := 0, r1 := 0, r2 := 0, tl := reds$	
Event	ml_out	$\hat{=}$ when $a < CAP \wedge r1 = 1$ then $a := a + 1$ end
Event	ml_in	$\hat{=}$ when $c > 0$ then $c := c - 1$ end
Event	il_out	$\hat{=}$ when $c < CAP \wedge r2 = 1$ then $c := c + 1$ end
Event	il_in	$\hat{=}$ when $a > 0$ then $a := a - 1$ end
Event	ml_tl_green	$\hat{=}$ when $c = 0 \wedge r1 = 0 \wedge tl = reds$ then $r1 := 1 \parallel tl := mlgreen$ end
Event	il_tl_green	$\hat{=}$ when $a = 0 \wedge r2 = 0 \wedge tl = reds$ then $r2 := 1 \parallel tl := ilgreen$ end
Event	ml_tl_red	$\hat{=}$ when $r1 = 1 \wedge tl = mlgreen$ then $r1 := 0 \parallel tl := reds$ end
Event	il_tl_red	$\hat{=}$ when $r2 = 1 \wedge tl = ilgreen$ then $r2 := 0 \parallel tl := reds$ end

Fig. 5. The *Bridge* machine with control incorporated within the guards

In contrast to the previous specification, we cannot directly see the flow of control on this machine anymore. There is no way of detecting that variables a and c are used for holding data values, whereas variables $r1$, $r2$ and tl are used for regulating the ordering of events. Furthermore, the switching of traffic lights and the requirement on the car drivers respecting traffic lights is now mixed together. The need for separation of different requirements on the model which we had in the bridge with CSP control is gone.

3.3 Abrial's Event-B Bridge

The standard Event-B approach taken by Abrial in the development of the bridge in [1] is to proceed by a series of refinement steps focusing on the state invariants, each of which introduces new events (such as the traffic lights), and where the control emerges gradually as proof obligations are discharged. For example, the requirement that at least one light must be red emerges from the requirement that all events should preserve the invariant that the bridge should not contain cars travelling in both directions.

The resulting bridge system is quite different to that in Figure 5, reflecting the fact that allowing implicit control to emerge naturally in an Event-B development is a different approach to imposing the flow of control explicitly, as we propose in this paper. However, it is recognised (anecdotally) [8] that establishing deadlock-freedom for such developments is often difficult in practice.

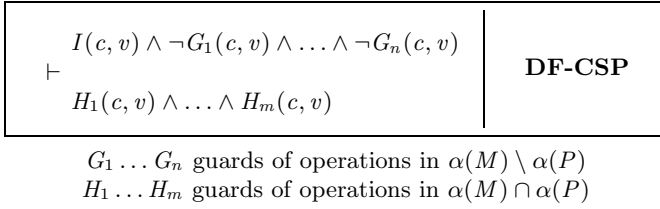


Fig. 6. Deadlock freedom for CSP control

4 Deadlock

Having formally defined the meaning of a combined CSP and Event-B specification now, we will next look at our two main issues: establishing deadlock-freedom in this section, and compositional refinement in the next section.

If a CSP control description P is introduced to a deadlock-free Event-B machine M , then there is a possibility that the additional constraints introduced by P might lead to a deadlock. This is possible since both CSP and Event-B define restrictions on the execution of events, and whenever these restrictions are not consistent for shared events the combined model may deadlock. In terms of the failures-divergence semantics this means that there is a trace after which all events are being refused.

Definition 2. *Let P be a CSP process and M an Event-B machine. The combination $P \parallel M$ is said to deadlock if there is a trace $tr \in (\alpha(P) \cup \alpha(M))^*$ such that $(tr, \alpha(P) \cup \alpha(M)) \in failures(P \parallel M)$.*

We will generally introduce a CSP controller over events which will be made available by the Event-B part of the description. Such events might always be enabled, but more generally we would only require them to be enabled at points where none of the other events (i.e. those not in the CSP, and therefore under the control of the Event-B) are enabled. This design principle gives rise to the proof rule **DF-CSP** given in Figure 6. This condition could for instance be established using the Rodin toolset [2]. In this rule, $G_1 \dots G_n$ are the guards of the operations in $\alpha(M) \setminus \alpha(P)$ and $H_1 \dots H_m$ are the guards of operations in $\alpha(M) \cap \alpha(P)$. The proof rule requires that whenever all of the events coming from the Event-B machine alone are disabled, then all events jointly controlled by the CSP process and the Event-B machine need to be enabled in the machine. Thus the machine cedes control at that point to the CSP controller.

Theorem 1. *Let P be a deadlock-free CSP process and M a divergence-free Event-B machine which satisfies DF-CSP. Then $P \parallel M$ is deadlock-free.*

This theorem considers the case where, whenever all of the events controlled by M alone are not enabled, then *all* of the events shared with P are enabled. In such a situation, deadlock-freedom of P yields deadlock-freedom of $P \parallel M$.

Observe that if $\alpha(M) \cap \alpha(P) \neq \emptyset$, then the condition on the operation guards of M implies that M is deadlock-free.

Theorem 1 is applicable to the *Bridge1* example of Section 3. In that example we have that

$$\begin{aligned}\alpha(\text{Bridge1}) \setminus \alpha(\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2}) &= \{ml_in, il_in\} \\ \alpha(\text{Bridge1}) \cap \alpha(\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2}) &= \{ml_out, il_out, \\ &\quad ml_tl_green, il_tl_green\}\end{aligned}$$

If all of the guards in $\alpha(\text{Bridge1}) \setminus \alpha(\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2})$ are false, then we have $c = 0 \wedge a = 0$. This implies each of the guards in $\alpha(\text{Bridge1}) \cap \alpha(\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2})$, and hence implies their conjunction. This is the condition to conclude deadlock-freedom of $\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2} \parallel \text{Bridge1}$.

The condition establishes that *all* of *Bridge1*'s events shared with $\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2}$ are enabled whenever all of the events that *Bridge1* controls independently are blocked. In such a state *Bridge1* does not constrain $\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2}$ at all, so $\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2}$'s deadlock-freedom extends to $\text{TL1} \parallel \text{REQ1} \parallel \text{REQ2} \parallel \text{Bridge1}$.

Theorem 1 is also applicable to the example $P \parallel M2$ of Figure 1. However, it is not applicable to $P \parallel M1$, since the two events of M are shared with P , but their guards are never both true, i.e. $\bigwedge_{op \in \alpha(M1) \cap \alpha(P)} G_{op}(c, v)$ does not hold.

A more general theorem for deadlock-freedom is available, as a specialisation of a result presented in [19] concerned with blocking B operations in classical B. It is applicable to sequential controllers, i.e. those made up of prefix, choice, and recursion, and focuses on the choices provided by the controller after any particular trace.

To state the theorem we need first to define for *sequential* CSP terms P :

- *offers*(P): the offers made by P at stages before a recursive call;
- *pass*(P): the traces corresponding to a single complete pass through a recursively defined process

Definition 3. For a CSP term P , the set *offers*(P) is defined inductively as follows:

$$\begin{aligned}\text{offers}(a \rightarrow P) &= \{(\langle \rangle, \{a\})\} \cup \{(\langle a \rangle \wedge tr, \text{Off}) \mid (tr, \text{Off}) \in \text{offers}(P)\} \\ \text{offers}(P1 \square P2) &= \{(\langle \rangle, \text{Off1} \cup \text{Off2}) \mid (\langle \rangle, \text{Off1}) \in \text{offers}(P1) \\ &\quad \wedge (\langle \rangle, \text{Off2}) \in \text{offers}(P2)\} \\ &\quad \cup \{(tr, \text{Off1}) \mid (tr, \text{Off1}) \in \text{offers}(P1) \wedge tr \neq \langle \rangle\} \\ &\quad \cup \{(tr, \text{Off2}) \mid (tr, \text{Off2}) \in \text{offers}(P2) \wedge tr \neq \langle \rangle\} \\ \text{offers}(P1 \sqcap P2) &= \text{offers}(P1) \cup \text{offers}(P2) \\ \text{offers}(S) &= \{\}\end{aligned}$$

Definition 4. For a CSP term P , the set *pass*(P) is defined inductively as follows:

$$\begin{aligned}
pass(a \rightarrow P) &= \{\langle a \rangle \wedge tr \mid tr \in pass(P)\} \\
pass(P1 \square P2) &= pass(P1) \cup pass(P2) \\
pass(P1 \sqcap P2) &= pass(P1) \cup pass(P2) \\
pass(S) &= \{\langle \rangle\}
\end{aligned}$$

Theorem 2. For a recursive definition $N \hat{=} P$, if $\alpha(P) = \alpha(M)$ and if there is a (control loop invariant) predicate *CLI* such that

- $[T]CLI$
- $\forall tr, Off.(tr, Off) \in offers(P) \Rightarrow (CLI \Rightarrow [tr](\bigvee_{op \in Off} G_{op}(c, v)))$
- $tr \in pass(P) \Rightarrow (CLI \Rightarrow [tr]CLI)$

then $P \parallel M$ is deadlock-free.

Consider P and $M1$ from Figure [□](#). We obtain

$$\begin{aligned}
offers(P) &= \{(\langle \rangle, \{up\}), (\langle up \rangle, \{down\})\} \\
pass(P) &= \{\langle up, down \rangle\}
\end{aligned}$$

We identify the control loop invariant *CLI* as $n = 0$, and check the conditions in turn:

- $[n := 0](n = 0)$ is indeed true.
- Checking the condition for the two offers in $offers(P)$: $n = 0 \Rightarrow [\langle \rangle]G_{up}$, and $n = 0 \Rightarrow [\langle up \rangle]G_{down}$ are both true.
- Checking the condition for the single pass: $n = 0 \Rightarrow [\langle up, down \rangle](n = 0)$ is also true.

The conditions are all true, so we conclude that $P \parallel M1$ is deadlock-free.

5 Refinement

In addition to introducing control to Event-B models, we are also interested in further *developing* an existing $CSP \parallel$ Event-B model. Both CSP and Event-B come with existing definitions of refinement [5](#) or development: in CSP this is process refinement and in Event-B data refinement. These guarantee the refinement to only have less traces or less failures than the abstract specification.

In contrast to process refinement, Event-B refinements usually also introduce new events. The corresponding notion of refinement in CSP would need to first *hide* (\backslash) these events in the concrete process and then check for trace or failures refinement. Hiding turns visible events into invisible, internal τ events. Thus for instance checking for trace refinement with new events A means checking $P \sqsubseteq_T Q \backslash A$.

The data refinement on machines discussed in Section [2.2](#) thus induces traces, failures, and divergences refinement. If $M_1 \sqsubseteq_D M_2$ in the Event-B setting, then $M_1 \sqsubseteq M_2 \backslash A$ in each of the CSP semantic models, where $A = \alpha(M_2) \setminus \alpha(M_1)$, the set of new events introduced in M_2 .

Our objective is to achieve a compositional framework for refinement (like for integrations of CSP and Object-Z [18,15]). In the case of trace refinement, the refinement relations are compositional. In other words, separately refining the components of a CSP||Event-B model results in a trace refinement of the model as a whole. Hence safety properties are preserved. This is expressed in Theorem 3.

Theorem 3. *Let P and P' be CSP processes such that $P \sqsubseteq_T P' \setminus A_1$, and M and M' Event-B machines such that $M \sqsubseteq_D M'$ with new events A_2 . Then the following holds:*

$$P \parallel M \sqsubseteq_T (P' \parallel M') \setminus (A_1 \cup A_2) .$$

Unfortunately a similar result for failures refinement does not hold, and it is not in general possible to deduce particular liveness behaviour of $P' \parallel M'$ from that of $P \parallel M$. This is because parallel composition does not in general preserve liveness properties. However, Theorem 1 is applicable directly to $P' \parallel M'$, thus still allowing deadlock-freedom results to be established directly for the refined models. Furthermore, we are able to obtain a less general result: if there is no intersection between the new events introduced into P and those introduced into M , then failures refinement is preserved.

Theorem 4. *Let P and P' be CSP processes such that $P \sqsubseteq_F P' \setminus A_1$, and M and M' Event-B machines such that $M \sqsubseteq_D M'$ with new events A_2 , where $A_1 \cap \alpha(M') = \emptyset$ and $A_2 \cap \alpha(P') = \emptyset$. Then the following holds:*

$$P \parallel M \sqsubseteq_F (P' \parallel M') \setminus (A_1 \cup A_2) .$$

Returning to our *Bridge* example, the bridge may occasionally need to be raised (to allow large ships through). This should occur only when there are no cars on the bridge in either direction, and also when the traffic lights are red in both directions. The lights should remain red until the bridge is lowered again.

This new feature is introduced in terms of new events in the Event-B model and the CSP description. The CSP description is augmented to capture the required relationship between the bridge lifting and the lights:

$$\begin{aligned} TL2 &= ml_tl_green \rightarrow ml_tl_red \rightarrow TL2 \\ &\square il_tl_green \rightarrow il_tl_red \rightarrow TL2 \\ &\square bridge_raise \rightarrow bridge_lower \rightarrow TL2 \end{aligned}$$

The requirement that no cars should be on the bridge when it is raised is captured naturally as a new Event-B machine *Bridge2* consisting of *Bridge1* augmented with the following event:

$$\boxed{\text{bridge_raise} \hat{=} \text{when } a = 0 \wedge c = 0 \text{ then skip end}}$$

Considering the control and the model separately, we have $TL1 \sqsubseteq_T TL2 \setminus A$, where $A = \{\text{bridge_raise}, \text{bridge_lower}\}$, and also $Bridge1 \sqsubseteq_D Bridge2$ with new events A .

Theorem 3 yields that $TL1 \parallel Bridge1 \sqsubseteq_T (TL2 \parallel Bridge2) \setminus A$, and hence that

$$TL1 \parallel REQ1 \parallel REQ2 \parallel Bridge1 \sqsubseteq_T (TL2 \parallel REQ1 \parallel REQ2 \parallel Bridge2) \setminus A$$

This demonstrates that the new feature is compatible with the existing system.

Furthermore, *Bridge2* meets **DF-CSP**, and $TL2 \parallel REQ1 \parallel REQ2$ is deadlock-free, so we can also conclude that $TL2 \parallel REQ1 \parallel REQ2 \parallel Bridge2$ is deadlock-free.

6 Conclusion

This paper has illustrated how CSP and Event-B descriptions can be combined and what reasoning can be performed on the combined models. The work resonates closely with [10] but is wider in scope because we want to consider using the process descriptions to specify requirements of a system which may not already be defined in the Event-B model. The benefit of splitting responsibility across both CSP and Event-B is that requirements can be dealt with separately. We must however investigate how global invariants can be expressed. In our example, one might say that if the most recent event is `ml_tl_green` then the number of cars coming the other way should be zero. i.e., $last(tr) = ml_tl_green \Rightarrow c = 0$. Since we are now combining descriptions, we lose the benefit of being able to express all invariants as state predicates. Further work is needed before we can conclude what can be expressed using a combination which would have been difficult using only Event-B predicates.

This paper is the basis of our ongoing research; we want to consider developing conditions which ensure that an introduction of a CSP process in a specification constitutes a valid refinement step, possibly using ideas of [6]. Mussat describes in [14] that we should be very clear about the separation between system variables and those that depict the physical environment, and it will be interesting to investigate whether CSP can contribute to the clear delineation of these aspects.

Acknowledgements

We are grateful to the anonymous reviewers for their thoughtful and constructive suggestions.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.-R., Butler, M.J., Hallerstede, S., Voisin, L.: A Roadmap for the Rodin Toolset. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, p. 347. Springer, Heidelberg (2008)

3. Butler, M.J.: csp2B: A practical approach to combining CSP and B. In: FACS, pp. 182–196 (2000)
4. Butler, M.J., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
5. Derrick, J., Boiten, E.A.: Refinement in Z and Object-Z. Springer, Heidelberg (2001)
6. Derrick, J., Wehrheim, H.: Model transformations incorporating multiple views. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 111–126. Springer, Heidelberg (2006)
7. Fischer, C.: CSP-OZ - a combination of CSP and Object-Z. In: Bowman, H., Derrick, J. (eds.) Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, pp. 423–438 (July 1997)
8. Hoang, T.S.: Personal Communication, Email (May 25, 2010)
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
10. Iliasov, A.: On Event-B and Control Flow. Technical report, School of Computing Science, Newcastle University (July 2009)
11. Mahony, B.P., Dong, J.S.: Blending Object-Z and timed CSP: An introduction to TCOZ. In: Futatsugi, K., Kemmerer, R., Torii, K. (eds.) 20th International Conference on Software Engineering (ICSE 1998). IEEE Press, Los Alamitos (1998)
12. Métayer, C., Abrial, J.-R., Voisin, L.: Event-B language. RODIN Project Deliverable 3.2, <http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf> (accessed 25/5/10)
13. Morgan, C.: Of wp and CSP. In: Beauty is Our Business: a Birthday Salute to E. W. Dijkstra, pp. 319–326 (1990)
14. Mussat, L.: Modèles Réactifs. Technical report, ClearSy (July 2008)
15. Olderog, E.-R., Wehrheim, H.: Specification and (property) inheritance in CSP-OZ. *Sci. Comput. Program.* 55(1-3), 227–257 (2005)
16. Schneider, S.: Concurrent and Real-time Systems: The CSP approach. Wiley, Chichester (1999)
17. Smith, G.: A semantic integration of Object-Z and CSP for the specification of concurrent systems. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 62–81. Springer, Heidelberg (1997)
18. Smith, G., Derrick, J.: Specification, Refinement and Verification of Concurrent Systems-An Integration of Object-Z and CSP. *Formal Methods in System Design* 18(3), 249–284 (2001)
19. Treharne, H., Schneider, S.: How to drive a B machine. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) B 2000, ZUM 2000, and ZB 2000. LNCS, vol. 1878, pp. 188–208. Springer, Heidelberg (2000)
20. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)

Towards Probabilistic Modelling in Event-B

Anton Tarasyuk^{1,2}, Elena Troubitsyna², and Linas Laibinis²

¹ Turku Centre for Computer Science

² Åbo Akademi University

Joukahaisenkatu 3-5 A, 20520 Turku, Finland

{anton.tarasyuk,elena.troubitsyna,linas.laibinis}@abo.fi

Abstract. Event-B provides us with a powerful framework for correct-by-construction system development. However, while developing dependable systems we should not only guarantee their functional correctness but also quantitatively assess their dependability attributes. In this paper we investigate how to conduct probabilistic assessment of reliability of control systems modeled in Event-B. We show how to transform an Event-B model into a Markov model amendable for probabilistic reliability analysis. Our approach enables integration of reasoning about correctness with quantitative analysis of reliability.

Keywords: Event-B, cyclic system, refinement, probability, reliability.

1 Introduction

System development by refinement is a formalised model-driven approach to developing complex systems. Refinement enables correct-by-construction development of systems. Its top-down development paradigm allows us to cope with system complexity via abstraction, gradual model transformation and proofs. Currently the use of refinement is mainly limited to reasoning about functional correctness. Meanwhile, in the area of dependable system development – the area where the formal modelling is mostly demanded – besides functional correctness it is equally important to demonstrate that the system adheres to certain quantitatively expressed dependability level. Hence, there is a clear need for enhancing formal modelling with a capability of stochastic reasoning about dependability.

In this paper we propose an approach to introducing probabilities into Event-B modelling [1]. Our aim is to enable quantitative assessment of dependability attributes, in particular, reliability of systems modelled in Event-B. We consider cyclic systems and show that their behaviour can be represented via a common Event-B modelling pattern. We show then how to augment such models with probabilities (using a proposed probabilistic choice operator) that in turn would allow us to assess their reliability.

Reliability is a probability of system to function correctly over a given period of time under a given set of operating conditions [23,24,17]. It is often assessed using the classical Markov modelling techniques [9]. We demonstrate that Event-B models augmented with probabilities can be given the semantic of a Markov

process (or, in special cases, a Markov chain). Then refinement of augmented Event-B models essentially becomes reliability-parameterised development, i.e., the development that not only guarantees functional correctness but also ensures that reliability of refined model is preserved or improved. The proposed approach allows us to smoothly integrate quantitative dependability assessment into the formal system development.

The paper is structured as follows. In Section 2 we overview our formal framework – Event-B. In Section 3 we introduce a general pattern for specifying cyclic systems. In Section 4 we demonstrate how to augment Event-B models with probabilities to enable formal modelling and refinement of fully probabilistic systems. In Section 5 we generalise our proposal to the cyclic systems that also contain non-determinism. Finally, in Section 6 we overview the related work and give concluding remarks.

2 Introduction to Event-B

The B Method [2] is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [19,5]. Event-B is a formal framework derived from the B Method to model parallel, distributed and reactive systems. The Rodin platform [21] provides automated tool support for modelling and verification (by theorem proving) in Event-B. Currently Event-B is used in the EU project Deploy [6] to model several industrial systems from automotive, railway, space and business domains.

In Event-B a system specification is defined using the notion of an abstract state machine [20]. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state. A general form of Event-B models is given in Fig. 1.

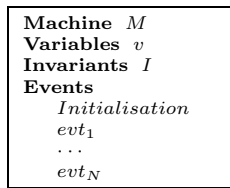


Fig. 1. An Event-B machine

The machine is uniquely identified by its name M . The state variables, v , are declared in the **Variables** clause and initialised in the *init* event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **Events** clause. Generally, an event can be defined as follows:

$$evt \hat{=} \mathbf{when } g \mathbf{ then } S \mathbf{ end,}$$

where the guard g is a conjunction of predicates over the state variables v and the action S is an assignment to the state variables. If the guard g is *true*, an event can be described simply as

$$evt \hat{=} \mathbf{begin} S \mathbf{end},$$

In its general form, an event can also have local variables as well as parameters. However, in this paper we use only the simple forms given above.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of assignments. The assignments can be either deterministic or non-deterministic. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A nondeterministic assignment is denoted either as $x \in S$, where S is a set of values, or $x \mid P(x, y, x')$, where P is a predicate relating initial values of x, y to some final value of x' . As a result of such a non-deterministic assignment, x can get any value belonging to S or according to P .

The semantics of Event-B events is defined using so called before-after (BA) predicates [20]. A before-after predicate describes a relationship between the system states before and after execution of an event, as shown in Fig. 2.

Action (S)	$BA(S)$
$x := E(x, y)$	$x' = E(x, y) \wedge y' = y$
$x \in S$	$\exists t. (t \in Set \wedge x' = t) \wedge y' = y$
$x \mid P(x, y, x')$	$\exists t. (P(x, t, y) \wedge x' = t) \wedge y' = y$

Fig. 2. Before-after predicates

where x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the after state. A before-after predicate for Event-B events is then constructed as follows:

$$BA(evt) = g \wedge BA(S).$$

The formal semantics provides us with a foundation for establishing correctness of Event-B specifications. In particular, to verify correctness of a specification, we need to prove that its initialisation and all events preserve the invariant.

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract specification. These new events correspond to stuttering steps that are not visible at the abstract level. By verifying correctness of refinement, we ensure that all invariant properties of (more) abstract machines are preserved. A detailed description of the formal semantics of Event-B and foundations of the verification process can be found in [20].

3 Modelling of Cyclic Systems in Event-B

In this paper, we focus on modelling systems with cyclic behaviour, i.e. the systems that iteratively execute a predefined sequence of steps. Typical representatives of such cyclic systems are control and monitoring systems. An iteration of a control system includes reading the sensors that monitor the controlled physical processes, processing the obtained sensor values and setting actuators according to a predefined control algorithm. In principle, the system could operate in this way indefinitely long. However, different failures may affect the normal system functioning and lead to a shutdown. Hence, during each iteration the system status should be re-evaluated to decide whether it can continue its operation.

In general, *operational* states of a system, i.e., the states where system functions properly, are defined by some predicate $J(v)$ over the system variables. Usually, essential properties of the system (such as safety, fault tolerance, liveness properties) can be guaranteed only while system stays in the operational states. The predicate $J(v)$ partitions the system state space S into two disjoint classes of states – *operational* (S_{op}) and *non-operational* (S_{nop}) states, where $S_{op} \hat{=} \{s \in S \mid J.s\}$ and $S_{nop} \hat{=} S \setminus S_{op}$.

Abstractly, we can specify a cyclic system in Event-B as shown in Fig. 3. In the machine CS , the variable st abstractly models the system state, which can be either operational ($J(st)$ is true) or failed ($J(st)$ is false). The event $iter$ abstractly models one iteration of the system execution. As a result of this event, the system can stay operational or fail. In the first case, the system can execute its next iteration. In the latter case, the system deadlocks.

```

Machine CS
Variables st
Invariants
  st ∈ STATE
...
Events
  Initialisation ≐
    begin
      st := J(st')
    end
  iter ≐
    when
      J(st)
    then
      st := STATE
    end

```

Fig. 3. A cyclic system

The **Invariants** clause (besides defining the variable types) can contain other essential properties of the system. Usually they are stated only over the operational states, i.e., they are of the form:

$$J(st) \Rightarrow \dots$$

We can refine the abstract specification CS by introducing specific implementation details. For example, we may explicitly introduce new events modelling

the environment as well as reading the sensors or setting the actuators. The event *iter* can be also refined, e.g., into *detection* operation, which decides whether the system can continue its normal operation or has to shut down due to some unrecoverable failure. However, the Event-B refinement process will preserve the cyclic nature of the system described in the abstract specification *CS*.

The only other constraint we put on the refinement process is that all the new events introduced in refined models can be only enabled in operational system states, e.g., the event guards should contain the condition $J(v)$. To enforce this constraint, we propose a simple syntactic extension of the Event-B model structure. Specifically, we introduce a new clause **Operational guards** containing state predicates precisely defining the subset of operational system states. This is a shorthand notation implicitly adding the corresponding guard conditions to all events enabled in the operational states (except initialisation). We also assume that, like model invariants, operational guards are inherited in all refined models. By using this new clause, we can rewrite the system *CS* as follows.

Machine <i>CS</i>
Variables <i>st</i>
Invariants
<i>st</i> ∈ <i>STATE</i>
...
Operational guards
$J(st)$
Events
<i>Initialisation</i> ≐
begin
<i>st</i> : $J(st')$
end
<i>iter</i> ≐
begin
<i>st</i> :∈ <i>STATE</i>
end

Fig. 4. A cyclic system

In general, the behaviour of some cyclic system *M* can be intuitively described by the sequential composition (*Initialisation*; **do** $J \rightarrow E$ **do**), where **do** $J \rightarrow E$ **do** is a while-loop with the operational guard *J* and the body *E* that consists of all the machine events except initialisation. For example, the behaviour of *CS* can be described simply as (*Initialisation*; **do** $J \rightarrow iter$ **do**).

Each iteration of the loop maps the current operational system state into a subset of *S*. The resulting set of states represents all possible states that can be reached due to system nondeterministic behaviour. Therefore, an iteration of a cyclic system *M* can be defined as a partial function \mathcal{I}_M of the type $S_{op} \rightarrow \mathcal{P}(S)$.¹ The concrete definition of \mathcal{I}_M can be derived from the composition of before-after predicates of the involved events. Moreover, we can also consider the behaviour of the overall system and observe that the final state of every iteration defines the initial state of the next iteration provided the system has not failed.

The specification pattern for modelling cyclic systems defined above restricts the shape of Event-B models. This restriction allow us to propose a scalable

¹ Equivalently, we can define an iteration as a relation between S_{op} and *S*.

approach to integrating probabilistic analysis of dependability into Event-B. This approach we present next.

4 Stochastic Modelling in Event-B

4.1 Introducing Probabilistic Choice

Hallerstede and Hoang [7] have extended the Event-B framework with a new operator – *qualitative probabilistic choice*, denoted \oplus . This operator assigns new values to variables with some positive but generally unknown probability. The extension aimed at introducing into Event-B the concept of “almost-certain convergence” – probabilistically certain termination of new event operations introduced by model refinement. The new operator can replace a nondeterministic choice (assignment) statement in the event actions. It has been shown that any probabilistic choice statement always refines its demonic nondeterministic counterpart [13]. Hence such an extension is not interfering with traditional refinement process.

In this paper we aim at introducing *quantitative* probabilistic choice, i.e., the operator \oplus with precise probabilistic information about how likely a particular choice should be made. In other words, it behaves according to some known probabilistic distribution. The quantitative probabilistic assignment

$$x \oplus x_1 @ p_1; \dots; x_n @ p_n,$$

where $\sum_{i=1}^n p_i = 1$, assigns to the variable x a new value x_i with the corresponding non-zero probability p_i . Similarly to Hallerstede and Hoang, we can introduce probabilistic choice only to replace the existing demonic one.

To illustrate the proposed extension, in Fig 5 we present a small example of a probabilistic communication protocol implementing transmission over unreliable channel. Since the channel is unreliable, sent messages may be lost. In the model AM shown on the left-hand side, the occurrence of faults is modelled nondeterministically. Specifically, the variable msg_a is nondeterministically assigned *delivered* or *lost*. In the model AM' , the nondeterministic choice is replaced by the probabilistic one, where the non-zero constant probabilities p and $1 - p$ express how likely a message is getting delivered or lost. According to the theory of probabilistic refinement [13], the machine AM' is a refinement of the machine AM . The model refinement relation is denoted \sqsubseteq .

Next we show how to define refinement between probabilistic systems modelled in (extended) Event-B. In particular, our notion of model refinement can be specialized to quantitatively demonstrate that the refined system is at least as reliable as its more abstract counterpart.

4.2 Fully Probabilistic Systems

Let us first consider fully probabilistic systems, i.e., systems containing only probabilistic nondeterminism. The quantitative information present in a

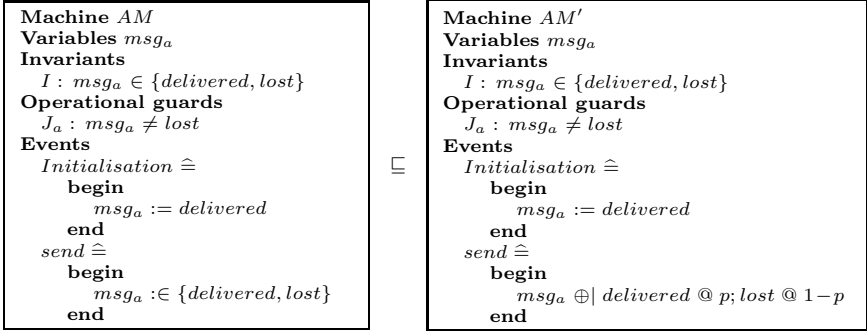


Fig. 5. A simple communication protocol: introducing probabilities

probabilistic Event-B model requires lifting the notion of the system state to a probabilistic distribution over it:

Definition 1 (Probabilistic distribution). For a system state space S , the set of distributions over S is

$$\bar{S} \hat{=} \{ \Delta : S \rightarrow [0, 1] \mid \sum_{s \in S} \Delta.s = 1 \},$$

where $\Delta.s$ is the probability of reaching the state s .

Each iteration of a fully probabilistic system then maps some initial operational state to a subset of S according to some probabilistic distribution, i.e., we can define a single iteration $\mathcal{P}\mathcal{I}_M$ of a probabilistic cyclic system M as a partial function of the type $S_{op} \rightarrow \bar{S}$.

There is a simple connection between iteration \mathcal{I}_M of a cyclic system M and and its probabilistic counterpart $\mathcal{P}\mathcal{I}_M$ – some state is reachable by \mathcal{I}_M if and only it is reachable by $\mathcal{P}\mathcal{I}_M$ with a non-zero probability:

$$\forall s, s'. s \in \mathbf{dom}.\mathcal{I}_M \wedge s' \in \mathcal{I}_M.s \Leftrightarrow s \in \mathbf{dom}.\mathcal{P}\mathcal{I}_M \wedge \mathcal{P}\mathcal{I}_M.s.s' > 0,$$

where \mathbf{dom} is the function domain operator.

For example, it is straightforward to see that for our model AM of the communication channel, the iteration function \mathcal{I}_{AM} is

$$\mathcal{I}_{AM} = \{delivered \mapsto \{delivered, lost\}\},$$

while the probabilistic iteration function $\mathcal{P}\mathcal{I}_{AM'}$ for the model AM' is

$$\mathcal{P}\mathcal{I}_{AM'} = \{delivered \mapsto \{delivered \mapsto p, lost \mapsto (1-p)\}\}.$$

As it was mentioned before, all elements of system state are partitioned into two disjoint classes of operational and non-operational states. For any state $s \in S_{op}$, its distribution Δ is defined by probabilistic choice statements (assignments)

presented in an Event-B machine. However, once the system fails, it stays in the failed (non-operational) state. This means that, for any state $s \in S_{nop}$, its distribution Δ is such that $\Delta.s = 1$ and $\Delta.s' = 0$, if $s' \neq s$.

Once we know the probabilistic state distribution Δ , we can quantitatively assess the probability that the operational guard J is preserved by a single iteration. However, our goal is to evaluate system reliability. In engineering, reliability [24,17] is generally measured by the probability that an entity \mathcal{E} can perform a required function under given conditions for the time interval $[0, t]$:

$$R(t) = \mathbf{P}\{\mathcal{E} \text{ not failed over time } [0, t]\}.$$

Hence reliability can be expressed as the probability that J remains *true* during a certain number of iterations, i.e., the probability of system staying operational for k iterations:

$$R(t) = \mathbf{P}\{\Box^{\leq k} J\}.$$

Here we use the modal operator \Box borrowed from temporal logic (LTL or (P)CTL, for instance). The formula $(\Box^{\leq k} J)$ means that J holds *globally* for the first k iterations. It is straightforward to see that this property corresponds to the standard definition of reliability given above.

Let M and M' be probabilistic Event-B models of cyclic systems. We strengthen the notion of Event-B refinement by additionally requiring that the refined model will execute more iterations before shutdown with a higher probability:

Definition 2 (Refinement for probabilistic cyclic systems)

For two probabilistic Event-B models M and M' of cyclic systems such that $M \hat{=} (\text{Initialisation}; \mathbf{do} J \rightarrow E \mathbf{do})$ and $M' \hat{=} (\text{Initialisation}'; \mathbf{do} J' \rightarrow E' \mathbf{do})$, we say that M' is a refinement of M , if and only if

1. M' is an Event-B refinement of M ($M \sqsubseteq M'$), and
2. $\forall k \in \mathbb{N}_1 \cdot \mathbf{P}\{\Box^{\leq k} J\} \leq \mathbf{P}\{\Box^{\leq k} J'\}$.

Remark 1. If the second condition of Definition 2 holds not for all k , but for some interval $k \in 1..K$, $K \in \mathbb{N}_1$, we say that M' is a *partial* refinement of M for $k \leq K$.

From the reliability point of view, a comparison of probabilistic distributions corresponds to a comparison of how likely the system would fail in its next iteration. This consideration allows us to define an order over the set \bar{S} of system distributions:

Definition 3 (Ordering over distributions). For two distributions $\Delta, \Delta' \in \bar{S}$ we define the ordering relation \preceq as follows

$$\Delta \preceq \Delta' \iff \sum_{s \in S_{op}} \Delta.s \leq \sum_{s \in S_{op}} \Delta'.s.$$

It is easy to see that the ordering relation \preceq defined in this way is reflexive and transitive and hence is a total preorder on S . Let us note that the defined order is not based on pointwise comparison between the corresponding single state probabilities. Instead, we rely on the accumulated likelihood that the system stays operational.

McIver and Morgan [13] have considered deterministic probabilistic programs with possible nontermination. They have defined the set of (sub-)distributions for terminating programs, with the order over distributions introduced as $\Delta \preceq \Delta' \Leftrightarrow (\forall s \in S \cdot \Delta.s \leq \Delta'.s)$. Such a pointwise definition of an order is too strong for our purposes. We focus on quantitative evaluation of system reliability, treating all the operational states in system distributions as one class, i.e., we do not distinguish between single operational states or their groups. In our future work it would be interesting to consider a more fine-grained classification of operational states, e.g., taking into account different classes of degraded states of the system.

The order over final state distributions can be in turn used to define the order over the associated initial states:

Definition 4 (Ordering over states). *Let M be a probabilistic cyclic system. Then, for its iteration $\mathcal{P}\mathcal{I}_M$, any initial states $s_i, s_j \in S_{op}$ and distributions $\Delta_i, \Delta_j \in \tilde{S}$ such that $\Delta_i = \mathcal{P}\mathcal{I}_M.s_i$ and $\Delta_j = \mathcal{P}\mathcal{I}_M.s_j$, we define the ordering relation \preceq_M as*

$$s_i \preceq_M s_j \Leftrightarrow \Delta_i \preceq \Delta_j$$

We can use this state ordering to represent the system state space S as an ordered set $\{s_1, \dots, s_n\}$, where $n \in \mathbb{N}_{\geq 2}$ and $(\forall i \in 1..(n-1) \cdot \Delta_{i+1} \preceq \Delta_i)$.

Generally, all the non-operational states S_{nop} can be treated as a singleton set, since we do not usually care at which particular state the operational guard has been violated. Therefore, by assuming that $S = \{s_1, \dots, s_n\}$ and $S_{nop} = \{s_n\}$, it can be easily shown that s_n is the least element (bottom) of S :

$$\Delta_n.s_n = 1 \Rightarrow \forall i \in 1..n \cdot s_n \preceq_M s_i$$

Now let us consider the behaviour of some cyclic system M in detail. We can assume that the initial system state s_1 belongs to the ordered set $\{s_1, \dots, s_n\}$. This is a state where the system works “perfectly”. After its first iteration, the system goes to some state s_i with the probability $\Delta_1.s_i$ and s_i becomes the current system state. At this point, if $i = n$, system shutdown is initiated. Otherwise, the system starts a new iteration and, as a result, goes to some state s_j with the probability $\Delta_i.s_j$ and so on. It is easy to see that this process is completely defined by the following state transition matrix

$$P_M = \begin{pmatrix} \Delta_1.s_1 & \Delta_1.s_2 & \dots & \Delta_1.s_n \\ \Delta_2.s_1 & \Delta_2.s_2 & \dots & \Delta_2.s_n \\ \vdots & \vdots & \ddots & \vdots \\ \Delta_n.s_1 & \Delta_n.s_2 & \dots & \Delta_n.s_n \end{pmatrix},$$

which in turn unambiguously defines the underlying Markov process (absorbing discrete time Markov chain, to be precise).

Let us note that the state transition matrix of a Markov chain and its initial state allow us to calculate the probability that the defined Markov process (after k steps) will be in a state s_i (see [9] for example). Let assume that the operational states of the system are ordered according to Definition [4] and initially a system is in the state s_1 . Then we can rewrite the second condition of Definition [2] in the following way:

Proposition 1. *For two probabilistic Event-B models M and M' such that $M \hat{=} (\text{Initialisation}; \mathbf{do} J \rightarrow E \mathbf{do})$ and $M' \hat{=} (\text{Initialisation}'; \mathbf{do} J' \rightarrow E' \mathbf{do})$, the inequality*

$$\forall k \in \mathbb{N}_1 \cdot \mathbf{P}\{\Box^{\leq k} J\} \leq \mathbf{P}\{\Box^{\leq k} J'\}$$

is equivalent to

$$\forall k \in \mathbb{N}_1 \cdot ((P_{M'})^k)_{1n'} \leq ((P_M)^k)_{1n}, \tag{1}$$

where $S = \{s_1, \dots, s_n\}$ and $S' = \{s_1, \dots, s_{n'}\}$ are the ordered system state spaces of M and M' accordingly, and $(\dots)_{1n}$ is a $(1n)$ -th element of a matrix.

Proof. Directly follows from our definition of the order on state distributions and fundamental theorems of the Markov chains theory. ■

In general, the initial system state is not necessarily the given state s_1 but can be defined by some initial state distribution Δ_0 . In this case the inequality ([1]) should be replaced with

$$([\Delta'_0] \cdot P_{M'}^k)(n') \leq ([\Delta_0] \cdot P_M^k)(n),$$

where $[\Delta_0] = \begin{pmatrix} \Delta_0.s_1 \\ \vdots \\ \Delta_0.s_n \end{pmatrix}$, $[\Delta'_0] = \begin{pmatrix} \Delta'_0.s_1 \\ \vdots \\ \Delta'_0.s_{n'} \end{pmatrix}$ and $([\Delta_0] \cdot P_M^k)(n)$ is the n -th component of the column vector $([\Delta_0] \cdot P_M^k)$.

To illustrate our approach to refining fully probabilistic systems, let us revisit our transmission protocol example. To increase reliability of transmission, we refine the protocol to allow the sender to repeat message sending in case of delivery failure. The maximal number of such attempts is given as the predefined positive constant N . The resulting Event-B model CM is presented in Fig [6]. Here the variable *att* represents the current sending attempt. Moreover, the event *send* is split to model the situations when the threshold N has been accordingly reached and not reached.

The Event-B machine CM can be proved to be a probabilistic refinement of its abstract probabilistic model (the machine AM' in Fig [5]) according to Definition [2].

In this section we focused on fully probabilistic systems. In the next section we generalize our approach to the systems that also contain nondeterminism.

```

Machine  $CM$ 
Variables  $msg_c, att$ 
Invariants
 $I_1 : msg_c \in \{delivered, try, lost\}$ 
 $I_2 : att \in 1..N$ 
Operational guards
 $J_c : msg_c \neq lost$ 
Events
  Initialisation  $\hat{=}$ 
    begin
       $msg_c := delivered$ 
       $att := 1$ 
    end
  start  $\hat{=}$ 
    when
       $msg_c = delivered$ 
    then
       $msg_c := try$ 
    end
  send1  $\hat{=}$ 
    when
       $msg_c = try \wedge att < N$ 
    then
       $msg_c, att \oplus | (delivered, 1) @ p; (try, att+1) @ 1-p$ 
    end
  send2  $\hat{=}$ 
    when
       $msg_c = try \wedge att = N$ 
    then
       $msg_c, att \oplus | (delivered, 1) @ p; (lost, att) @ 1-p$ 
    end

```

Fig. 6. A simple communication protocol: probabilistic refinement

4.3 Probabilistic Systems with Nondeterminism

For a cyclic system M containing both probabilistic and demonic nondeterminism we define a single iteration as the partial function $\mathcal{P}\mathcal{I}_M$ of the type $S_{op} \rightarrow \mathcal{P}(\bar{S})$, i.e., as a mapping of the operational state into a set of distributions over S .

Nondeterminism has a demonic nature in Event-B. Hence such a model represent a worst case scenario, i.e., choosing the “worst” of operative sub-distributions – the distributions with a domain restriction on S_{op} . From reliability perspective, it means that while assessing reliability of such a system we obtain the lowest bound estimate of reliability. In this case the notions of probabilistic system refinement and the state ordering are defined as follows:

Definition 5 (Refinement for nondeterministic-probabilistic systems).
 For two nondeterministic-probabilistic Event-B models M and M' of cyclic systems such that $M \hat{=} (Initialisation; \mathbf{do} J \rightarrow E \mathbf{do})$ and $M' \hat{=} (Initialisation'; \mathbf{do} J' \rightarrow E' \mathbf{do})$, we say that M' is a refinement of M , if and only if

1. M' is an Event-B refinement of M ($M \sqsubseteq M'$);
2. $\forall k \in \mathbb{N}_1 \cdot \mathbf{P}_{min}\{\Box^{\leq k} J\} \leq \mathbf{P}_{min}\{\Box^{\leq k} J'\}$,

where $\mathbf{P}_{min}\{\Box^{\leq k} J\}$ is the minimum probability that J remains true during the first k iterations.

Remark 2. If the second refinement condition of the Definition 5 holds not for all k , but for some interval $k \in 1..K$, $K \in \mathbb{N}_1$, we say that M' is a *partial* refinement of M for $k \leq K$.

Definition 6 (Ordering over distributions)

For two sets of distributions $\{\Delta_{i_l} \mid l \in 1..L\}$ and $\{\Delta_{j_k} \mid k \in 1..K\} \in \mathcal{P}(\bar{S})$, we define the ordering relation \preceq as

$$\{\Delta_{i_l} \mid l \in 1..L\} \preceq \{\Delta_{j_k} \mid k \in 1..K\} \Leftrightarrow \min_l \left(\sum_{s \in S} \Delta_{i_l} \cdot s \right) \leq \min_k \left(\sum_{s \in S} \Delta_{j_k} \cdot s \right).$$

As in the previous section, the order over final state distributions can be in turn used to define the order over the associated initial states:

Definition 7 (Ordering over states).

Let M be a nondeterministic-probabilistic system. Then, for its iteration $\mathcal{P}\mathcal{I}_M$, any initial states $s_i, s_j \in S_{op}$ and sets of distributions $\{\Delta_{i_l} \mid l \in 1..L\}, \{\Delta_{j_k} \mid k \in 1..K\} \in \mathcal{P}(\bar{S})$ such that $\{\Delta_{i_l} \mid l \in 1..L\} = \mathcal{P}\mathcal{I}_M \cdot s_i$ and $\{\Delta_{j_k} \mid k \in 1..K\} = \mathcal{P}\mathcal{I}_M \cdot s_j$, we define the ordering relation \preceq_M as

$$s_i \preceq_M s_j \Leftrightarrow \{\Delta_{i_l} \mid l \in 1..L\} \preceq \{\Delta_{j_k} \mid k \in 1..K\}$$

The underlying Markov process representing the behaviour of a nondeterministic-probabilistic cyclic system is a simple form of a Markov decision process. For every $i \in 1, \dots, (n - 1)$, let us define $\underline{\Delta}_i = \min_l \left(\sum_{s \in S} \Delta_{i_l} \cdot s \right)$ and $\underline{\Delta}_n = \Delta_n$. Then, the state transition matrix that represents the worst-case scenario system behaviour is defined in the following way:

$$\underline{P}_M = \begin{pmatrix} \underline{\Delta}_1 \cdot s_1 & \underline{\Delta}_1 \cdot s_2 & \dots & \underline{\Delta}_1 \cdot s_n \\ \underline{\Delta}_2 \cdot s_1 & \underline{\Delta}_2 \cdot s_2 & \dots & \underline{\Delta}_2 \cdot s_n \\ \vdots & \vdots & \ddots & \vdots \\ \underline{\Delta}_n \cdot s_1 & \underline{\Delta}_n \cdot s_2 & \dots & \underline{\Delta}_n \cdot s_n \end{pmatrix},$$

and the second refinement condition of Definition 5 can be rewritten as follows:

Proposition 2. For two nondeterministic-probabilistic Event-B models M and M' such that $M \hat{=} (\text{Initialisation}; \text{do } J \rightarrow E \text{ do})$ and $M' \hat{=} (\text{Initialisation}'; \text{do } J' \rightarrow E' \text{ do})$, the inequality

$$\forall k \in \mathbb{N} \cdot \mathbf{P}\{\square^{\leq k} J\} \leq \mathbf{P}\{\square^{\leq k} J'\}$$

is equivalent to

$$\forall k \in \mathbb{N}_1 \cdot ((\underline{P}_{M'})^k)_{1n'} \leq ((\underline{P}_M)^k)_{1n}, \tag{2}$$

where $S = \{s_1, \dots, s_n\}$ and $S' = \{s_1, \dots, s_{n'}\}$ are the ordered system state spaces of M and M' accordingly.

Proof. This proof is the same as the proof for Proposition 1. ■

Similarly as for fully-probabilistic systems, if the initial system state is not a single state s_1 , but instead it is defined by some initial state distribution Δ_0 , then the inequality (2) is replaced by

$$([\Delta'_0] \cdot \underline{P}_{M'}^k)(n') \leq ([\Delta_0] \cdot \underline{P}_M^k)(n).$$

4.4 Discussion

For fully probabilistic systems, we can often reduce the state space size using the lumping technique [9] or equally probabilistic bisimulation [12]. For nondeterministic probabilistic systems, a number of bisimulation techniques [8,22] have been also developed.

For simple system models, deriving the set of state distributions \bar{S} and calculating reliability probabilities P_M^k for each refinement step can be done manually. However, for complex real-size systems this process can be extremely time and effort consuming. Therefore, it is beneficial to have an automatic tool support for routine calculations. Development and verification of Event-B models is supported by the Rodin Platform [19] – integrated extensible development environment for Event-B. However, at the moment the support for quantitative verification is sorely missing. To prove probabilistic refinement of Event-B models according to Definition 2 and Definition 5, we need to extend the Rodin platform with a dedicated plug-in or integrate some external tool.

One of the available automated techniques widely used for analysing systems that exhibit probabilistic behaviour is probabilistic model checking [4,10]. In particular, the probabilistic model checking frameworks like PRISM or MRMC [18,16] provide good tool support for formal modelling and verification of discrete- and continuous-time Markov processes. To enable the quantitative reliability analysis of Event-B models, it would be advantageous to develop a Rodin plug-in enabling automatic translation of Event-B models to existing probabilistic model checking frameworks.

5 Related Work and Conclusions

5.1 Related Work

The Event-B framework has been extended by Hallerstede and Hoang [7] to take into account model probabilistic behaviour. They introduce qualitative probabilistic choice operator to reason about almost certain termination. This operator attempts to bound demonic nondeterminism that, for instance, allows us to demonstrate convergence of certain protocols. However, this approach is not suitable for reliability assessment since explicit quantitative representation of reliability is not supported.

Several researches have already used quantitative model checking for dependability evaluation. For instance, Kwiatkowska et al. [11] have proposed an

approach to assessing dependability of control systems using continuous time Markov chains. The general idea is similar to ours – to formulate reliability as a system property to be verified. This approach differs from ours because it aims at assessing reliability of already developed systems. However, dependability evaluation late at the development cycle can be perilous and, in case of poor results, may lead to major system redevelopment causing significant financial and time losses. In our approach reliability assessment proceeds hand-in-hand with the system development by refinement. It allows us to assess dependability of designed system on the early stages of development, for instance, every time when we need to estimate impact of unreliable component on the system reliability level. This allows a developer to make an informed decision about how to guarantee a desired system reliability.

A similar topic in the context of refinement calculus has been explored by Morgan et al. [14,13]. In this approach the probabilistic refinement has been used to assess system dependability. Such an approach is much stronger than the approach described in this paper. Probabilistic refinement allows the developers to obtain algebraic solutions even without pruning the system state space. Meanwhile, probabilistic verification gives us only numeric solutions for restricted system models. In a certain sense, our approach can be seen as a property-wise refinement evaluation. Indeed, while evaluating dependability, we essentially check that, for the same samples of system parameters, the probability of system to hold a certain property is not decreased by refinement.

5.2 Conclusions

In this paper we proposed an approach to integrating probabilistic assessment of reliability into Event-B modelling. We defined reliability of a cyclic system as the probability of the system to stay in its operational state for a given number of iterations. Our approach to augmenting Event B models with probabilities allows us to give the semantic of a Markov process (or, in special cases, a Markov chain) to augmented models. In turn, this allow us to algebraically compute reliability by using any of numerous automated tools for reliability estimation.

In general, continuous-time Markov processes are more often used for dependability evaluation. However, the theory of refinement of systems with continuous behaviour has not reached maturity yet [3,15]. In this paper we showed that, by restricting the shape of Event-B models and augmenting them with probabilities, we can make a smooth transition to representing a cyclic system as a Markov process. This allow us to rely on standard techniques for assessing reliability.

In our future work it would be interesting to explore continuous-time reasoning as well as generalise the notion of refinement to take into account several dependability attributes.

Acknowledgments

This work is supported by IST FP7 DEPLOY Project. We also wish to thank the anonymous reviewers for their helpful comments.

References

1. Abrial, J.R.: Extending B without Changing it (for Developing Distributed Systems). In: Habiras, H. (ed.) First Conference on the B method, pp. 169–190. IRIN Institut de recherche en informatique de Nantes (1996)
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. Back, R.J.R., Petre, L., Porres, I.: Generalizing Action Systems to Hybrid Systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 202–213. Springer, Heidelberg (2000)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Craigen, D., Gerhart, S., Ralson, T.: Case study: Paris metro signaling system. IEEE Software, 32–35 (1994)
6. EU-project DEPLOY, <http://www.deploy-project.eu/>
7. Hallerstede, S., Hoang, T.S.: Qualitative probabilistic modelling in Event-B. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 293–312. Springer, Heidelberg (2007)
8. Hansson, H.: Time and Probability in Formal Design of Distributed Systems. Elsevier, Amsterdam (1995)
9. Kemeny, J.G., Snell, J.L.: Finite Markov Chains. D. Van Nostrand Company (1960)
10. Kwiatkowska, M.: Quantitative verification: models techniques and tools. In: ESEC/FSE 2007, pp. 449–458. ACM, New York (2007)
11. Kwiatkowska, M., Norman, G., Parker, D.: Controller dependability analysis by probabilistic model checking. In: Control Engineering Practice, pp. 1427–1434 (2007)
12. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94, 1–28 (1991)
13. McIver, A.K., Morgan, C.C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, Heidelberg (2005)
14. McIver, A.K., Morgan, C.C., Troubitsyna, E.: The probabilistic steam boiler: a case study in probabilistic data refinement. In: Proc. International Refinement Workshop, ANU, Canberra. Springer, Heidelberg (1998)
15. Meinicke, L., Smith, G.: A Stepwise Development Process for Reasoning about the Reliability of Real-Time Systems. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 439–458. Springer, Heidelberg (2007)
16. MRMC – Markov Reward Model Checker, <http://www.mrmc-tool.org/>
17. O’Connor, P.D.T.: Practical Reliability Engineering, 3rd edn. John Wiley & Sons, Chichester (1995)
18. PRISM – Probabilistic Symbolic Model Checker, <http://www.prismmodelchecker.org/>
19. Rigorous Open Development Environment for Complex Systems (RODIN): IST FP6 STREP project, <http://rodin.cs.ncl.ac.uk/>
20. Rigorous Open Development Environment for Complex Systems (RODIN): Deliverable D7, Event-B Language, <http://rodin.cs.ncl.ac.uk/>
21. RODIN. Event-B Platform, <http://www.event-b.org/>
22. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2), 250–273 (1995)
23. Storey, N.: Safety-Critical Computer Systems. Addison-Wesley, Reading (1996)
24. Villemeur, A.: Reliability, Availability, Maintainability and Safety Assessment. John Wiley & Sons, Chichester (1995)

Safe Commits for Transactional Featherweight Java^{*}

Thi Mai Thuong Tran and Martin Steffen

Department of Informatics, University of Oslo, Norway

Abstract. Transactions are a high-level alternative for low-level concurrency-control mechanisms such as locks, semaphores, monitors. A recent proposal for integrating transactional features into programming languages is *Transactional Featherweight Java* (TFJ), extending Featherweight Java by adding transactions. With support for *nested* and *multi-threaded* transactions, its transactional model is rather expressive. In particular, the constructs governing transactions—to start and to commit a transaction—can be used freely with a *non-lexical* scope. On the downside, this flexibility also allows for an incorrect use of these constructs, e.g., trying to perform a commit outside any transaction. To catch those kinds of errors, we introduce a static type and effect system for the safe use of transactions for TFJ. We prove the soundness of our type system by subject reduction.

1 Introduction

With CPU speeds and memory capacities ever increasing, and especially with the advent of multiprocessor and multi-core architectures, effective parallel programming models and suitable language support are in need to take full advantage of the architectural advances. Transactions, a well-known and successful concept originating from database systems, have recently been proposed to be directly integrated into *programming languages*. As known from databases, transactions offer valuable safety and failure guarantees: atomicity, consistency, isolation, and durability, or ACID for short. Atomicity means that the code inside a transaction is executed completely or not at all, consistency that all transactions have the same “view” on shared data, isolation says that when a transaction is running, other transactions cannot interfere, and durability states successfully committed changes are persistent. One characteristic difference of transactions compared to locks is a non-blocking behavior. All threads/transactions may run in parallel provided that they guarantee the mentioned ACID properties. As a result, transactional programming languages may make better use of parallelism and resources in concurrent systems, and may avoid also deadlock situations.

As mechanism for concurrency control, they can be seen as a high-level, more abstract, and more compositional alternative to more conventional means for concurrency control, such as locks, semaphores, monitors, etc. How to syntactically capture transactional programming in the language may vary. One option is lexical scoping, e.g., using an *atomic* keyword, similar to the *synchronized* keyword in Java for lock-handling. More flexible is non-lexical scoping, where transactions can be started and finished (i.e.,

^{*} The work has been partly supported by the EU-project FP7-231620 [HATS](#) (Highly Adaptable and Trustworthy Software using Formal Methods).

committed) freely. One proposal supporting non-lexical scoping of transaction handling is *Transactional Featherweight Java* (TFJ) [15]. In the free use of the transactional constructs, it resembles also the way Java 5.0 allows for lock handling (using the lock and unlock methods via the Lock-interface). The start of a transaction in TFJ programs is marked by the `onacid` keyword and the end by the `commit` keyword. The transactional model of TFJ is quite general and supports *nested* transactions which means a transaction can contain one or more child transactions, which is very useful for composability and partial rollback. Furthermore, TFJ supports *multi-threaded* transactions, i.e., one transaction can contain internal concurrency. To commit an entire transaction, all child transaction must have committed and the child threads and the thread itself must commit at the same time. The flexibility of non-lexical use of `onacid` and `commit` comes at a cost: not all usages of starting and committing transactions “make sense”. In particular, it is an error to perform a commit without being inside a transaction; we call such an error a *commit error*. In this paper, we introduce a static type and effect system to prevent these errors by keeping track of starting and committing transactions. The static analysis is formulated as a type and effect system [18]. We concentrate on the effect part, as the part dealing with the ordinary types works in a standard manner and is straightforward. See [20] for details.

The paper is organized as follows. After Section 2 which recapitulates the syntax and the operational semantics of the calculus, Section 3 defines the effect system to prevent *commit errors*. The soundness of the type system relative to the given semantics is shown in Section 4. Section 5 concludes with related and future work. In particular we draw some parallel to the lock handling in Java 5.

2 An Object-Oriented Calculus with Transactions

Next we present the syntax and semantics of TFJ. It is, with some adaptations, taken from [15] and a variant of Featherweight Java (FJ) [13] extended with *transactions* and a construct for thread creation. The main adaptations are: we added standard constructs such as sequential composition (in the form of the `let`-construct) and conditionals. Besides that, we did not use evaluation-context based rules for the operational semantics.

2.1 Syntax

FJ is a core language originally introduced to study typing issues related to Java, such as inheritance, subtype polymorphism, type casts. A number of extensions have been developed for other language features, so FJ is today a generic name for Java-related core calculi. Following [15] we include imperative features such as destructive field updates, further concurrency and support for transactions. Table 1 shows the abstract syntax of TFJ. A program consists of a number of processes/threads $t\langle e \rangle$ running in parallel, where t is the thread’s identifier and e is the expression being executed.. The syntactic category L captures class definitions. In absence of inheritance, a class $\text{class } C\{\vec{f} : \vec{T}; K; \vec{M}\}$ consists of a name C , a list of fields \vec{f} with corresponding type declarations \vec{T} (assuming that all f_i ’s are different), a constructor K , and a list \vec{M} of method definitions. A constructor $C(\vec{f}:\vec{T})\{\text{this}.\vec{f} := \vec{f}\}$ of the corresponding class C initializes the fields of

Table 1. Abstract syntax

$P ::= \mathbf{0} \mid P \parallel P \mid t \langle e \rangle$	processes/threads
$L ::= \text{class } C \{ \vec{f} : \vec{T}; K; \vec{M} \}$	class definitions
$K ::= C(\vec{f} : \vec{T}) \{ \text{this}.\vec{f} := \vec{f} \}$	constructors
$M ::= m(\vec{x} : \vec{T}) \{ e \} : T$	methods
$e ::= v \mid v.f \mid v.f := v \mid \text{if } v \text{ then } e \text{ else } e \mid \text{let } x : T = e \text{ in } e \mid v.m(\vec{v})$	expressions
$\mid \text{new } C(\vec{v}) \mid \text{spawn } e \mid \text{onacid} \mid \text{commit}$	
$v ::= r \mid x \mid \text{null}$	values

instances of that class, these fields are mentioned as the formal parameters of the constructor. We assume that each class has exactly one constructor; i.e., we do not allow constructor overloading. Similarly, we do not allow method overloading by assuming that all methods defined in a class have a different name; likewise for fields. A method definition $m(\vec{x} : \vec{T}) \{ e \} : T$ consists of the name m of the method, the formal parameters \vec{x} with their types \vec{T} , the method body e , and finally the return type T of the method.

In the syntax, v stands for values, i.e., expressions that can no longer be evaluated. In the core calculus, we leave unspecified standard values like booleans, integers, \dots , so values can be object references r , variables x or null. The expressions $v.f$ and $v_1.f := v_2$ represent field access and field update respectively. Method calls are written $v.m(\vec{v})$ and object instantiation is $\text{new } C(\vec{v})$. The next two expressions deal with the basic, sequential control structures: $\text{if } v \text{ then } e_1 \text{ else } e_2$ represents conditions, and the $\text{let } x : T = e_1 \text{ in } e_2$ represents sequential composition: first e_1 is evaluated, and afterwards e_2 , where the eventual value of e_1 is bound to the local variable x . Consequently, standard sequential composition $e_1; e_2$ is syntactic sugar for $\text{let } x : T = e_1 \text{ in } e_2$ where the variable x does not occur free in e_2 . The language is multi-threaded: $\text{spawn } e$ starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two constructs onacid and commit , two dual operations dealing with transactions. The expression onacid starts a new transaction and executing commit successfully terminates a transaction. The syntax is restricted concerning where to use general expressions e . E.g., Table 1 does not allow field updates $e_1.f := e_2$, where the object whose field is being updated and the value used in the right-hand side are represented by general expressions. It would be straightforward to relax the abstract syntax that way. We have chosen this presentation, as it slightly simplifies the operational semantics and the (presentation of the) type and effect system later. Of course, this is not a real restriction in expressivity.

2.2 Semantics

This section describes the operational semantics of TFJ with some adaptations at two different levels: a local and a global semantics. The local semantics is given in Table 2. These local rules deal with the evaluation of *one* single *expression/thread* and reduce configurations of the form $E \vdash e$. Thus, local transitions are of the form $E \vdash e \rightarrow E' \vdash e'$, where e is one expression and E a *local environment*. At the local level, the relevant commands only concern the current thread.

Definition 1. A local environment E of type $LEnv$ is a finite sequence of the form $l_1:\rho_1, \dots, l_k:\rho_k$, i.e., of pairs of transaction labels l_i and a corresponding log ρ_i . We write $|E|$ for the size of the local environment (number of pairs $l:\rho$ in the local environment).

Transactions are identified by labels l , and as transactions can be nested, a thread can execute “inside” a number of transactions. So, the E in the above definition is ordered, with e.g. l_k to the right refers to the inner-most transaction, i.e., the one most recently started and committing removes bindings from right to left. The number $|E|$ of a thread represents the nesting depth of the thread, i.e., how many transactions the thread has started but not yet committed. The corresponding logs ρ_i can, in a first approximation, be thought of as “local copies” of the heap including bindings from references to objects. The log ρ_i keeps track of changes of the threads actions concerning transaction l_i . The exact structure of such environments and the logs have no influence on our static analysis, and indeed, the environments may be realized in different ways (e.g., [15] gives two different flavors, a “pessimistic”, lock-based one and an “optimistic” one). Relevant for our type and effect system will be only a number of *abstract properties* of the environments, formulated in Definition 3 later. As the local rules in Table 2 are pretty standard, and correspond to the ones of [15]. The first four rules deal straightforwardly with the basic, sequential control flow. Unlike the first four rules, the remaining ones do access the heap. Thus, the local environment E is consulted to look up object references and then *changed* in the step. The access and update of E is given abstractly by corresponding access functions *read*, *write*, and *extend* (which look-up a reference on the heap, update a reference, resp. allocate an entry for a new reference on the heap). The details can be found in [15] but note that also the *read*-function used in the rules actually *changes* the environment from E to E' in the step. The reason is that in a transaction-based implementation, read-access to a variable may be *logged*, i.e., remembered appropriately, to be able to detect conflicts and to do a roll-back if the transaction fails. This logging may change the local environment. The premises assume the class table is given implicitly where *fields*(C) looks up fields of class C and *mbody*(m, C) looks up the method m of class C . Otherwise, the rules for field lookup, field update, method calls, and object instantiation are standard.

The five rules of the *global* semantics are given in Table 3. The semantics works on configurations of the form $\Gamma \vdash P$, where P is a *program* and Γ is a global environment. Besides that, we need a special configuration *error* representing an error state. Basically, a program P consists of a number of threads evaluated in parallel (cf. Table 1), where each thread corresponds to one expression, whose evaluation is described by the local rules. Now that we describe the behavior of a number of (labeled) threads $t\langle e \rangle$, we need one E for each thread t . This means, Γ is a “sequence” (or rather a set) of $t:E$ bindings where t is the name of a thread and E is its corresponding local environment.

Definition 2. A global environment Γ of type $GEnv$ is a finite mapping, written as $t_1:E_1, \dots, t_k:E_k$, from threads names t_i to local environments E_i (the order of bindings does not play a role, and each thread name can occur at most once).

So global steps are of the form $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$ or $\Gamma \vdash P \Longrightarrow \text{error}$. As for the local rules, the formulation of the global steps makes use of a number of functions accessing and changing the (this time global) environment. As before, those functions are left

abstract and only later we will formalize abstract properties that Γ and E considered as abstract data types must satisfy. Rule G-PLAIN simply *lifts* a local step to the global level, using the reflect-operation, which roughly makes local updates of a thread globally visible. Rule G-SPAWN deals with starting a thread. The next three rules treat the two central commands of the calculus, those dealing directly with the transactions. The first one G-TRANS covers onacid, which starts a transaction. The *start* function creates a new label l in the local environment E of thread t . The two rules G-COMM and G-COMM-ERROR formalize the successful commit resp. the failed attempt to commit a transaction. In G-COMM, the label of the transaction l to be committed is found (right-most) in the local context E . Furthermore, the function *intranse*(l, Γ) finds the identities $t_1 \dots t_k$ of all concurrent threads in the transaction l and which all join in the commit. In the erroneous case of G-COMM-ERROR, the local environment E is empty; i.e., the thread executes outside of any transactions, which constitutes an error.

The next section continues with the effect system, as part of a “type and effect” system. The underlying types T include names C of classes, basic types B (natural numbers, booleans, etc.) and Void for typing side-effect-only expressions. The corresponding type system for judgements of the form $\Gamma \vdash e : T$ (“under type assumptions Γ , expression e has type T ”) is standard and omitted here (cf. the technical report [20]).

3 The Effect System

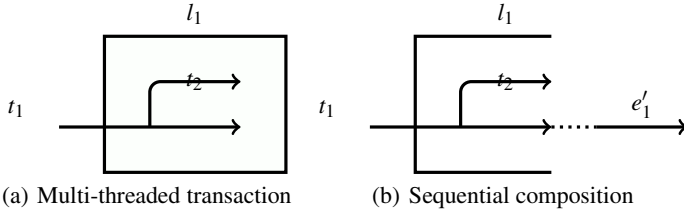
The effect system assures that starting and committing transactions is done “properly”, in particular to avoid committing when outside a transaction, which we call *commit errors*. To catch commit errors at compile time, the system keeps track of onacids and commits; we refer to the number of onacids minus the number of commits as the *balance*. E.g., for an expression $e = \text{onacid}; e_1; \text{commit}; \text{commit}$, the *balance* equals $1 - 2 = -1$. An execution of a thread is *balanced*, if there are no pending transactions, i.e., if the balance is 0. The situation gets slightly more involved when dealing with

Table 2. Semantics (local)

$E \vdash \text{let } x : T = v \text{ in } e \rightarrow E \vdash e[v/x]$	R-RED
$E \vdash \text{let } x_2 : T_2 = (\text{let } x_1 : T_1 = e_1 \text{ in } e) \text{ in } e' \rightarrow E \vdash \text{let } x_1 : T_1 = e_1 \text{ in } (\text{let } x_2 : T_2 = e \text{ in } e')$	R-LET
$E \vdash \text{let } x : T = (\text{if true then } e_1 \text{ else } e_2) \text{ in } e \rightarrow E \vdash \text{let } x : T = e_1 \text{ in } e$	R-COND ₁
$E \vdash \text{let } x : T = (\text{if false then } e_1 \text{ else } e_2) \text{ in } e \rightarrow E \vdash \text{let } x : T = e_2 \text{ in } e$	R-COND ₂
$\frac{\text{read}(r, E) = E', C(\bar{u}) \quad \text{fields}(C) = \bar{f}}{E \vdash \text{let } x : T = r.f_i \text{ in } e \rightarrow E' \vdash \text{let } x : T = u_i \text{ in } e}$	R-LOOKUP
$\frac{\text{read}(r, E) = E', C(\bar{r}) \quad \text{write}(r \mapsto C(\bar{r}) \downarrow_i', E') = E''}{E \vdash \text{let } x : T = r.f_i := r' \text{ in } e \rightarrow E'' \vdash \text{let } x : T = r' \text{ in } e}$	R-UPD
$\frac{\text{read}(r, E) = E', C(\bar{r}) \quad \text{mbody}(m, C) = (\bar{x}, e)}{E \vdash \text{let } x : T = r.m(\bar{r}) \text{ in } e' \rightarrow E' \vdash \text{let } x : T = e[\bar{r}/\bar{x}][r/\text{this}] \text{ in } e'}$	R-CALL
$\frac{r \text{ fresh} \quad E' = \text{extend}(r \mapsto C(\bar{\text{null}}), E)}{E \vdash \text{let } x : T = \text{new } C() \text{ in } e \rightarrow E' \vdash \text{let } x = r \text{ in } e}$	R-NEW
$E \vdash \text{let } x : T = \text{new } C() \text{ in } e \rightarrow E' \vdash \text{let } x = r \text{ in } e$	

Table 3. Semantics (global)

$E \vdash e \rightarrow E' \vdash e' \quad \Gamma \vdash t : E \quad \text{reflect}(t, E', \Gamma) = \Gamma'$	G-PLAIN
$\Gamma \vdash P \parallel t(e) \Longrightarrow \Gamma' \vdash P \parallel t(e')$	
$t' \text{ fresh} \quad \text{spawn}(t, t', \Gamma) = \Gamma'$	G-SPAWN
$\Gamma \vdash P \parallel t(\text{let } x : T = \text{spawn } e_1 \text{ in } e_2) \Longrightarrow \Gamma' \vdash P \parallel t(\text{let } x : T = \text{null in } e_2) \parallel t'(e_1)$	
$l \text{ fresh} \quad \text{start}(l, t, \Gamma) = \Gamma'$	G-TRANS
$\Gamma \vdash P \parallel t(\text{let } x : T = \text{onacid in } e) \Longrightarrow \Gamma' \vdash P \parallel t(\text{let } x : T = \text{null in } e)$	
$\Gamma = \Gamma'', t : E \quad E = E', l : p \quad \text{intranse}(l, \Gamma) = \vec{l} = t_1 \dots t_k$ $\text{commit}(\vec{l}, \vec{E}, \Gamma) = \Gamma' \quad t_1 : E_1, t_2 : E_2, \dots, t_k : E_k \in \Gamma \quad \vec{E} = E_1, E_2, \dots, E_k$	G-COMM
$\Gamma \vdash P \parallel \dots \parallel t_i(\text{let } x : T_i = \text{commit in } e_i) \parallel \dots \Longrightarrow \Gamma' \vdash P \parallel \dots \parallel t_i(\text{let } x : T_i = \text{null in } e_i) \parallel \dots$	
$\Gamma = \Gamma'', t : E \quad E = \emptyset$	G-COMM-ERROR
$\Gamma \vdash P \parallel t(\text{let } x : T = \text{commit in } e) \Longrightarrow \text{error}$	

**Fig. 1.** Transactions and multi-threading

multi-threading. TFJ supports not only nested transactions, but *multi-threaded* transactions: inside one transaction there may be more than one thread active at a time. Due to this internal concurrency, the effect of a transaction may be non-deterministic. Figure 1(a) shows a simple situation with two threads t_1 and t_2 , where t_1 starts a transaction with the label l_1 and spawns a new thread t_2 inside the transaction. An example expression resulting in the depicted behavior of Figure 1(b) is $e_1 = \text{onacid}; \text{spawn } e_2; e'_1$, where e_1 is the expression evaluated by thread t_1 , and e_2 by the freshly created t_2 . In TFJ's concurrency model, to terminate the parent transaction l_1 , both t_1 and t_2 must *join* via a common commit. To keep track we must take into account that e_2 and the rest e'_1 of the original thread are executed in parallel, and furthermore, that when executing e_2 in the new thread t_2 , one onacid has already been executed by t_1 , namely before the spawn-operation. Hence, we need to keep track of the balance not just for the thread under consideration, but take into account the balance of the newly created threads, as well. Even if a spawning thread and a spawned thread run in parallel, the situation wrt. the analysis is not symmetric. Considering the balance for the left of $\text{onacid}; \text{spawn } e_2$, the balance for both “threads” after execution amounts to $+1$, i.e., both threads are executing inside one enclosing transaction. When calculating the combined effect for $\text{onacid}; \text{spawn } e_2; e'_1$, the balance value of onacid is treated differently from the one of

e_2 , as the control flow of the sequential composition connects the trailing e'_1 with `onacid`, but not with the thread of e_2

To sum up: to determine the effect in terms of the balance, we need to calculate the balance for *all* threads potentially concerned, which means for the thread executing the expression being analysed plus all threads (potentially) spawned during that execution. From all threads, the one which carries the expression being evaluated plays a special role, and is treated specially. Therefore, we choose a pair of an integer n and a (finite) multi-set S of integers to represent the effect after evaluating an expression as follows:

$$n, S : \text{Int} \times (\text{Int} \rightarrow \text{Nat}) . \quad (1)$$

The integer n represents the balance of the thread of the given expression, the multi-set the balance numbers for the threads potentially spawned by the expression. We write \emptyset for the empty multi-set, \cup for the multi-set union. The multi-set can be seen as a function of type $\text{Int} \rightarrow \text{Nat}$ (the multi-set's characteristic function), and we write $\text{dom}(S)$ for the set of elements of S , ignoring their multiplicity. As an example: we use also the set-like notation $\{-3, 1, 1, 2\}$ to represent the finite mapping $-3 \mapsto 1, 1 \mapsto 2, 2 \mapsto 1$ (and all other integers to 0). As a further operation, we use “addition” and “subtraction” of such multisets and integers illustrated on a small example: $\{-3, 1, 1, 2\} + 5$ gives $\{2, 6, 6, 7\}$. Based on S , we know how many newly created threads with their corresponding balances in the current expression, including threads with the same balance. The judgements of the analysis are thus of the following form:

$$n_1 \vdash e :: n_2, S , \quad (2)$$

which reads as: starting with a balance of n_1 , executing e results in a balance of n_2 and the balances for new threads spawned by e are captured by S . The balance for the new threads in S is calculated *cumulatively*; i.e., their balance includes n_1 , the contribution of e before the thread is spawned, plus the contribution of the new thread itself.

The effect system is given in Table 4. For clarity, we do not integrate the effect system with the underlying type system. Instead, we concentrate on the effects in isolation. Variables, the null-expression, field lookup, and object creation have no effect (cf. T-VAR, T-NULL, T-LOOKUP, and T-NEW in Table 4). A field update has no effect (cf. T-UPD), as we require that the left- and the right-hand side of the assignment are already evaluated. In contrast, the two dual commands of `onacid` and `commit` have the expected effect: they simply increase, resp. decrease the balance by one (cf. T-ONACID and T-COMMIT). A class declaration (cf. T-CLASS) has no effect and no newly created threads, therefore the balance is zero and the multiset of balances equals \emptyset . Rule T-METH deals with method declarations. In this rule, we require that all spawned threads in the method body must have the balance 0 after evaluating the expression e , that the balance of the method itself has the form $n_1 \rightarrow n_2$ where n_1 is interpreted as pre-condition, i.e., it is safe to call the method *only* in a state where the balance is at least n_1 . The number n_2 as the post-condition corresponds to the balance after exiting the method, when called with balance n_1 as pre-condition. The precondition n_1 is needed to assure that at the call-sites the method is only used where the execution of the method body does not lead to a negative balance (see also the T-CALL-rules below). Rule T-SUB captures a notion of *subsumption* where by $S_1 \leq S_2$ we mean the subset

Table 4. Effect system

$\frac{}{n \vdash x :: n, \emptyset} \text{ T-VAR}$	$\frac{}{n \vdash \text{null} :: n, \emptyset} \text{ T-NULL}$	$\frac{}{n \vdash v.f :: n, \emptyset} \text{ T-LOOKUP}$	$\frac{}{n \vdash \text{new } C :: n, \emptyset} \text{ T-NEW}$
$\frac{n \vdash v_1 :: n, \emptyset \quad n \vdash v_2 :: n, \emptyset}{n \vdash v_1.f_j := v_2 :: n, \emptyset} \text{ T-UPD}$	$\frac{}{n \vdash \text{onacid} :: n + 1, \emptyset} \text{ T-ONACID}$		$\frac{n \geq 1}{n \vdash \text{commit} :: n - 1, \emptyset} \text{ T-COMMIT}$
$\frac{K = C(\vec{f} : \vec{T}) \{ \text{this}.\vec{f} := \vec{f} \} \quad \vdash \vec{M} :: \vec{n}_1 \rightarrow \vec{n}_2, \vec{S}}{\vdash \text{class } C \{ \vec{f} : \vec{T}; K; \vec{M} \} :: 0, \emptyset} \text{ T-CLASS}$		$\frac{n_1 \vdash e :: n_2, \{0, \dots\}}{\vdash m(\vec{x} : \vec{T}) \{e\} :: n_1 \rightarrow n_2, \{0, \dots\}} \text{ T-METH}$	
$\frac{n \vdash e :: n', S_1 \quad S_1 \leq S_2}{n \vdash e :: n', S_2} \text{ T-SUB}$	$\frac{n_0 \vdash e_1 :: n_1, S_1 \quad n_1 \vdash e_2 :: n_2, S_2}{n_0 \vdash \text{let } x : T = e_1 \text{ in } e_2 :: n_2, S_1 \cup S_2} \text{ T-LET}$		
$\frac{n \vdash e :: n', S}{n \vdash \text{spawn } e :: n, S \cup \{n'\}} \text{ T-SPAWN}$	$\frac{n \vdash v :: n, \emptyset \quad n \vdash e_1 :: n', S_1 \quad n \vdash e_2 :: n', S_2}{n \vdash \text{if } v \text{ then } e_1 \text{ else } e_2 :: n', S_1 \cup S_2} \text{ T-COND}$		
$\frac{n \vdash v :: n, \emptyset \quad n \vdash v_i :: n, \emptyset \quad \text{mtype}(C, m) :: n'_1 \rightarrow n'_2, S \quad n = n'_1}{n \vdash v.m(\vec{v}) :: n'_2 - n'_1 + n, S - n'_1 + n} \text{ T-CALL}_1$			
$\frac{n \vdash v :: n, \emptyset \quad n \vdash v_i :: n, \emptyset \quad \text{mtype}(C, m) :: n'_1 \rightarrow n'_2, \emptyset \quad n > n'_1}{n \vdash v.m(\vec{v}) :: n'_2 - n'_1 + n, \emptyset} \text{ T-CALL}_2$			
$\frac{\{E\} \vdash e :: 0, \{0, 0, \dots\}}{t : E \vdash t(e) : ok} \text{ T-THREAD}$		$\frac{\Gamma_1 \vdash P_1 : ok \quad \Gamma_2 \vdash P_2 : ok}{\Gamma_1, \Gamma_2 \vdash P_1 \parallel P_2 : ok} \text{ T-PAR}$	

relation on multi-sets¹. In a let-expression (cf. T-LET), representing sequential composition, the effects are accumulated. Creating a new thread by executing spawn e does not change the balance of the executing thread (cf. T-SPAWN). The spawned expression e in the new thread is analyzed starting with the same balance n in its pre-state. The resulting balance n' of the new thread is given back in the conclusion as part of the balances of the spawned threads, i.e., as part of the multi-set. For conditionals if v then e_1 else e_2 (cf. T-COND), the boolean condition v does not change the balance, and the rule insists that the two branches e_1 and e_2 agree on a balance n' .

For method calls, we distinguish two situations (cf. T-CALL₁ and T-CALL₂), depending on whether the method being called creates new threads or not. In the latter case, the multi-set of balances for method m in class C is required to be empty by the third premise of the rule. In that situation, the precondition of the method can be interpreted in a “loose” manner: the current balance n in the state before the call must be *at least* as big as the pre-condition n'_1 . If, however, the method may spawn a new thread (cf. T-CALL₁), the pre-condition is interpreted *strictly*, i.e., we require $n = n'_1$ (with this equality, T-CALL₁ could be simplified; we chose this representation to stress the connection with T-CALL₂, where $n > n'_1$). Allowing the loose interpretation also in that situation would make the method callable in different levels of nestings at the caller side; however, only exactly *one* level actually is appropriate, as with concurrent

¹ The non-structural rule of subsumption makes the system non syntax-directed. To turn it to an algorithm, one would have to disallow subsumption and derive a minimal multiset instead.

threads inside a transaction, all threads must join in a commit to terminate the transaction. A thread $t\langle e \rangle$ is well-typed (cf. T-THREAD), if the expression has balance 0 after termination, starting with a balance corresponding to the length $|E|$ of the local environment E . We use *ok* to indicate that the thread is well-typed, i.e., without commit-error. This balance in the pre-state corresponds to the level of nesting inside transactions, the thread $t\langle e \rangle$ currently executes in. A program is well typed, if all threads in the system are well-typed (cf. T-PAR). We illustrate the system with the following two examples:

Example 1. The following derivation applies the effect system to the expression e_1 ; $\text{spawn}(e_2; \text{spawn } e_3)$; $e_4 :: n_4, \{n_2, n_3\}$, when starting with a balance of 0.

$$\begin{array}{c}
 \frac{n_2 \vdash e_3 :: n_3, \{\}}{n_1' \vdash e_2 :: n_2, \{\} \quad n_2 \vdash \text{spawn } e_3 :: n_2, \{n_3\}} \\
 \frac{n_1' \vdash (e_2; \text{spawn } e_3) :: n_2, \{n_3\}}{n_1' \vdash \text{spawn}(e_2; \text{spawn } e_3) :: n_1', \{n_2, n_3\} \quad n_1' \vdash e_4 :: n_4, \{\}} \\
 \frac{0 \vdash e_1 :: n_1', \{\} \quad n_1' \vdash \text{spawn}(e_2; \text{spawn } e_3); e_4 :: n_4, \{n_2, n_3\}}{0 \vdash e_1; \text{spawn}(e_2; \text{spawn } e_3); e_4 :: n_4, \{n_2, n_3\}}
 \end{array}$$

The derivation demonstrates sequential composition and thread creation with a starting balance of 0 for simplicity. Remember that sequential composition $e_1; e_2$ is syntactic sugar for $\text{let } x:T = e_1 \text{ in } e_2$, where x does not occur free in e_2 ; i.e., assume that the expressions e_1, \dots, e_4 themselves have the following balances $0 \vdash e_i :: n_i', \{\}$, which implies $n_1' \vdash e_2 :: n_1' + n_2' = n_2, \{\}$, $n_2 \vdash e_3 :: n_2 + n_3' = n_3, \{\}$, and $n_1' \vdash e_4 :: n_1' + n_4' = n_4, \{\}$. \square

Example 2. Assume the following code fragment:

```

...
void n(){ onacid; m(10); }

void m(i){
    commit;
    if (i ≤ 0)
    then onacid;
    else ....; onacid; this.m(i-1); }

void main(){ n(); commit; }

```

First observe that the program shows no commit-errors during run-time. Method m calls itself recursively and the two branches of the conditional in its body both execute one *onacid* each. Especially, method m is called (in this fragment) only via method n , especially after n has performed an *onacid*, i.e., m is called *inside* one transaction. If m were called *outside* a transaction it would result in an error, as the body of m starts by executing a *commit*-statement. In our effect system, method m can be declared as of effect $1 \rightarrow 1$, which expresses not only that the body of m does not change the balance, but that as a precondition, it must be called *only* from call-sites where the balance is ≥ 1 , as is the case in the body of n (cf. also T-METH and T-CALL). So the declarations of the two shown methods are of the form $n() : \text{Void} \rightarrow \text{Void}, 0 \rightarrow 1$ and $m(i) : \text{Int} \rightarrow \text{Void}, 1 \rightarrow 1$. For recursive calls, an effect like $1 \rightarrow 1$ can be interpreted as *loop invariant*: the body of the method must not change the balance to be well-typed. However, not every method needs to be balanced; the non-recursive method n is one example which (together with the call to m) has a net-balance of 1. \square

4 Soundness of the Type and Effect System

Next we prove that the type and effect system does what it is designed to do, namely absence of commit errors.

Lemma 1 (Subject reduction (local)). *Let $n = |E|$. If $n \vdash e :: n', S'$ and $E \vdash e \rightarrow E' \vdash e'$, then $|E'| = n$ and $n \vdash e' :: n', S'$.*

Proof. By straightforward induction on the rules of Table 2, observing that by the properties of *read*, *write*, and *extend*, $|E| = |E'|$. \square

The global semantics accesses and changes the global environments Γ . These manipulations are captured in various functions, which are kept “abstract” in this semantics (as in [15]). To perform the subject reduction proof, however, we need to impose certain requirements on those functions:

Definition 3. *The properties of the abstract functions are specified as follows:*

1. *The function *reflect* satisfies the following condition: if $\text{reflect}(t, E, \Gamma) = \Gamma'$ and $\Gamma = t_1:E_1, \dots, t_k:E_k$, then $\Gamma' = t_1:E'_1, \dots, t_k:E'_k$ with $|E_i| = |E'_i|$ (for all i).*
2. *The function *spawn* satisfies the following condition: Assume $\Gamma = t : E, \Gamma''$ and $t' \notin \Gamma$ and $\text{spawn}(t, t', \Gamma) = \Gamma'$, then $\Gamma' = \Gamma, t':E'$ s.t. $|E| = |E'|$.*
3. *The function *start* satisfies the following condition: if $\text{start}(l, t_i, \Gamma) = \Gamma'$ for a $\Gamma = t_1:E_1, \dots, t_i:E_i, \dots, t_k:E_k$ and for a fresh l , then $\Gamma' = t_1:E_1, \dots, t_i:E'_i, \dots, t_k:E_k$, with $|E'_i| = |E_i| + 1$.*
4. *The function *intranse* satisfies the following condition: Assume $\Gamma = \Gamma'', t:E$ s.t. $E = E', l:\rho$ and $\text{intranse}(l, \Gamma) = \vec{t}$, then

 - (a) $t \in \vec{t}$ and
 - (b) for all $t_i \in \vec{t}$ we have $\Gamma = \dots, t_i : (E'_i, l:\rho_i), \dots$
 - (c) for all threads t' with $t' \notin \vec{t}$ and where $\Gamma = \dots, t' : (E', l':\rho')$, we have $l' \neq l$.*
5. *The function *commit* satisfies the following condition: if $\text{commit}(\vec{t}, \vec{E}, \Gamma) = \Gamma'$ for a $\Gamma = \Gamma'', t:(E, l:\rho)$ and for a $\vec{t} = \text{intranse}(l, \Gamma)$ then $\Gamma' = \dots, t_j:E'_j, \dots, t_i:E'_i, \dots$ where $t_i \in \vec{t}, t_j \notin \vec{t}, t_j:E_j \in \Gamma$, with $|E'_j| = |E_j|$ and $|E'_i| = |E_i| - 1$.*

Lemma 2 (Subject reduction). *If $\Gamma \vdash P : ok$ and $\Gamma \vdash P \Longrightarrow \Gamma' \vdash P'$, then $\Gamma' \vdash P' : ok$.*

Proof. Proceed by case analysis on the rules of the operational semantics from Table 3 (except rule G-COMMERROR for commit errors). For simplicity (and concentrating on the effect, not the values of expressions) we use ; for sequential composition in the proof, and not the more general let-construct.

Case: G-PLAIN

From the premises of the rule, we get for the form of the program that $P = P'' \parallel t\langle e \rangle$, furthermore for t 's local environment $\Gamma \vdash t : E$ and $E \vdash e \rightarrow E' \vdash e'$ as a local step. Well-typedness $\Gamma \vdash P : ok$ implies $n \vdash e :: n', S'$ for some n' and S' , where $n = |E|$. By subject reduction for the local steps (Lemma 1) $n \vdash e' :: n', S'$. By the properties of the *reflect*-operation, $|E'| = n$, so we derive for the thread t

$$\frac{n \vdash e' :: 0, \{0, \dots\}}{\Gamma', t:E' \vdash t\langle e' \rangle : ok}$$

from which the result $\Gamma' \vdash P'' \parallel t\langle e' \rangle : ok$ follows (using T-PAR and the properties of *reflect* from Definition 3[1]).

Case: G-SPAWN

In this case, $P = P'' \parallel t\langle \text{spawn } e_1; e_2 \rangle$ and $P' = P'' \parallel t\langle \text{null}; e_2 \rangle \parallel t'\langle e_1 \rangle$ (from the premises of G-SPAWN). The well-typedness assumption $\Gamma \vdash P : ok$ implies the following sub-derivation:

$$\frac{\frac{\frac{n \vdash e_1 : 0, S_1}{n \vdash \text{spawn } e_1 : n, S_1 \cup \{0\}} \quad n \vdash e_2 : 0, S_2}{n \vdash \text{spawn } e_1; e_2 : 0, \{0, \dots\}}}{t:E \vdash t\langle \text{spawn } e_1; e_2 \rangle : ok} \quad (3)$$

with $S_1 = \{0, \dots\}$ and $S_2 = \{0, \dots\}$. By the properties of *reflect*, the global environment Γ' after the reduction step is of the form $\Gamma, t':E'$ where t' is fresh and $|E'| = |E|$ (see Definition 3.2). So we can derive

$$\frac{\frac{\frac{n \vdash e_1 : 0, \{0, \dots\}}{t:E \vdash t\langle \text{null}; e_2 \rangle : ok} \quad t':E' \vdash t'\langle e_1 \rangle : ok}{t:E, t':E' \vdash t\langle \text{null}; e_2 \rangle \parallel t'\langle e_1 \rangle : ok}}$$

The left sub-goal follows from T-THREAD, T-SEQ, T-NUL, and the right sub-goal of the previous derivation (3). The right open sub-goal directly corresponds to the left sub-goal of derivation (3).

Case: G-TRANS

In this case, $P = P'' \parallel t\langle \text{onacid}; e \rangle$ and $P' = P'' \parallel t\langle \text{null}; e \rangle$. The well-typedness assumption $\Gamma \vdash P : ok$ implies the following sub-derivation (assume that $|E| = n$):

$$\frac{\frac{\frac{n \vdash \text{onacid} :: n+1, \emptyset \quad n+1 \vdash e :: 0, \{0, \dots\}}{n \vdash \text{onacid}; e :: 0, \{0, \dots\}}}{t:E \vdash t\langle \text{onacid}; e \rangle : ok} \quad (4)$$

For the global environment Γ' after the step, we are given $\Gamma' = \text{start}(l, t, \Gamma)$ from the premise of rule G-TRANS. By the properties of *start* from Definition 3.3, we have $\Gamma' = \Gamma'', t:E'$ with $|E'| = n+1$. So with the help of right sub-goal of the previous derivation (4), we can derive for thread t after the step:

$$\frac{n+1 \vdash e :: 0, \{0, \dots\}}{t:E' \vdash t\langle e \rangle : ok}$$

Since furthermore the local environments of all other threads remain unchanged (cf. again Definition 3.3), the required $\Gamma' \vdash P' : ok$ can be derived, using T-PAR.

Case: G-COMM

In this case, $P = P'' \parallel \vec{t}\langle \text{commit}; \vec{e} \rangle$ and $P' = P'' \parallel \vec{t}\langle \vec{e} \rangle$. The well-typedness assumption $\Gamma \vdash P : ok$ implies the following sub-derivation for thread t :

$$\frac{\frac{\frac{n \vdash \text{commit} :: n-1, \emptyset \quad n-1 \vdash e_i : 0, \{0, \dots\}}{n \vdash \text{commit}; e_i : 0, \{0, \dots\}}}{t_i:E_i \vdash t_i\langle \text{commit}; e_i \rangle : ok} \quad (5)$$

For the global environment Γ' after the step, we are given $\Gamma' = \text{commit}(\vec{t}, \vec{E}, \Gamma)$ from the premise of rule G-TRANS, where $\vec{t} = \text{intranse}(l, \Gamma)$ and \vec{E} are the corresponding local environments. By the properties of commit from Definition 3.5, we have for the local environments \vec{E}' of threads \vec{t} after the step that $|E'_i| = n - 1$. So we obtain by T-THREAD, using the right sub-goal of derivation (5):

$$\frac{n - 1 \vdash e_i :: 0, \{0, \dots\}}{t_i : E'_i \vdash t_i(e_i) : \text{ok}}$$

For the threads $t_j(e_j)$ different from \vec{t} , according to the Definition 3.5, we have $|E'_j| = |E_j|$ so $t_j : E'_j \vdash t_j(e'_j) : \text{ok}$ straightforwardly. As a result, we have $\Gamma' \vdash P' : \text{ok}$. \square

Lemma 3. *If $\Gamma \vdash P : \text{ok}$ then it is not the case that $\Gamma \vdash P \implies \text{error}$.*

Proof. Let $\Gamma \vdash P : \text{ok}$ and assume for a contradiction that $\Gamma \vdash P \rightarrow \text{error}$. From the rules of the operational semantics it follows that $P = t \langle \text{commit}; e \rangle \parallel P'$ for some thread t , where the step $\Gamma \vdash P \rightarrow \text{error}$ is done by t (executing the commit-command). Furthermore, the local environment E for the thread t is empty:

$$\frac{E = \emptyset}{\Gamma', t : E \vdash t \langle \text{commit}; e \rangle \parallel P' \rightarrow \text{error}} \text{G-COMM}$$

To be well-typed, i.e., for the judgment $\Gamma \vdash t \langle \text{commit}; e \rangle \parallel P' : \text{ok}$ to be derivable, it is easy to see that the derivation must contain $\Gamma', t : \emptyset \vdash t \langle \text{commit}; e \rangle : n, S$ as sub-derivation (for some n and S). By inverting rule T-THREAD, we get that $0 \vdash \text{let commit in } e : 0, \{0, 0, \dots\}$ is derivable (since $|E| = 0$). This is a contradiction, as the balance after commit would be negative (inverting rules T-LET and T-COMMIT). \square

Corollary 1 (Well-typed programs are commit-error free). *If $\Gamma \vdash P : \text{ok}$ then it is not the case that $\Gamma \vdash P \implies^* \text{error}$,*

Proof. A direct consequence of the subject reduction Lemma 2 and Lemma 3. \square

5 Conclusion

This work took the TFJ language design from [15] as starting point. That paper is not concerned with static analysis, but develops and investigates two different operational semantics for TFJ that assure transactional guarantees. As mentioned, however, the flexibility of the language may lead to run-time errors when executing a commit outside any transaction; we called such situations *commit-errors*. To statically prevent commit-errors, we presented a static type and effect system, which keeps track of the commands for starting and finishing transactions. We proved soundness of the type system.

A comparison with explicit locks of Java. The built-in support for concurrency control in Java is lock-based; each object comes equipped with a (re-entrant) lock, which can be used to specify synchronized blocks and, as a special case, synchronized methods. The lock can achieve mutual exclusion between threads that compete for the lock before

doing something critical. Thus, the built-in, lock-based (i.e., “pessimistic”) concurrency control in Java offers *lexically scoped* protection based on mutual exclusion. While offering basic concurrency control, the scheme has been criticized as too rigid, and consequently, Java 5 now additionally supports explicit locks with non-lexical scope. The `ReentrantLock` class and the `Lock` interface allow more freedom, offering explicit `lock` and `unlock` operations. Locking and unlocking can be compared, to some extent, to starting and committing a transaction, even if there are differences especially wrt. failure and progress properties. See e.g., [3] for a discussion of such differences. Besides the more behavioral differences, such as different progress guarantees or deadlocking behavior, the lock handling in Java 5 and the transactional model of TFJ differ in the following aspects, as relevant for the type analysis (cf. Table 5).

One basic difference is that we proposed a *static* scheme to catch commit errors, whereas in Java, improper use of locking and unlocking is checked at *run-time*. Both schemes, as mentioned, have all the flexibility of non-lexical scoping. The rest of Table 5 deals with the structure of protected areas (the transaction or the execution protected by a lock) and the connection to the threading model. One difference is that locks have an identity available at the program level, whereas transactions have not. Furthermore, locks and monitors in Java are *re-entrant*, i.e., one particular thread holding a lock can recursively re-enter a critical section or monitor. Re-entrance is not an issue in TFJ: a thread leaves a transaction by committing it (which terminates the transaction), hence re-entrance into the same transaction makes no sense. Transactions in TFJ can be nested. Of course, in Java, a thread can hold more than one lock at a time; however, the critical sections protected by locks do not follow a first-in-last-out discipline, and the section are not nested as they are independent. For nested transactions in contrast, a commit to a child transaction is propagated to the surrounding parent transaction, but not immediately further, until that parent commits its changes in turn. Finally, TFJ allows multi-threaded transactions, whereas monitors and locks in Java are meant to ensure mutual exclusion. In particular, if an activity inside a monitor spawns a new thread, the new thread starts executing *outside* any monitor, in other words, a new thread holds *no* locks. In [17], we discuss the differences and similarities in more depth by comparing the analysis developed here with a corresponding one that deals with the safe use of a statically allocated number of locks.

Related work. There have been a number of further proposals for integrating transactional features into programming languages. For transactional languages, lexical scope for transactions, so called atomic blocks, have been proposed, using e.g., an

Table 5. Transactional Featherweight Java and explicit locks of Java

	Java 5.0	TFJ
when?	run-time	compile time
non-lexical scope	yes	yes
program level identity	yes	no
re-entrance	yes	no
nested transactions/critical sections	no	yes
internal multi-threading	no	yes

atomic-construct or similar. Examples are Atomos [4], the AME calculus [1], and many proposals for software transactional memory [8,21], but none of them deals with assuring statically proper use of the corresponding constructs. When dealing with concurrency, most static analyses focus on avoiding data races and deadlocks, especially for multi-threaded Java programs. Static type systems have also been used to impose restrictions assuring transactional semantics, e.g. in [9,11,14]. A type system for *atomicity* is presented in [7,6]. [2] develops a type system for statically assuring proper lock handling for the JVM, i.e., on the level of byte code. Their system assures what is known as *structured locking*, i.e., (in our terminology), each method body is balanced as far as the locks are concerned, and at no point, the balance reaches below 0. Since the work does not consider non-lexical locking as in Java 5, the conditions apply *per method* only. Also the Rcc/Java type system tries to keep track of which locks are held (in an approximate manner), noting which field is guarded by which lock, and which locks must be held when calling a method. Especially *safe lock* analysis, supported e.g. by the Indus tool [19] as part of Bandera, is a static analysis that checks whether a lock is held indefinitely (in the context of multi-threaded Java). Software model checking is a prominent, alternative way to assure quality of software. By using some form of abstraction (typically ignoring data parts and working on an abstract, automata-based representation), model checking can be used as a form of static analysis of concrete programs, as well. The Blast analyzer [11] allows automatic verification for checking temporal safety properties of C programs (using counter-example guided abstraction refinement), and has been extended to deal with concurrent programs, as well [5]. Similarly, Java PathFinder is an automatic, model-checking tool (based on Spin) to analyze Java programs [10].

Future work. The work presented here can be extended to deal with more complex language features, e.g. when dealing with higher-order functions. In that setting, the effect part and its connection to the type system become more challenging. Furthermore, we plan to adopt the results for a different language design, more precisely to the language Creol [16], which is based on asynchronously communicating, active objects, in contrast to Java, whose concurrency is based on multi-threading. As discussed, there are similarities between lock-handling in Java 5 and the transactions as treated here. We plan to use similar techniques as explored here to give static guarantees for lock-based concurrency, as well. Of practical relevance is to extend the system from type *checking* to type *inference*, potentially along the lines [12].

Acknowledgements. We thank the anonymous reviewers for their helpful suggestions.

References

1. Abadi, M., Birell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: Proceedings of POPL 2008. ACM, New York (2008)
2. Bigliardi, G., Laneve, C.: A type system for JVM threads. In: Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC 2000), p. 2003 (2000)
3. Blundell, C., Lewis, E.C., Martin, M.K.: Subtleties of transactional memory atomicity semantics. IEEE Computer Architecture Letters 5(2) (2006)

4. Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., Oluktun, K.: The ATOMOS transactional programming language. In: ACM Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada. ACM, New York (2006)
5. Davare, A.: Concurrent BLAST, Internal Report, EECS Berkely. Mentors Rupak Majumdar and Ranjit Jhala, Mentors (2003)
6. Flanagan, C., Freund, S.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Proceedings of POPL 2004, pp. 256–267. ACM, New York (2004)
7. Flanagan, C., Quadeer, S.: A type and effect system for atomicity. In: ACM Conference on Programming Language Design and Implementation, San Diego, California. ACM, New York (2003)
8. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Eighteenth OOPSLA 2003. SIGPLAN Notices. ACM, New York (2003)
9. Harris, T., Peyton Jones, S.M.S., Herlihy, M.: Composable memory transactions. In: PPOPP 2005: 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60 (June 2005)
10. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer* 2(4), 366–381 (2000)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
12. Igarashi, A., Kobayashi, N.: Resource usage analysis. *ACM Transactions on Programming Languages and Systems* 27(2), 264–313 (2005)
13. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: OOPSLA 1999. SIGPLAN Notices, pp. 132–146. ACM, New York (1999)
14. Isard, M., Birell, A.: Automatic mutual exclusion. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems (2007)
15. Jagannathan, S., Vitek, J., Welc, A., Hosking, A.: A transactional object calculus. *Science of Computer Programming* 57(2), 164–186 (2005)
16. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science* 365(1-2), 23–66 (2006)
17. Tran, T.M.T., Owe, O., Steffen, M.: Safe typing for transactional vs. lock-based concurrency in multi-threaded Java. In: Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010 (accepted for publication 2010)
18. Nielson, F., Nielson, H.-R., Hankin, C.L.: Principles of Program Analysis. Springer, Heidelberg (1999)
19. Ranganath, V.P., Hatcliff, J.: Slicing concurrent Java programs using Indus and Kaveri. *International Journal of Software Tools and Technology Transfer* 9(5), 489–504 (2007)
20. Steffen, M., Tran, T.M.T.: Safe commits for Transactional Featherweight Java. Technical Report 392, University of Oslo, Dept. of Computer Science (October 2009)
21. Welc, A., Jagannathan, S., Hosking, A.: Transactional monitors for concurrent objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 518–541. Springer, Heidelberg (2004)

Certified Absence of Dangling Pointers in a Language with Explicit Deallocation*

Javier de Dios, Manuel Montenegro, and Ricardo Peña

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

jdcastro@aventia.com, montenegro@fdi.ucm.es, ricardo@sip.ucm.es

Abstract. *Safe* is a first-order eager functional language with facilities for programmer controlled destruction of data structures. It provides also *regions*, i.e. disjoint parts of the heap, where the program allocates data structures, so that the runtime system does not need a garbage collector. A region is a collection of cells, each one big enough to allocate a data constructor. Deallocating cells or regions may create dangling pointers. The language is aimed at inferring and certifying memory safety properties in a Proof Carrying Code like environment. Some of its analyses have been presented elsewhere. The one relevant to this paper is a type system and a type inference algorithm guaranteeing that well-typed programs will be free of dangling pointers at runtime.

Here we present how to generate formal certificates about the absence of dangling pointers property inferred by the compiler. The certificates are Isabelle/HOL proof scripts which can be proof-checked by this tool when loaded with a database of previously proved theorems. The key idea is proving an Isabelle/HOL theorem for each syntactic construction of the language, relating the static types inferred by the compiler to the dynamic properties about the heap that will be satisfied at runtime.

Keywords: Memory management, type-based analysis, formal certificates, proof assistants.

1 Introduction

Certifying program properties consists of providing mathematical evidence about them. In a Proof Carrying Code (PCC) environment [14], these proofs should be checked by an appropriate tool. The certified properties may be obtained either manually, interactively, or automatically, but whatever is the effort needed for generating them, the PCC paradigm insists on their checking to be fully automatic.

In our setting, the certified property (absence of dangling pointers) is automatically inferred as the product of several static analyses, so that the certificate can be generated by the compiler without any human intervention. Certifying the

* Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), S2009/TIC-1465 (PROMETIDOS), and MEC FPU grant AP2006-02154.

inferred property is needed in our case to convince a potential consumer that the static analyses are sound and that they have been correctly implemented in the compiler.

Our functional language *Safe*, described below, is equipped with type-based analyses for inferring *regions* where data structures are located [13], and for detecting when a program with explicit deallocation actions is free of dangling pointers [12]. One may wonder why a functional language with explicit deallocation may be useful and why not using a more conventional one such as e.g. C. Explicit deallocation is a low-level facility which, when used without restrictions, may create complex heap structures and programs difficult or impossible to analyse for pointer safety. On the contrary, functional languages have more structure and the explicit deallocation can be confined to a small part of it (in our case, to pattern matching), resulting in heap-safe programs most of the time and, more importantly, amenable to a safety analysis in an automatic way.

Region inference was proved optimal in [13]: assigning data to regions minimises their lifetimes, subject to allocating/deallocating regions in a stack-based way. On the other hand, explicit deallocation eliminates garbage before the region mechanism does, so memory leaks are not a major concern here.

The above analyses have been manually proved correct in [11], but we embarked on the certification task by several reasons:

- The proof in [11] was very much involved. There were some subtleties that we wanted to have formally verified in a proof assistant.
- The implementation was also very involved. Generating and checking certificates is also a way of increasing our trust in the implementation.
- A certificate is a different matter than proving analyses correct, since the proof it contains must be related to every specific compiled program.

In this paper we describe how to create a certificate from the type annotations inferred by the analyses. The key idea is creating a database of theorems, proved once forever, relating these static annotations to the dynamic properties the compiled programs are expected to satisfy. There is a *proof rule* for each syntactic construction of the language and a theorem proving its soundness. These proof-rules generate *proof obligations*, which the generated certificate must discharge. We have chosen the proof assistant Isabelle/HOL [16] both for constructing and checking proofs. To the best of our knowledge, this is the first system certifying absence of dangling pointers in an automatic way.

The certificates are produced at the intermediate language level called *Core-Safe*, at which also the analyses are carried out. This deviates a bit from the standard PCC paradigm where certificates are at the bytecode/assembly language level, i.e. they certify properties satisfied by the executable code. We chose instead to formally verify the compiler’s back-end: *Core-Safe* is translated in two steps to the bytecode language of the Java Virtual Machine, and these steps have been verified in Isabelle/HOL [76], so that the certified property is preserved across compilation. This has saved us the (huge) effort of translating *Core-Safe* certificates to the JVM level, while nothing essential is lost: a scenario

can be imagined where the *Core-Safe* code and its certificate are sent from the producer to a consumer and, once validated, the consumer uses the certified back-end for generating the executable code. On the other hand, our certificates are smaller than the ones which could be obtained at the executable code level.

In the next section we describe the relevant aspects of *Safe*. In Sec. 3 a first set of proof rules related to explicit deallocation is presented, while a second set related to implicit region deallocation is explained in Sec. 4. Section 5 is devoted to certificate generation and Sec. 6 presents related work and concludes.

2 The Language

Safe is a first-order eager language with a syntax similar to Haskell's. Its runtime system uses *regions*, i.e. disjoint parts of the heap where the program allocates data structures. The smallest memory unit is the *cell*, a contiguous memory space big enough to hold a data construction. A cell contains the mark of the constructor and a representation of the free variables to which the constructor is applied. These may consist either of basic values, or of pointers to other constructions. Each cell is allocated at constructor application time. A *region* is a collection of cells. It is created empty and it may grow and shrink while it is active. Region deallocation frees all its cells. The allocation and deallocation of regions is bound to function calls. A *working region*, denoted by *self*, is allocated when entering the call and deallocated when exiting it. Inside the function, data structures not belonging to the output may be built there. The region arguments are explicit in the intermediate code but not in the source, since they are inferred by the compiler [13]. The following list sorting function builds an intermediate tree not needed in the output:

```
treесort xs = inorder (makeTree xs)
```

After region inference, the code is annotated with region arguments (those occurring after the @):

```
treесort xs @ r = inorder (makeTree xs @ self) @ r
```

so that the tree is created in *treесort*'s *self* region and deallocated upon termination of *treесort*.

Besides regions, destruction facilities are associated with pattern matching. For instance, we show here a destructive function splitting a list into two:

```
unshuffle []! = ([], [])
unshuffle (x:xs)! = (x:xs2, xs1) where (xs1, xs2) = unshuffle xs
```

The ! mark is the way programmers indicate that the matched cell must be deleted. The space consumption is reduced with respect to a conventional version because, at each recursive call, a cell is deleted by the pattern matching. At termination, the whole input list has been returned to the runtime system.

The *Safe* front-end desugars *Full-Safe* and produces a bare-bones functional language called *Core-Safe*. The transformation starts with region inference and continues with Hindley-Milner type inference, pattern matching desugaring, and

$prog \rightarrow$	$\overline{data_i^n}; \overline{dec_j^m}; e$	{Core-Safe program}
$data \rightarrow$	$\mathbf{data} \ T \ \overline{\alpha_i^n} \ @ \ \overline{\rho_j^m} = \overline{C_k \ \overline{t_{ks}^{nk}} \ @ \ \rho_m}$	{recursive, polymorphic data type}
$dec \rightarrow$	$f \ \overline{x_i^n} \ @ \ \overline{r_j^l} = e$	{recursive, polymorphic function}
$e \rightarrow$	a	{atom: literal c or variable x }
	$x \ @ \ r$	{copy data structure x into region r }
	$x!$	{reuse data structure x }
	$a_1 \oplus a_2$	{primitive operator application}
	$f \ \overline{a_i^n} \ @ \ \overline{r_j^l}$	{function application}
	$\mathbf{let} \ x_1 = be \ \mathbf{in} \ e$	{non-recursive, monomorphic}
	$\mathbf{case} \ x \ \mathbf{of} \ \overline{alt_i^n}$	{read-only case}
	$\mathbf{case!} \ x \ \mathbf{of} \ \overline{alt_i^n}$	{destructive case}
$alt \rightarrow$	$C \ \overline{x_i^n} \ \rightarrow \ e$	{case alternative}
$be \rightarrow$	$C \ \overline{a_i^n} \ @ \ r$	{constructor application}
	e	

Fig. 1. Core-Safe syntax

some other simplifications. In Fig. 1 we show the syntax of *Core-Safe*. A program is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression e whose value is the program result. The over-line abbreviation $\overline{x_i^n}$ stands for $x_1 \cdots x_n$. **case!** expressions implement destructive pattern matching, constructions are only allowed in **let** bindings, and atoms —or just variables— are used in function applications, **case/case!** discriminant, copy and reuse. Region arguments are explicit in constructor and function applications and in copy expressions. As an example, we show the *Core-Safe* version of the `unshuffle` function above:

```

unshuffle x34 @ r1 r2 r3 = case! x34 of
  x49 : x50 -> let x40 = unshuffle x50 @ r2 r1 self in
                let x15 = case x40 of (x45,x46) -> x45 in
                let x16 = case x40 of (x47,x48) -> x48 in
                let x38 = x49 : x16 @ r1 in
                let x39 = (x38,x15) @ r3 in x39
[]           -> let x36 = [] @ r1 in
                let x35 = [] @ r2 in
                let x37 = (x36,x35) @ r3 in x37

```

2.1 Operational Semantics

In Figure 2 we show the big-step operational semantics rules of the most relevant core language expressions. We use v, v_i, \dots to denote either heap pointers or basic constants, p, p_i, q, \dots to denote heap pointers, and a, a_i, \dots to denote either program variables or basic constants (atoms). The former are named x, x_i, \dots and the latter c, c_i etc. Finally, we use r, r_i, \dots to denote region arguments.

A judgement of the form $E \vdash (h, k), e \Downarrow (h', k), v$ states that expression e is successfully reduced to normal form v under runtime environment E and heap h with $k+1$ regions, ranging from 0 to k , and that a final heap h' with $k+1$ regions is produced as a side effect. Runtime environments E map program variables to

$$\begin{array}{c}
 E \vdash (h, k), c \Downarrow (h, k), c \text{ [Lit]} \quad E[x \mapsto v] \vdash (h, k), x \Downarrow (h, k), v \text{ [Var}_1\text{]} \\
 \\
 \frac{(f \overline{x_i^n} @ \overline{r_j^m} = e) \in \Sigma}{\overline{[x_i \mapsto E(a_i)]^n}, \overline{[r_j \mapsto E(r_j^m)]^m}, \text{self} \mapsto k+1] \vdash (h, k+1), e \Downarrow (h', k+1), v} \text{ [App]} \\
 \frac{E \vdash (h, k), f \overline{a_i^n} @ \overline{r_j^m} \Downarrow (h' \upharpoonright_k, k), v}{E \vdash (h, k), \mathbf{let} \ x_1 = e_1 \ \mathbf{in} \ e_2 \Downarrow (h'', k), v} \text{ [Let]} \\
 \frac{j \leq k \quad \text{fresh}(p) \quad E \cup [x_1 \mapsto p] \vdash (h \uplus [p \mapsto (j, C \overline{v_i^n})], k), e_2 \Downarrow (h', k), v}{E[r \mapsto j, \overline{a_i} \mapsto \overline{v_i^n}] \vdash (h, k), \mathbf{let} \ x_1 = C \overline{a_i^n} @ r \ \mathbf{in} \ e_2 \Downarrow (h', k), v} \text{ [Let}_C\text{]} \\
 \frac{C = C_r \quad E \cup [\overline{x_{ri}} \mapsto \overline{v_i^{nr}}] \vdash (h, k), e_r \Downarrow (h', k), v}{E[x \mapsto p] \vdash (h[p \mapsto (j, C \overline{v_i^{nr}})], k), \mathbf{case} \ x \ \mathbf{of} \ \overline{C_i \overline{x_{ij}^{ni}} \rightarrow e_i^m} \Downarrow (h', k), v} \text{ [Case]} \\
 \frac{C = C_r \quad E \cup [\overline{x_{ri}} \mapsto \overline{v_i^{nr}}] \vdash (h, k), e_r \Downarrow (h', k), v}{E[x \mapsto p] \vdash (h \uplus [p \mapsto (j, C \overline{v_i^{nr}})], k), \mathbf{case!} \ x \ \mathbf{of} \ \overline{C_i \overline{x_{ij}^{ni}} \rightarrow e_i^m} \Downarrow (h', k), v} \text{ [Case!]}
 \end{array}$$

Fig. 2. Operational semantics of *Safe* expressions

values and region variables to actual region numbers in the range $\{0 \dots k\}$. We adopt the convention that for all E , if c is a constant, $E(c) = c$.

In a heap (h, k) , h is a finite mapping from pointers p to construction cells w of the form $(j, C \overline{v_i^n})$, where $j \leq k$, meaning that the cell resides in a region j in scope. By $h[p \mapsto w]$ we denote a heap mapping h where the binding $[p \mapsto w]$ exists and is highlighted, $h \uplus [p \mapsto w]$ denotes the disjoint union of h and $[p \mapsto w]$, and $(h \upharpoonright_k, k)$ is the heap obtained by deleting from (h, k') the bindings living in regions greater than k .

The semantics of a program $dec_1; \dots; dec_n; e$ is the semantics of the main expression e in an environment Σ containing all the function declarations. We only comment the rules related to allocation/deallocation, which may create dangling pointers in the heap. The rest are the usual ones for an eager language.

Rule *App* shows when a new region is allocated. The formal identifier *self* is bound to the newly created region $k+1$ so that the function body may create bindings in this region. Before returning, all cells created in region $k+1$ are deleted. This action is a source of possible dangling pointers. Rule *Case!* expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is another source of possible dangling pointers.

2.2 Safe Type System

We distinguish between functional and non-functional types. Non-functional algebraic types may be safe types (internally marked as s), condemned types (marked as d), or in-danger types (marked as r). In-danger types arise as an intermediate step during typing and are useful to control the side-effects of deconstructions, but function arguments can only receive either safe or condemned types. The intended semantics of these types is the following:

- **Safe types** (s): Data structures (DS) of this type can be read, copied or used to build other DSs. They cannot be destroyed.
- **Condemned types** (d): A DS directly involved in a **case!** action. Its recursive descendants inherit a condemned type. They cannot be used to build other DSs, but they can be read/copied before being destroyed.
- **In-danger types** (r): A DS sharing a recursive descendant of a condemned DS, so it can potentially contain dangling pointers.

Functional types can be polymorphic both in the Hindley-Milner sense and in the region sense: they may contain polymorphic type variables (denoted $\rho, \rho' \dots$) representing regions. If a region type variable occurs several times in a type, then the actual runtime regions of the corresponding arguments should be the same. Constructor applications have one region argument $r : \rho$ whose type occurs as the outermost region in the resulting algebraic type $T \bar{s} @ \bar{\rho}^m$ (i.e. $\rho_m = \rho$). Constructors are given types forcing its recursive substructures and the whole structure to live in the same region. For example, for lists and trees:

$$\begin{aligned}
[] &: \forall a \rho. \rho \rightarrow [a] @ \rho \\
(:) &: \forall a \rho. a \rightarrow [a] @ \rho \rightarrow \rho \rightarrow [a] @ \rho \\
\text{Empty} &: \forall a \rho. \rho \rightarrow \text{BSTree } a @ \rho \\
\text{Node} &: \forall a \rho. \text{BSTree } a @ \rho \rightarrow a \rightarrow \text{BSTree } a @ \rho \rightarrow \rho \rightarrow \text{BSTree } a @ \rho
\end{aligned}$$

Function types may have zero or more region arguments. For instance, the type inferred for `unshuffle` is:

$$\forall a \rho_1 \rho_2 \rho_3 \rho_4. [a]! @ \rho_4 \rightarrow \rho_1 \rightarrow \rho_2 \rightarrow \rho_3 \rightarrow ([a] @ \rho_1, [a] @ \rho_2) @ \rho_3$$

where $!$ is the external mark of a condemned type (internal mark d). Types without external marks are assumed to be safe.

The constructor types are collected in an environment Σ , easily built from the **data** type declarations. In typing environments Γ we can find region type assumptions $r : \rho$, variable type assumptions $x : t$, and polymorphic scheme assumptions for function symbols $f : \forall \bar{a} \forall \bar{\rho}. t$. The operators between typing environments used in the typing rules are shown in Fig. 3. The usual operator $+$ demands disjoint domains. Operators \otimes and \oplus are defined only if common variables have the same type, which must be safe in the case of \oplus . Operator \triangleright^L is an asymmetric composition used to type **let** expressions. Predicate $utype?(t, t')$ tells whether the underlying types (i.e. without marks) of t and t' are the same, while $unsafe?$ is true for types with a mark r or d .

In Fig. 4 we show the most relevant rules of the type system, which illustrate the use of the above environment operators. Function $sharerec(x, e)$ is the result of a sharing analysis and returns the set of free variables in the scope of expression e which at runtime may share a recursive descendant of variable x . An important consequence of having a sharing analysis is the unusual feature of our type system that the typing environment may contain type assumptions for variables which are not free in the typed expression.

Predicates inh and $inh!$ restrict the types of the **case/case!** patterns according to the type of the discriminant. The most important restriction is that the recursive patterns of a condemned discriminant must also be condemned. Predicate

Operator	$\Gamma_1 \bullet \Gamma_2$ defined if	Result of $(\Gamma_1 \bullet \Gamma_2)(x)$
+	$dom(\Gamma_1) \cap dom(\Gamma_2) = \emptyset$	$\Gamma_1(x)$ if $x \in dom(\Gamma_1)$ $\Gamma_2(x)$ otherwise
\otimes	$\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2) . \Gamma_1(x) = \Gamma_2(x)$	$\Gamma_1(x)$ if $x \in dom(\Gamma_1)$ $\Gamma_2(x)$ otherwise
\oplus	$\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2) . \Gamma_1(x) = \Gamma_2(x)$ $\wedge safe?(\Gamma_1(x))$	$\Gamma_1(x)$ if $x \in dom(\Gamma_1)$ $\Gamma_2(x)$ otherwise
\triangleright^L	$\forall x \in dom(\Gamma_1) \cap dom(\Gamma_2) . utype?(\Gamma_1(x), \Gamma_2(x))$ $\wedge \forall x \in dom(\Gamma_1) . unsafe?(\Gamma_1(x)) \rightarrow x \notin L$	$\Gamma_2(x)$ if $x \notin dom(\Gamma_1) \vee$ $x \in dom(\Gamma_1) \cap dom(\Gamma_2) \wedge safe?(\Gamma_1(x))$ $\Gamma_1(x)$ otherwise

Fig. 3. Operators on type environments

$$\begin{array}{c}
 \frac{\Gamma_1 \vdash e_1 : s_1 \quad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \quad utype?(\tau_1, s_1) \quad \neg danger?(\tau_1)}{\Gamma_1 \triangleright^{fv(e_2)} \Gamma_2 \vdash \text{let } x_1 = e_1 \text{ in } e_2 : s} \text{ [LET]} \\
 \\
 \frac{\begin{array}{c} \overline{t_i^n} \rightarrow \overline{p_j^l} \rightarrow T \ @\overline{p}^m \leq \sigma \quad \Gamma = [f : \sigma] + \bigoplus_{j=1}^l [r_j : \rho_j] + \bigoplus_{i=1}^n [a_i : t_i] \\ R = \bigcup_{i=1}^n \{sharerec(a_i, f \ \overline{a_i^n} @ \overline{\tau_j^l}) - \{a_i\} \mid cmd?(t_i)\} \quad \Gamma_R = \{y : danger(type(y)) \mid y \in R\} \end{array}}{\Gamma_R + \Gamma \vdash f \ \overline{a_i^n} @ \overline{\tau_j^l} : T \ @\overline{p}^m} \text{ [APP]} \\
 \\
 \frac{\begin{array}{c} \forall i \in \{1..n\}. \Sigma(C_i) = \sigma_i \quad \forall i \in \{1..n\}. \overline{s_i^{n_i}} \rightarrow \rho \rightarrow T \ @\rho \leq \sigma_i \\ \Gamma \geq_{\text{case}} x \text{ of } \overline{C_i \ \overline{x_{ij}^{n_i}} \rightarrow e_i^n} [x : T @ \rho] \quad \forall i \in \{1..n\}. \forall j \in \{1..n_i\}. inh(\tau_{ij}, s_{ij}, \Gamma(x)) \\ \forall i \in \{1..n\}. \Gamma + [\overline{x_{ij} : \tau_{ij}}]^{n_i} \vdash e_i : s \end{array}}{\Gamma \vdash \text{case } x \text{ of } \overline{C_i \ \overline{x_{ij}^{n_i}} \rightarrow e_i^n} : s} \text{ [CASE]} \\
 \\
 \frac{\begin{array}{c} (\forall i \in \{1..n\}). \Sigma(C_i) = \sigma_i \quad \forall i \in \{1..n\}. \overline{s_i^{n_i}} \rightarrow \rho \rightarrow T \ @\rho \leq \sigma_i \\ R = sharerec(x, \text{case! } x \text{ of } \overline{C_i \ \overline{x_{ij}^{n_i}} \rightarrow e_i^n}) \quad \forall i \in \{1..n\}. \forall j \in \{1..n_i\}. inh!(t_{ij}, s_{ij}, T ! @ \rho) \\ \forall z \in R, i \in \{1..n\}. z \notin fv(e_i) \quad \forall i \in \{1..n\}. \Gamma + [x : T \ # @ \rho] + [\overline{x_{ij} : t_{ij}}]^{n_i} \vdash e_i : s \\ \Gamma_R = \{y : danger(type(y)) \mid y \in R - \{x\}\} \end{array}}{\Gamma_R \otimes \Gamma + [x : T ! @ \rho] \vdash \text{case! } x \text{ of } \overline{C_i \ \overline{x_{ij}^{n_i}} \rightarrow e_i^n} : s} \text{ [CASE!]}
 \end{array}$$

Fig. 4. Some *Safe* typing rules for expressions

danger? is true for r -marked types, while function *danger*(t) attaches a mark r to a safe type t . For a complete description, see [11]. An inference algorithm for this type system has been developed in [12].

3 Cell Deallocation by Destructive Pattern Matching

The idea of the certificate is to ask the compiler to deliver some static information inferred during the type inference phase, and then to use a database of previously proved lemmas relating this information with the dynamic properties the program is expected to satisfy at runtime. In this case, the static information consists of a mark $m \in \{s, r, d\}$ —respectively meaning *safe*, *in-danger*, and *condemned* type— for every variable, and the dynamic property the certificate must prove is that the heap remains closed during evaluation.

By $fv(e)$ we denote the set of free variables of expression e , excluding function names and region variables, and by $dom(h)$ the set $\{p \mid [p \mapsto w] \in h\}$. A *static assertion* has the form $\llbracket L, \Gamma \rrbracket$, where $L \subseteq dom(\Gamma)$ is a set of program

variables and Γ a mark environment assigning a mark to each variable in L and possibly to some other variables. We will write $\Gamma[x] = m$ to indicate that x has mark $m \in \{s, r, d\}$ in Γ . We say that a *Core-Safe* expression e satisfies a static assertion, denoted $e : \llbracket L, \Gamma \rrbracket$, if $fv(e) \subseteq L$ and some semantic conditions below hold. Our certificate for a given program consists of proving a static assertion $\llbracket L, \Gamma \rrbracket$ for each *Core-Safe* expression e resulting from compiling the program.

If E is the runtime environment, the intuitive idea of a variable x being typed with a safe mark s is that all the cells in the heap h reached at runtime by $E(x)$ do not contain dangling pointers and they are disjoint from unsafe cells. The idea behind a condemned variable x is that the cell pointed to by $E(x)$ will be removed from the heap and all live cells reaching any of $E(x)$'s recursive descendants by following a pointer chain are in danger. We use the following definitions, formally specified in Isabelle/HOL:

$closure(E, X, h)$	Set of locations reachable in heap h by $\{E(x) \mid x \in X\}$
$closure(v, h)$	Set of locations reachable in h by location v
$recReach(E, x, h)$	Set of recursive descendants of $E(x)$ including itself
$recReach(v, h)$	Set of recursive descendants of v in h including itself
$closed(E, L, h)$	There are no dangling pointers in h , i.e. $closure(E, L, h) \subseteq dom(h)$
$p \rightarrow_h^* V$	There is a pointer path in h from p to a $q \in V$

By abuse of notation, we will write $closure(E, x, h)$ and also $closed(v, h)$. Now, we define the following two sets, respectively denoting the safe and unsafe locations of the live heap, as functions of L, Γ, E , and h :

$$S_{L, \Gamma, E, h} \stackrel{\text{def}}{=} \bigcup_{x \in L, \Gamma[x]=s} \{closure(E, x, h)\}$$

$$R_{L, \Gamma, E, h} \stackrel{\text{def}}{=} \bigcup_{x \in L, \Gamma[x]=d} \{p \in closure(E, L, h) \mid p \rightarrow_h^* recReach(E, x, h)\}$$

Definition 1. *Given the following properties*

$$P1 \equiv E \vdash (h, k), e \Downarrow (h', k), v$$

$$P2 \equiv dom(\Gamma) \subseteq dom(E)$$

$$P3 \equiv L \subseteq dom(\Gamma)$$

$$P4 \equiv fv(e) \subseteq L$$

$$P5 \equiv \forall x \in dom(E). \forall z \in L.$$

$$\Gamma[z] = d \wedge recReach(E, z, h) \cap closure(E, x, h) \neq \emptyset \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$$

$$P6 \equiv \forall x \in dom(E). closure(E, x, h) \neq closure(E, x, h') \rightarrow x \in dom(\Gamma) \wedge \Gamma[x] \neq s$$

$$P7 \equiv S_{L, \Gamma, E, h} \cap R_{L, \Gamma, E, h} = \emptyset$$

$$P8 \equiv closed(E, L, h)$$

$$P9 \equiv closed(v, h')$$

we say that expression e satisfies the static assertion $\llbracket L, \Gamma \rrbracket$, denoted $e : \llbracket L, \Gamma \rrbracket$, if $P3 \wedge P4 \wedge (\forall E \ h \ k \ h' \ v. P1 \wedge P2 \rightarrow P5 \wedge P6 \wedge (P7 \wedge P8 \rightarrow P9))$.

A notion of satisfaction relative to the validity of a function environment Σ_M , denoted $e, \Sigma_M : \llbracket L, \Gamma \rrbracket$, is also defined.

Property $P1$ defines any runtime evaluation of e . Properties $P2$ to $P4$ just guarantee that each free variable has a type and a value. Properties $P5$ to $P7$ formalise the dynamic meaning of safe and condemned types: if some variable

$$\begin{array}{c}
 c, \Sigma_M \vdash (\emptyset, \emptyset) \text{ LIT} \quad x, \Sigma_M \vdash (\{x\}, \Gamma + [x : s]) \text{ VAR1} \\
 \hline
 e_1 \neq C \overline{a_i^n} \quad e_1, \Sigma_M \vdash (L_1, \Gamma_1) \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : s]) \quad \text{def}(\Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 \text{let } x_1 = e_1 \text{ in } e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 e_1 \neq C \overline{a_i^n} \quad e_1, \Sigma_M \vdash (L_1, \Gamma_1) \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : d]) \quad \text{def}(\Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 \text{let } x_1 = e_1 \text{ in } e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 L_1 = \{\overline{a_i^n}\} \quad \Gamma_1 = [\overline{a_i} \mapsto s^n] \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2' + [x_1 : s]) \quad \text{def}(\Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 \text{let } x_1 = C \overline{a_i^n} @_r \text{ in } e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 L_1 = \{\overline{a_i^n}\} \quad \Gamma_1 = [\overline{a_i} \mapsto s^n] \quad x_1 \notin L_1 \quad e_2, \Sigma_M \vdash (L_2, \Gamma_2 + [x_1 : d]) \quad \text{def}(\Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 \text{let } x_1 = C \overline{a_i^n} @_r \text{ in } e_2, \Sigma_M \vdash (L_1 \cup (L_2 - \{x_1\}), \Gamma_1 \triangleright^{L_2} \Gamma_2') \\
 \hline
 \forall i. (e_i, \Sigma_M \vdash (L_i, \Gamma_i) \quad \forall j. \Gamma_i[x_{ij}] \neq d) \quad \Gamma \supseteq \bigotimes_i (\Gamma_i \setminus \{\overline{x_{ij}}\}) \quad x \in \text{dom}(\Gamma) \quad L = \{x\} \cup (\bigcup_i (L_i - \{\overline{x_{ij}}\})) \\
 \hline
 \text{case } x \text{ of } \overline{C_i \overline{x_{ij}}} \rightarrow e_i, \Sigma_M \vdash (L, \Gamma) \\
 \hline
 \forall i. (e_i, \Sigma_M \vdash (L_i, \Gamma_i) \quad \forall j. \Gamma_i[x_{ij}] = d \rightarrow j \in \text{RecPos}(C_i)) \\
 L' = \bigcup_i (L_i - \{\overline{x_{ij}}\}) \quad \Gamma \supseteq (\bigotimes_i \Gamma_i \setminus \{\{\overline{x_{ij}}\} \cup \{x\}\}) + [x : d] \quad \forall z \in \text{dom}(\Gamma). \Gamma[z] \neq s \rightarrow (\forall i. z \notin L_i) \\
 \hline
 \text{case! } x \text{ of } \overline{C_i \overline{x_{ij}}} \rightarrow e_i, \Sigma_M \vdash (L' \cup \{x\}, \Gamma) \\
 \hline
 \Sigma_M(g) = \overline{m_i^n} \quad L = \{\overline{x_i^n}\} \quad \Gamma_0 = \bigoplus_{i=1}^n [a_i : m_i] \text{ defined} \quad \Gamma \supseteq \Gamma_0 \\
 \hline
 g \overline{a_i^n} @ r_j^m, \Sigma_M \vdash (L, \Gamma) \\
 \hline
 f \overline{x_i^n} @ \overline{r_j^m} = e_f \quad L_f = \{\overline{x_i^n}\} \quad \Gamma_f = [\overline{x_i} \mapsto \overline{m_i^n}] \quad e_f, \Sigma_M \uplus [f \mapsto \overline{m_i^n}] \vdash (L_f, \Gamma_f) \\
 \hline
 e_f, \Sigma_M \vdash (L_f, \Gamma_f)
 \end{array}$$

Fig. 5. Proof rules for explicit deallocation

can share a recursive descendant of a condemned one, or its closure changes during evaluation, it should occur as unsafe in the environment.

The key properties are *P8* and *P9*. If they were proved for all the judgements of any e 's derivation, they would guarantee that the live part of the heap would remain closed, hence there would not be dangling pointers. We have proved that *P8* is an ‘upwards’ invariant in any derivation, while *P9* is a ‘downwards’ one. Formally:

Theorem 1 (closedness). *Consider a derivation $E \vdash (h, k), e \Downarrow (h', k), v$. If $e : \llbracket L, \Gamma \rrbracket$, $P2(\Gamma, E)$, $P7(L, \Gamma, E, h)$ and $P8(E, L, h)$ hold, then $P8(E_i, L_i, h_i)$ and $P9(v_i, h'_i)$ hold for all judgements $E_i \vdash (h_i, k_i), e_i \Downarrow (h'_i, k_i), v_i$ belonging to that derivation.*

But *P2*, *P7* and *P8* trivially hold for the empty heap, empty environment Γ , and empty set L of free variables, which are the ones corresponding to the initial expression, so *P8* and *P9* hold across the whole derivation of the program.

In Fig. 5 we show the proof rules related to this property. The following soundness theorem (a lemma for each expression) has been interactively proved by induction on the derivations obtained with these rules.

Theorem 2 (soundness). *If $e, \Sigma_M \vdash (L, \Gamma)$ then $e, \Sigma_M : \llbracket L, \Gamma \rrbracket$.*

When proving the soundness of the *APP* rule, the closedness of the heap before returning from g does not in principle guarantee that the heap will remain closed after deallocating the heap topmost region. Proving this, needs a separate

collection of theorems showing that the value returned by g does not contain cells in that region. This part of the problem is explained in Sec. 4.

For each expression e , the compiler generates a pair (L, Γ) . According to e 's syntax, the certificate chooses the appropriate proof rule, checks that its premises are satisfied, and applies it in order to get the conclusion e , $\Sigma_M \vdash (L, \Gamma)$. For example, in an application expression $g \overline{a}_i^n @ \overline{r}_j^m$, the certificate access to Σ_M and checks that the given environment Γ contains the actual arguments a_i with these marks m_i assigned. It also checks that operator \oplus , requiring any duplicated actual argument to be safe (see Fig. 3), is well-defined, and then it applies the proof rule *APP*.

4 Region Deallocation

We present here the proof rules certifying that region deallocation does not create dangling pointers. As before, the compiler delivers static information about the region types used by the program variables and expressions, and a soundness theorem relates this information to the runtime properties of the actual regions.

In an algebraic type $T \overline{t}_i^m @ \overline{\rho}_j^l$, the last region type variable ρ_l of the list is always the most external one, i.e. the region where the cell of the most external constructor is allocated. By *regions* (t) we denote the set of region type variables occurring in the type t . There is a reserved identifier ρ_{self}^f for every defined function f , denoting the region type variable assigned to the working region *self* of function f . We will assume that the expression e being certified belongs to the body of a context function f or to the *main* expression.

By θ, θ_i, \dots we denote *typing environments*, i.e. mappings from program variables and region arguments to types. For region arguments, $\theta(r) = \rho$ means that ρ is the type variable the compiler assigns to argument r .

In function or constructor applications, the set of generic region types used in the signature of an applied function g (of a constructor C) must be related to the actual region types used in the application. Also, some ordinary polymorphic type variables of the signature may become instantiated by algebraic types introducing additional regions. Let us denote by μ the *type instantiation mapping* used by the compiler. This mapping should correctly map the types of the formal arguments to the types of the corresponding actual arguments.

Definition 2. *Given the instantiated types \overline{t}_i^n , the instantiated region types $\overline{\rho}_j^m$, the arguments of the application \overline{a}_i^n , \overline{r}_j^m , and the typing mapping θ , we say that the application is argument preserving, denoted $argP(\overline{t}_i^n, \overline{\rho}_j^m, \overline{a}_i^n, \overline{r}_j^m, \theta)$, if: $\forall i \in \{1..n\} . t_i = \theta(a_i) \wedge \forall j \in \{1..m\} . \rho_j = \theta(r_j)$.*

For functions, the certificate incrementally constructs a global environment Σ_T keeping the most general types of the functions already certified. For constructors, the compiler provides a global environment Γ_T giving its polymorphic most general type. If $\Gamma_T(C) = \overline{t}_i^n \rightarrow \rho \rightarrow T \overline{t}_j^l @ \overline{\rho}_i^m$, the following property, satisfied by the type system, is needed for proving the proof rules below:

Definition 3. Predicate $\text{wellT}(\overline{t}_i^n, \rho, T \overline{t}_j^l @ \overline{\rho}_i^m)$, read *well-typed*, is defined as $\rho_m = \rho \wedge \bigcup_{i=1}^n \text{regions}(t_i) \subseteq \text{regions}(T \overline{t}_j^l @ \overline{\rho}_i^m)$

So far for the static concepts. We move now to the dynamic or runtime ones. By η, η_i, \dots we denote *region instantiation mappings* from region type variables to runtime regions identifiers in scope. Region identifiers k, k_i, \dots are just natural numbers denoting offsets of the actual regions from the bottom of the region stack. If k is the topmost region in scope, then for all ρ , $0 \leq \eta(\rho) \leq k$ holds. The intended meaning of $k' = \eta(\rho)$ is that, in a particular execution of the program, the region type ρ has been instantiated to the actual region k' . Admissible region instantiation mappings should map ρ_{self}^f to the topmost region, and other region types to lower regions.

Definition 4. Assuming that k denotes the topmost region of a given heap, we say that the mapping η is *admissible*, denoted *admissible* (η, k) , if:

$$\rho_{\text{self}}^f \in \text{dom}(\eta) \wedge \eta(\rho_{\text{self}}^f) = k \wedge \forall \rho \in \text{dom}(\eta) - \{\rho_{\text{self}}^f\} . \eta(\rho) < k$$

The important notion is *consistency* between the static information θ and the dynamic one E, η, h, h' . Essentially, it tells us that the static region types, its runtime instantiation to actual regions, and the actual regions where the data structures are stored in the heap, do not contradict each other.

Definition 5. We say that the mappings θ, η , the runtime environment E , and the heap h are *consistent*, denoted *consistent* (θ, η, E, h) , if:

1. $\forall x \in \text{dom}(E) . \text{consistent}(\theta(x), \eta, E(x), h)$ where:

$$\begin{aligned} \text{consistent}(B, \eta, c, h) &= \text{true} && \text{-- } B \text{ denotes a basic type} \\ \text{consistent}(a, \eta, v, h) &= \text{true} && \text{-- } a \text{ denotes a type variable} \\ \text{consistent}(T \overline{t}_i^m @ \overline{\rho}_j^l, \eta, p, h) &= \exists j C \overline{v}_k^n \mu \overline{t}_{kC}^n \overline{\rho}_{jC}^l . h(p) = (j, C \overline{v}_k^n) \\ &\wedge \rho_l \in \text{dom}(\eta) \wedge \eta(\rho_l) = j \\ &\wedge \Gamma_T(C) = \overline{t}_{kC}^n \rightarrow \rho_{lC} \rightarrow T \overline{t}_{iC}^m @ \overline{\rho}_{jC}^l \\ &\wedge \mu(T \overline{t}_{iC}^m @ \overline{\rho}_{jC}^l) = T \overline{t}_i^m @ \overline{\rho}_j^l \\ &\wedge \forall k \in \{1..n\} . \text{consistent}(\mu(\overline{t}_{kC}), \eta, v_k, h) \end{aligned}$$

2. $\forall r \in \text{dom}(E) . \theta(r) \in \text{dom}(\eta) \wedge E(r) = (\eta \cdot \theta)(r)$

3. $\text{self} \in \text{dom}(E) \wedge \theta(\text{self}) = \rho_{\text{self}}^f$

We are ready to define the satisfaction of a *static assertion* relating the static and dynamic properties referred to regions: A judgement of the form $e : [\theta, t]$ defines that, if expression e is evaluated with an environment E , a heap (h, k) , and an admissible mapping η consistent with θ , then η , the final heap h' , and the final value v are consistent with t . Formally:

Definition 6. An expression e satisfies the pair (θ, t) , denoted $e : [\theta, t]$ if

$$\begin{aligned} \forall E h k h' v \eta . E \vdash (h, k), e \Downarrow (h', k), v &\text{ -- } P1 \\ \wedge \text{dom}(E) \subseteq \text{dom}(\theta) &\text{ -- } P2 \\ \wedge \text{admissible}(\eta, k) &\text{ -- } P3 \\ \wedge \text{consistent}(\theta, \eta, E, h) &\text{ -- } P4 \\ \rightarrow \text{consistent}(t, \eta, v, h') &\text{ -- } P5 \end{aligned}$$

$$\begin{array}{c}
\frac{c, \Sigma_T \vdash \theta \rightsquigarrow B}{LIT} \quad \frac{x, \Sigma_T \vdash \theta \rightsquigarrow \theta(x)}{VAR1} \quad \frac{e_1, \Sigma_T \vdash \theta \rightsquigarrow t_1 \quad e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto t_1] \rightsquigarrow t_2}{\mathbf{let} \ x_1 = e_1 \ \mathbf{in} \ e_2, \Sigma_T \vdash \theta \rightsquigarrow t_2} LET \\
\frac{\Gamma_T(C) = \overline{t_i^n} \rightarrow \rho \rightarrow t \ \mathit{wellT}(\overline{t_i^n}, \rho, t) \quad e_2, \Sigma_T \vdash \theta \uplus [x_1 \mapsto \mu(t)] \rightsquigarrow t_2 \quad \mathit{argP}(\overline{\mu(t_i)^n}, \mu(\rho), \overline{a_i^n}, r, \theta)}{\mathbf{let} \ x_1 = C \ \overline{a_i^n} \ @ \ r \ \mathbf{in} \ e_2, \Sigma_T \vdash \theta \rightsquigarrow t_2} LET_C \\
\frac{\forall i. (\Gamma_T(C_i) = \overline{t_{ij}^{n_i}} \rightarrow \rho \rightarrow t \ \mathit{wellT}(\overline{t_{ij}^{n_i}}, \rho, t)) \quad \forall i. e_i, \Sigma_T \vdash \theta \uplus [x_{ij} \rightarrow \overline{\mu(t_{ij}^{n_i})}] \rightsquigarrow t' \quad \theta(x) = \mu(t)}{\mathbf{case} \ x \ \mathbf{of} \ \overline{C_i} \ \overline{x_{ij}^{n_i}} \rightarrow e_i^n, \Sigma_T \vdash \theta \rightsquigarrow t'} CASE \\
\frac{\forall i. (\Gamma_T(C_i) = \overline{t_{ij}^{n_i}} \rightarrow \rho \rightarrow t \ \mathit{wellT}(\overline{t_{ij}^{n_i}}, \rho, t)) \quad \forall i. e_i, \Sigma_T \vdash \theta \uplus [x_{ij} \rightarrow \overline{\mu(t_{ij}^{n_i})}] \rightsquigarrow t' \quad \theta(x) = \mu(t)}{\mathbf{case!} \ x \ \mathbf{of} \ \overline{C_i} \ \overline{x_{ij}^{n_i}} \rightarrow e_i^n, \Sigma_T \vdash \theta \rightsquigarrow t'} CASE! \\
\frac{\Sigma(g) = \overline{t_i^n} \rightarrow \overline{\rho_j^m} \rightarrow t_g \quad \rho_{self}^g \notin \mathit{regions}(t_g) \quad \mathit{argP}(\overline{\mu(t_i)^n}, \overline{\mu(\rho_j)^m}, \overline{a_i^n}, \overline{r_j^m}, \theta) \quad t = \mu(t_g)}{g \ \overline{a_i^n} \ @ \ \overline{r_j^m}, \Sigma_T \vdash \theta \rightsquigarrow t} APP \\
\frac{f \ \overline{x_i^n} \ @ \ \overline{r_j^m} = e_f \quad \theta_f = [x_i \mapsto \overline{t_i^n}, \overline{r_j} \mapsto \overline{\rho_j^m}, \mathit{self} \mapsto \rho_{self}] \quad e_f, \Sigma_T \cup \{f \mapsto \overline{t_i^n} \rightarrow \overline{\rho_j^m} \rightarrow t_f\} \vdash \theta_f \rightsquigarrow t_f}{e_f, \Sigma_T \vdash \theta_f \rightsquigarrow t_f} REC
\end{array}$$

Fig. 6. Proof rules for region deallocation

Theorem 3 (consistency). *If $e : \llbracket \theta, t \rrbracket$, $E \vdash (h, k)$, $e \Downarrow (h', k)$, v , $P2(E, \theta)$, $P3(\eta, k)$, $P4(\theta, \eta, E, h)$ hold, then $P3(\eta_i, k_i)$, $P4(\theta_i, \eta_i, E_i, h_i)$, $P5(t_i, \eta_i, v_i, h'_i)$ hold for all judgements $E_i \vdash (h_i, k_i)$, $e_i \Downarrow (h'_i, k_i)$, v_i belonging to that derivation.*

But $P2$, $P3$ and $P4$ trivially hold for the empty heap h_0 , $\mathit{dom}(E_0) = \mathit{dom}(\theta_0) = \{\mathit{self}\}$, $\theta_0(\mathit{self}) = \rho_{self}^{main}$, $k_0 = 0$, and $\eta_0(\rho_{self}^{main}) = E_0(\mathit{self}) = 0$, which are the ones corresponding to the initial expression, so $P3$, $P4$ and $P5$ hold across the whole program derivation. In Fig. 6 we show the proof rules related to regions.

Theorem 4 (soundness). *If $e, \Sigma_T \vdash \theta \rightsquigarrow t$ then $e, \Sigma_T : \llbracket \theta, t \rrbracket$.*

To prove it, there is a separate Isabelle/HOL theorem for each syntactic form. As we have said, region allocation/deallocation takes place at function call/return. The premise $\rho_{self}^g \notin \mathit{regions}(t_g)$ in the APP rule, together with properties $P3$ and $P5$ guarantee that the data structure returned by the function has no cells in the deallocated region corresponding to $\eta(\rho_{self}^g)$. So, deallocating this region cannot cause dangling pointers.

For each expression e , the compiler generates a pair (θ, t) (and a μ when needed). According to e 's syntax, the certificate applies the corresponding proof rule by previously discharging its premises, then deriving $e \vdash \theta \rightsquigarrow t$. For example, in an application expression, we assume that the most general type of the called function g is kept in the global environment Σ_T . The certificate receives the (θ, t, μ) for this particular application, gets g 's signature from Σ_T , checks $t = \mu(t_g)$ and the rest of premises of the APP proof rule, and then applies it.

5 Certificate Generation

Given the above sets of already proved theorems, certificate generation for a given program is a rather straightforward task. It consists of traversing the program's abstract syntax tree and producing the following information:

	Expression	L	Γ
e_1	$\stackrel{\text{def}}{=} \text{unshuffle } x_{50} @ r_2 r_1 \text{ self}$	$\{x_{50}\}$	$[x_{50} : d, x_{34} : r]$
e_2	$\stackrel{\text{def}}{=} x_{45}$	$\{x_{45}\}$	$[x_{45} : s, x_{34} : r]$
e_3	$\stackrel{\text{def}}{=} \text{case } x_{40} \text{ of } (x_{45}, x_{46}) \rightarrow e_2$	$\{x_{40}\}$	$[x_{40} : s, x_{34} : r]$
e_4	$\stackrel{\text{def}}{=} x_{48}$	$\{x_{48}\}$	$[x_{48} : s, x_{34} : r]$
e_5	$\stackrel{\text{def}}{=} \text{case } x_{40} \text{ of } (x_{47}, x_{48}) \rightarrow e_4$	$\{x_{40}\}$	$[x_{40} : s, x_{34} : r]$
e_6	$\stackrel{\text{def}}{=} x_{39}$	$\{x_{39}\}$	$[x_{39} : s, x_{34} : r]$
e_7	$\stackrel{\text{def}}{=} \text{let } x_{39} = (x_{38}, x_{15}) @ r_3 \text{ in } e_6$	$\{x_{15}, x_{38}\}$	$[x_{15} : s, x_{38} : s, x_{34} : r]$
e_8	$\stackrel{\text{def}}{=} \text{let } x_{38} = x_{49} : x_{16} @ r_1 \text{ in } e_7$	$\{x_{15}, x_{16}, x_{49}\}$	$[x_{15} : s, x_{16} : s, x_{49} : s, x_{34} : r]$
e_9	$\stackrel{\text{def}}{=} \text{let } x_{16} = e_5 \text{ in } e_4$	$\{x_{15}, x_{40}, x_{49}\}$	$[x_{15} : s, x_{40} : s, x_{49} : s, x_{34} : r]$
e_{10}	$\stackrel{\text{def}}{=} \text{let } x_{15} = e_3 \text{ in } e_2$	$\{x_{40}, x_{49}\}$	$[x_{40} : s, x_{49} : s, x_{34} : r]$
e_{11}	$\stackrel{\text{def}}{=} \text{let } x_{40} = e_1 \text{ in } e_{10}$	$\{x_{49}, x_{50}\}$	$[x_{49} : s, x_{50} : d, x_{34} : r]$
e_{12}	$\stackrel{\text{def}}{=} x_{37}$	$\{x_{37}\}$	$[x_{37} : s, x_{34} : r]$
e_{13}	$\stackrel{\text{def}}{=} \text{let } x_{37} = (x_{36}, x_{35}) @ r_3 \text{ in } e_{12}$	$\{x_{35}, x_{36}\}$	$[x_{35} : s, x_{36} : s, x_{34} : r]$
e_{14}	$\stackrel{\text{def}}{=} \text{let } x_{35} = [] @ r_2 \text{ in } e_{13}$	$\{x_{36}\}$	$[x_{36} : s, x_{34} : r]$
e_{15}	$\stackrel{\text{def}}{=} \text{let } x_{36} = [] @ r_1 \text{ in } e_{14}$	$\{\}$	$[x_{34} : r]$
e_{16}	$\stackrel{\text{def}}{=} \text{case! } x_{34} \text{ of } \{x_{49} : x_{50} \rightarrow e_{11}; [] \rightarrow e_{15}\}$	$\{x_{34}\}$	$[x_{34} : d]$

Fig. 7. Isabelle/HOL definitions of *Core-Safe* expressions, free variables, and mark environments for `unshuffle`

- A definition in Isabelle/HOL of the abstract syntax tree.
- A set of Isabelle/HOL definitions for the static objects inferred by the analyses: sets of free variables, mark environments, typing environments, type instantiation mappings, etc.
- A set of Isabelle/HOL proof scripts proving a lemma for each expression, consisting of first checking the premises of the proof-rule associated to the syntactic form of the expression, and then applying the proof rule.

This strategy results in small certificates and short checking times as the total amount of work is linear with program size. The heaviest part of the proof—the database of proved proof rules—has been done in advance and is reused by each certified program.

In Fig. 7 we show the Isabelle/HOL definitions for the elementary *Core-Safe* expressions of the `unshuffle` function defined in Sec. 2 together with the components L and Γ of the static assertions proving the absence of dangling pointers for cell deallocation. They are arranged bottom-up, from simple to compound expressions, because this is the order required by Isabelle/HOL for applying the proof rules. In Fig. 8 we show (this time top-down for a better understanding) the components θ , t , and μ of the static assertions for region deallocation for the expressions e_{14} , e_{15} , and e_{16} of Fig. 7. We show also the most general types of some constructors given by the global environment Γ_T .

The *Core-Safe* text for `unshuffle` consists of about 50 lines, while the certificate for it is about 1000 lines long, 300 of which are devoted to definitions. This expansion factor of 20 is approximately the same for all the examples we have certified so far, so confirming that certificate size grows linearly with program

$$\begin{array}{ll}
\theta_{16} \stackrel{\text{def}}{=} [x_{34} : [a]@_{\rho_4}, r_1 : \rho_1, r_2 : \rho_2, r_3 : \rho_3, self : \rho_{self}] & t_{16} \stackrel{\text{def}}{=} ([a]@_{\rho_1}, [a]@_{\rho_2})@_{\rho_3} \\
\theta_{15} \stackrel{\text{def}}{=} \theta_{16} & t_{15} \stackrel{\text{def}}{=} ([a]@_{\rho_1}, [a]@_{\rho_2})@_{\rho_3} \\
\theta_{14} \stackrel{\text{def}}{=} \theta_{15} + [x_{36} : [a]@_{\rho_1}] & t_{14} \stackrel{\text{def}}{=} ([a]@_{\rho_1}, [a]@_{\rho_2})@_{\rho_3} \\
\mu_{16} \stackrel{\text{def}}{=} \{a \mapsto a, \rho_1 \mapsto \rho_4\} & \Gamma_T([\] = \rho_1 \rightarrow [a]@_{\rho_1} \\
\mu_{15} \stackrel{\text{def}}{=} \{a \mapsto a, \rho_1 \mapsto \rho_1\} & \Gamma_T(\cdot) = a \rightarrow [a]@_{\rho_1} \rightarrow \rho_1 \rightarrow [a]@_{\rho_1} \\
\mu_{14} \stackrel{\text{def}}{=} \{a \mapsto a, \rho_1 \mapsto \rho_2\} &
\end{array}$$

Fig. 8. Isabelle/HOL definitions of typing mappings, and types for `unshuffle`

size. There is room for optimisation by defining an Isabelle/HOL tactic for each proof rule. This reduces both the size and the checking time of the certificate. We have implemented this idea in the region deallocation part.

The Isabelle/HOL proof scripts for the cell deallocation proof-rules reach 8 000 lines, while the ones devoted to region deallocation tally up to 4 000 lines more. Together they represent about 1.5 person-year effort. All the theories are available at <http://dalila.sip.ucm.es/safe/certifdangling>. There is also an on-line version of the *Safe* compiler at <http://dalila.sip.ucm.es/~safe> where users may remotely submit source files and browse all the generated intermediate files, including certificates. An extended version of this paper with proof schemes available can be found at <http://dalila.sip.ucm.es/safe>.

6 Related Work and Conclusion

Introducing pointers in a Hoare-style assertion logic and using a proof assistant for proving pointer programs goes back to the late seventies [9], where the Stanford Pascal Program Verifier was used. A more recent reference is [5], using the Jape proof editor. A formalisation of Bornat’s ideas in Isabelle/HOL was done by Mehta and Nipkow in [10], where they add a complete soundness proof.

A type system allowing safe heap destruction was studied in [1] and [2]. In [11] we made a detailed comparison with those works showing that our system accepts as safe some programs that their system rejects. Another difference is that we have developed a type inference algorithm [12] which they lack.

Connecting the results of a static analysis with the generation of certificates was done from the beginning of the PCC paradigm (see for instance [15]). A more recent work is [3].

Our work is more closely related to [4], where a resource consumption property obtained by a special type system developed in [8] is transformed into a certificate. The compiler is able to infer a linear upper bound on heap consumption and to certify this property by emitting an Isabelle/HOL script proving it. Our static assertions have been inspired by their *derived assertions*, used also there to connect static with dynamic properties. However, their heap is simpler to deal with than ours since it essentially consists of a free list of cells, and the only data type available is the list. We must also deal with regions and with any user-defined data type. This results in our complex notion of consistency.

Apart from the proofs themselves, our contribution has been defining the appropriate functions, predicates and relations such as *closure*, *recReach*, *closed*, *consistent*, . . . relating the static and the runtime information in such a way that the proof-rules could be proved correct.

References

1. Aspinall, D., Hofmann, M.: Another Type System for In-Place Update. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 36–52. Springer, Heidelberg (2002)
2. Aspinall, D., Hofmann, M., Konečný, M.: A Type System with Usage Aspects. *Journal of Functional Programming* 18(2), 141–178 (2008)
3. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate Translation for Optimizing Compilers. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 301–317. Springer, Heidelberg (2006)
4. Beringer, L., Hofmann, M., Momigliano, A., Shkaravska, O.: Automatic Certification of Heap Consumption. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 347–362. Springer, Heidelberg (2005)
5. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
6. de Dios, J., Peña, R.: A Certified Implementation on top of the Java Virtual Machine. In: Alpuente, M. (ed.) FMICS 2009. LNCS, vol. 5825, pp. 181–196. Springer, Heidelberg (2009)
7. de Dios, J., Peña, R.: Formal Certification of a Resource-Aware Language Implementation. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOL 2009. LNCS, vol. 5674, pp. 196–211. Springer, Heidelberg (2009)
8. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Proc. 30th ACM Symp. on Principles of Programming Languages, POPL 2003, pp. 185–197. ACM Press, New York (2003)
9. Luckham, D.C., Suzuki, N.: Verification of array, record and pointer operations in Pascal. *ACM Trans. on Prog. Lang. and Systems* 1(2), 226–244 (1979)
10. Mehta, F., Nipkow, T.: Proving Pointer Programs in Higher-Order Logic. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 121–135. Springer, Heidelberg (2003)
11. Montenegro, M., Peña, R., Segura, C.: A Type System for Safe Memory Management and its Proof of Correctness. In: ACM Principles and Practice of Declarative Programming, PPDP 2008, Valencia, Spain, pp. 152–162 (July 2008)
12. Montenegro, M., Peña, R., Segura, C.: An Inference Algorithm for Guaranteeing Safe Destruction. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 135–151. Springer, Heidelberg (2009)
13. Montenegro, M., Peña, R., Segura, C.: A simple region inference algorithm for a first-order functional language. In: Escobar, S. (ed.) WFLP 2009. LNCS, vol. 5979, pp. 145–161. Springer, Heidelberg (2010)
14. Nacula, G.C.: Proof-Carrying Code. In: ACM SIGPLAN-SIGACT Principles of Programming Languages, POPL 1997, pp. 106–119. ACM Press, New York (1997)
15. Nacula, G.C., Lee, P.: Safe Kernel Extensions Without Run-Time Checking. In: Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, Washington, pp. 229–243 (October 1996)
16. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)

Integrating Implicit Induction Proofs into Certified Proof Environments

Sorin Stratulat

LITA, Université Paul Verlaine-Metz, 57000, France
LORIA, 54000, France
stratulat@univ-metz.fr

Abstract. We give evidence of the direct integration and automated checking of implicit induction-based proofs inside certified reasoning environments, as that provided by the Coq proof assistant. This is the first step of a long term project focused on 1) mechanically certifying implicit induction proofs generated by automated provers like Spike, and 2) narrowing the gap between automated and interactive proof techniques inside proof assistants such that multiple induction steps can be executed completely automatically and mutual induction can be treated more conveniently. Contrary to the current approaches of reconstructing implicit induction proofs into scripts based on explicit induction tactics that integrate the usual proof assistants, our checking methodology is simpler and fits better for automation. The underlying implicit induction principles are separated and validated independently from the proof scripts that consist in a bunch of one-to-one translations of implicit induction proof steps. The translated steps can be checked independently, too, so the validation process fits well for parallelisation and for the management of large proof scripts. Moreover, our approach is more general; any kind of implicit induction proof can be considered because the limitations imposed by the proof reconstruction techniques no longer exist. An implementation that integrates automatic translators for generating fully checkable Coq scripts from Spike proofs is reported.

1 Motivations

Implicit induction proof techniques allow for automated reasoning on inductive properties of equational specifications. Up to now, implicit induction theorem provers, as Spike [3], have been successfully used in treating non-trivial case studies, for example, the validation of the JavaCard Platform [3] and the conformance algorithm of a telecommunications protocol [19]. Spike proofs are highly automated; in [3], almost half of the JavaCard bytecode instructions have been checked completely automatically, i.e. do not require users to provide additional lemmas, and done in a reasonable time. These proofs are shallow but large, involving several induction, case analysis and rewriting steps. Even if the theoretical backgrounds of implicit induction proof techniques are widely accepted by the scientific community, their implementation inside theorem provers is error-prone and their certification still stands for a challenge. The most common and

ancient validation technique is by human checking. It may work well when the proof size is reasonable, but is not suited for checking many (even easy) inference steps.

To the other extreme, the approach is to certify inference systems such that any proof developed inside certified proof environments is guaranteed to be sound. For example, the proofs done with the Coq proof assistant [24] are mechanically checked by the kernel of its inference system, which is small enough to be human checked and reliable. However, the process for certifying complex software systems is tedious [13], in the case of Spike it would require the validation of thousands of OCaml code lines. The midway approach that we adopted would therefore not consist in certifying inference systems, but in checking proofs that would play the role of test cases for the unreliable systems. More precisely, the Spike proofs are converted into Coq scripts checkable by the Coq kernel. During the last decade, different reasoning tools successfully applied this approach by developing conversion options for Coq [8,4].

The soundness of any implicit induction reasoning can be easily explained using ‘proof-by-contradiction’ arguments characterizing the ‘Descente Infinie’ induction-based approaches [21]. For example, proving that a non-empty and potentially infinite set of first-order ground formulas \mathbf{F} is true requires: 1) (the ‘well-foundedness’ requirement) a well-founded induction ordering over the formulas from \mathbf{F} , i.e. there are no infinite strictly descending sequences of formulas, and 2) (the ‘counterexample non-minimality’ requirement) to prove that for each false formula from \mathbf{F} , called *counterexample*, there exists a smaller one. The proof starts by assuming by contradiction that there is a counterexample in \mathbf{F} . Therefore, by 2), there is a smaller counterexample for which is an even smaller counterexample by applying 2) again, and so on. In this way, one can build an infinite strictly descending sequence of counterexamples (hence the name of ‘Descente Infinie’), which contradicts 1).

An example of implicit induction proof. Implicit induction proofs as performed by Spike can easily manipulate conjectures about specifications integrating mutually defined functions. Let’s consider the universally quantified axioms that mutually define the even, respectively the odd functions over the naturals:

$$\text{even}(0) = \text{true} \tag{1}$$

$$\text{odd}(x) = \text{true} \Rightarrow \text{even}(S(x)) = \text{true} \tag{2}$$

$$\text{odd}(x) = \text{false} \Rightarrow \text{even}(S(x)) = \text{false} \tag{3}$$

$$\text{odd}(0) = \text{false} \tag{4}$$

$$\text{even}(x) = \text{true} \Rightarrow \text{odd}(S(x)) = \text{true} \tag{5}$$

$$\text{even}(x) = \text{false} \Rightarrow \text{odd}(S(x)) = \text{false} \tag{6}$$

using the constructors 0 and successor S for the naturals. Let’s assume that we want to prove the conjectures $\text{odd}(S(\text{plus}(x, x))) = \text{true}$ and $\text{even}(\text{plus}(y, y)) = \text{true}$, where *plus* is the addition function over the naturals, defined by the axioms

$plus(0, x) = x$ and $plus(S(x), y) = S(plus(x, y))$. In addition, we assume the lemma $plus(x, S(y)) = S(plus(x, y))$.

To prove that the conjectures are true (w.r.t the above axioms) using a 'Descente Infinie' induction-based approach means to check the 'well-foundedness' and 'counterexample non-minimality' requirements. For example, the well-founded induction ordering from the first requirement, denoted by \ll_{rpo} , can be defined as a multiset extension of the RPO ordering \prec_{rpo} based on the precedence $0 <_F S <_F plus <_F even$ with multiset status for the defined functions, where odd has the same precedence as $even$.¹ The \prec_{rpo} and \ll_{rpo} orderings can also be used to orient the above axioms from left to right into rewrite rules. In general, the equality $a = b \Rightarrow l = r$ is oriented into $a = b \Rightarrow l \rightarrow r$ if l is the unique greatest term in the equality.

The rewrite rules are involved into rewrite operations that replace terms with smaller ones, essential in the quest for smaller counterexamples during the realization of the 'counterexample non-minimality' requirement. Let's assume that there is a counterexample in the first conjecture, $c_1 : odd(S(plus(n, n))) = true$, for some natural n . A smaller counterexample can be pointed out by performing a case analysis on $even(plus(n, n))$: if it is *false* then $odd(S(plus(n, n)))$ is *false* according to (6), i.e. $even(plus(n, n)) = false \Rightarrow false = true$; if it is *true*, then we have the tautology $c_3 : even(plus(n, n)) = true \Rightarrow true = true$, according to (5). So, $c_4 : even(plus(n, n)) = false \Rightarrow false = true$ is a counterexample smaller than $odd(S(plus(n, n))) = true$ because we replaced $odd(S(plus(n, n)))$ by smaller terms. c_4 can be rewritten with the second conjecture to get the smaller instance $c_2 : even(plus(n, n)) = true$. Since the rewritten equality, $c_7 : true = false \Rightarrow false = true$, is a tautology, c_2 is a counterexample.

We can go further and show that there exists at least one counterexample smaller than c_2 . Since n is a natural, it can be either 0 or $S(n')$, for some natural n' . If n is 0, $plus(0, 0)$ is 0, so $c_6 : even(0) = true$ can be rewritten by (11) to $c_9 : true = true$ which is a tautology. Therefore, n should be $S(n')$. Rewriting $even(plus(S(n'), S(n')))$ with the axiom $plus(S(x), y) \rightarrow S(plus(x, y))$ results $c_5 : even(S(plus(n', S(n')))) = true$. By using the lemma $plus(x, S(y)) \rightarrow S(plus(x, y))$, we obtain the smaller counterexample $c_8 : even(S(S(plus(n', n')))) = true$. Another case analysis on $odd(S(plus(n', n')))$ yields two new equalities: i) $c_{11} : odd(S(plus(n', n'))) = true \Rightarrow true = true$, which is a tautology, and ii) the smaller counterexample $c_{10} : odd(S(plus(n', n'))) = false \Rightarrow false = true$. The instance of the first conjecture $c'_1 : odd(S(plus(n', n'))) = true$ is smaller and can be used to rewrite it into the tautology $c_{12} : true = false \Rightarrow false = true$. Therefore, c'_1 is a smaller counterexample. To sum up, c'_1 is smaller than the original counterexample c_1 . The quest for smaller counterexamples can be similarly repeated *ad infinitum*, which contradicts the 'well-foundedness' requirement. So, we conclude that the conjecture $odd(S(plus(x, x))) = true$ is true. For similar reasons, $even(plus(x, x)) = true$ is also true.

¹ For more formal definitions, the reader may consult Section 2.

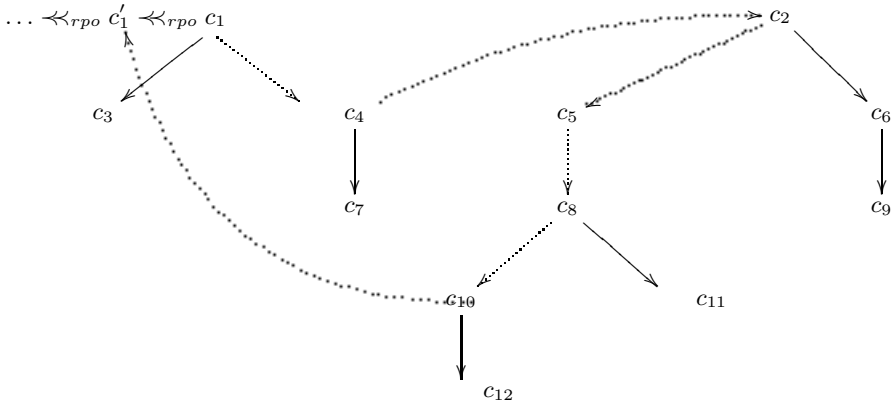


Fig. 1. Example of simultaneous induction proof; the assumption of false initial conjectures generates an infinite strictly descending sequence of counterexamples, as indicated by the dotted arrows

The proof employs *simultaneous induction*, i.e. instances of the first conjecture are used as induction hypotheses in the proof of the second, and viceversa, as depicted by the proof graph from Fig. 1. The nodes are labelled with the names of the ground equalities encountered during the proof. A branching node points out a case analysis operation on the corresponding equality. The solid (resp. dotted) arrows give pathways to tautologies (resp. smaller counterexamples). Notice the infinite pathway of counterexamples which justifies the application of the 'Descente Infinie' induction principle.

Related approaches. Previous attempts to validate Spike proofs have been done by Courant [9] and Kaliszyk [12] using the explicit induction tactics provided by Coq. Their approaches are limited mainly because the explicit induction proof methods require hierarchical manipulation of induction hypotheses; the proofs have a tree-shape such that the exchange of information is forbidden between different branches. Therefore, it is impossible to perform simultaneous induction proofs.

Up to now, the current solution is to reconstruct implicit into explicit induction proofs. In [9], only the proofs done with restricted versions of the Spike system (the K -systems) are considered. Their inference rules have to obey some conditions that would allow proof representations under the form of a tree labelled with judgements. On the other hand, [12] identifies explicit induction schemas from the proof steps that instantiate variables. More recently, Nahon et al. [15] proposed a theoretical foundation based on deduction modulo that permits automated construction of inductive proofs into the sequent calculus, ready for insertion into proof assistants. As Brotherston has already shown in his PhD thesis [6], the idea is to perform 'Descente Infinie'-style proofs using an extension of the sequent calculus with explicit inductive schemas that define

conjectures in terms of induction hypotheses and conclusions linked by shared variables. However, it is difficult to see how the Spike proofs can be reproduced by this calculus since Spike does not assume variable sharing between different conjectures.² In our opinion, implicit and explicit induction are just two different proof techniques; they can be even combined during the proof process, as shown in [22].

Some proof assistants already integrate automatizing mechanisms for induction reasoning, for example Coq [25], Agda [14], IsaPlanner [10], NuPrl [16] and Clam [7]. To the best of our knowledge, all of them use explicit inductive definition schemas. On the other hand, there is no similar work w.r.t. the direct integration of implicit induction techniques. The first successful but manual conversion and checking operations of an implicit induction proof by Coq have been reported in [23].

Structure of the paper. The main contribution of the paper is a methodology for mechanically checking potentially any implicit induction proof. We will explain in particular how the methodology works for validating Spike proofs with Coq. After presenting the basic notions and notations in Section 2, we will detail in Section 3 the implicit induction proof techniques, then introduce a simplified version of the Spike inference system but strong enough to prove the introductory example. Section 4 develops the idea of explicitly defining, then implementing the underlying implicit induction principles. The first part will prove the soundness of the 'Descente Infinie' induction principle instantiated for a particular well-founded induction ordering. The second part uses deductive reasoning based on current techniques such as rewriting, case analysis and tautology elimination in order to check the 'counterexample non-minimality' requirement. In addition, we will show and give examples using implementation details that one can build a one-to-one translation of the Spike inference rules into Coq tactics, hence the generality of our approach. The methodology is applied for automatically translating and validating the Spike proof of the introductory example and other non-trivial examples. The conclusions and future work are given in the last section.

2 Basic Notions

Conditional specifications consist of axioms representing conditional equalities between terms built on an alphabet of (arity-fixed) function symbols \mathcal{F} and (universally quantified) variables \mathcal{V} . The axioms define some of the function symbols, the other symbols are referred to as constructors. The specifications of interest are many-sorted and we assume that for each sort s there exists at least one constructor of sort s . The conjectures are clauses representing disjunctions of literals, where a literal is either an equality or an inequality between two terms. Sometimes, clauses that have at most one equality are represented as implications.

² For more details, the reader may consult the Spike proof of the introductory example from Subsection 3.1.

The set of terms is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and the set of ground (or no variable) terms by $\mathcal{T}(\mathcal{F})$. New terms (resp. clauses), called *instances*, can be built from existing terms (resp. clauses) by replacing variables with terms. The mappings from variables to terms are called substitutions. Two terms unify if there is a substitution σ such that $s\sigma$ and $t\sigma$ are syntactically equal, denoted by $s\sigma \equiv t\sigma$. A term s matches the term t if there exists a substitution σ such that $s\sigma \equiv t$. The subterm t of a clause C is identified by its position p , denoted by $C[t]_p$.

A *quasi-ordering* \leq is a reflexive and transitive binary relation, consisting of strict and equivalence parts. The strict part of a quasi-ordering is called *ordering* and denoted by $<$. A quasi-ordering \leq defined over the elements of a nonempty set A is *well-founded* if there do not exist infinite strictly descending sequences $\dots < x_2 < x_1$ of elements of A . A binary relation R is *stable under substitutions* if whenever $s R t$ then $(s\sigma) R (t\sigma)$, for any substitution σ . A *reduction ordering* is a transitive and irreflexive relation that is well-founded, stable under substitutions and stable under contexts (i.e. $s R t$ implies $u[s] R u[t]$). An example of syntactic reduction ordering over terms is \prec_{rpo} from Section [11](#). \prec_{rpo} is recursively defined as follows. Given $f \in \mathcal{F}$, a status function τ for \mathcal{F} returns $\tau(f) \in \{\text{lex}, \text{mul}\}$, for each $f \in \mathcal{F}$, where *lex* stands for lexicographic status and *mul* for multiset status. Given $<_{\mathcal{F}}$ an ordering over \mathcal{F} , an ordering \prec_{rpo} on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is defined as follows: for all terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $t \prec_{rpo} s$ if $s = f(s_1, \dots, s_m)$ and i) either $s_i = t$ or $t \prec_{rpo} s_i$ for some s_i , $1 \leq i \leq m$, or ii) $t = g(t_1, \dots, t_n)$, $t_i \prec_{rpo} s$ for all i , $1 \leq i \leq n$ and either a) $g <_{\mathcal{F}} f$, or b) $f = g$ and $(t_1, \dots, t_n) \prec_{rpo}^{\tau(f)} (s_1, \dots, s_n)$. \prec_{rpo}^{lex} is the lexicographic extension of \prec_{rpo} , i.e. $(a_1, \dots, a_n) \prec_{rpo}^{\text{lex}} (b_1, \dots, b_n)$ if either i) $a_1 \prec_{rpo} b_1$ or ii) $a_1 = b_1$ and $(a_2, \dots, a_n) \prec_{rpo}^{\text{lex}} (b_2, \dots, b_n)$. \prec_{rpo}^{mul} , also denoted by \ll_{rpo} in the rest of the paper, is the multiset extension of \prec_{rpo} . Two terms s and t are equivalent if either a) $s \equiv t$, or b) $s \equiv f(s_1, \dots, s_n)$, $t \equiv g(t_1, \dots, t_n)$, f and g have the same arity and precedence and, for the case when f and g have multiset status, it exists (t'_1, \dots, t'_n) such that s_i is equivalent with t'_i , for all $1 \leq i \leq n$, and (t'_1, \dots, t'_n) is a permutation of (t_1, \dots, t_n) . The relation \ll_{rpo} is defined in the next paragraph.

(Conditional) equalities can be transformed into (conditional) rewrite rules of the form $a = b \Rightarrow l \rightarrow r$ if l is greater than a , b and r . A rewrite system \mathcal{R} consists of a set of rewrite rules. The rewrite relation $\rightarrow_{\mathcal{R}}$ denotes rewrite operations only with rewrite rules from \mathcal{R} . The reflexive transitive (resp. equivalence) closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\rightarrow_{\mathcal{R}}^*$ (resp. $\leftrightarrow_{\mathcal{R}}$). Given a substitution σ , a rewrite rule $a = b \Rightarrow l \rightarrow r$ and a clause C such that $C[l\sigma]_u$, a *rewrite operation* replaces $C[l\sigma]_u$ by $a\sigma = b\sigma \Rightarrow C[r\sigma]_u$.³ Any clause can also be represented as the multiset of its literals. A well-founded and stable under substitutions ordering over clauses can be built as the multiset extension of a reduction ordering over terms, as follows. Given two multisets of terms A_1 and A_2 , we write $A_1 \ll_{rpo} A_2$ if, after the pairwise elimination of the equivalent terms from A_1 and A_2 , $\forall s \in A_1$, $\exists t \in A_2$ such that $s \prec_{rpo} t$. A_1 and A_2 are equivalent if both of them become empty after the elimination process. Finally, $\Phi_{\ll_{rpo} C}$ denotes the set $\{\psi\sigma \mid \psi \in$

³ For details on term rewriting, the reader may consult [\[11\]](#).

Φ , σ a substitution and $\{\psi\sigma \ll_{rpo} C\}$ of instances of clauses from Φ that are smaller than the clause C .

$s = t$ is an *inductive theorem* of a set of axioms Ax orientable into a rewrite system \mathcal{R} if, for any of its ground instances $s\sigma = t\sigma$, we have $s\sigma \xrightarrow{*}_{\mathcal{R}} t\sigma$. A ground equality is a *counterexample* if it is not an inductive theorem. $s = t$ is *false* if it ‘contains’ (i.e. one of its instances is) a counterexample. Tautologies are inductive theorems either of the form $(e \Rightarrow)t = t$, where e is an unconditional equality and t a term, or of the form $e_1 \Rightarrow e_2$, where $e_1 \equiv a = b$ and $e_2 \equiv a = b$ (or $b = a$), and a, b are terms.

3 Implicit Induction Proofs with Spike

In the introductory part, we have shown how the ‘Descente Infinie’ induction principle can help to justify the soundness of implicit induction proofs. We can give a more pragmatic application of this principle since well-founded orderings guarantee the existence of minimal elements. The proof is done by contradiction, by assuming that there is no such minimal element, as follows. We pick an arbitrary element from the set. Since it is not minimal, there exists a smaller one which is not minimal, and so on. In this way, an infinite strictly descending sequence of elements can be built. This contradicts the fact that the ordering is well-founded.

The implicit induction inference systems consist of inference rules that replace a conjecture with a potentially empty set of new conjectures. Proof derivations are built by the successive application of inference rules on an initial set of conjectures. We say that an implicit induction inference system is *sound* if the minimal counterexamples are preserved in the derivations, i.e. whenever an inference rule replaces a conjecture containing a minimal counterexample, there is a further state in the derivation, usually the next state, with a conjecture having an equivalent (w.r.t. well-founded induction quasi-ordering) minimal counterexample. The soundness property is interesting because we can state that the initial set of conjectures are true whenever the derivations end with an empty set of conjectures. Otherwise, assuming that there is a counterexample in the initial set of conjectures, there exists a minimal counterexample in the set of conjectures encountered in the derivation. Since any minimal counterexample is preserved, it should be present in the last state of the derivation. On the other hand, this is not possible because the last state is empty.

From a logical point of view, it is sufficient to show that the replaced minimal counterexample is a consequence of the equivalent minimal counterexample from the further state. The consequence relation is not affected if other true formulas like the axioms and smaller conjectures from the derivation, or equivalent conjectures from further states are involved [20]. These conjectures play the role of *induction hypotheses*.

Implicit induction proofs similar to that presented as example in Section 4 can be highly automated if the quest for smaller or equivalent minimal counterexamples is limited only to the conjectures from the next state. The trick is to store

in the current state previously replaced conjectures that do not contain minimal counterexamples, called *premises*⁴. In this case, the initial set of conjectures are true for any derivation starting with an empty set of premises and finishing with an empty set of conjectures.

3.1 The Spike Prover

Spike is an implicit induction prover that reasons on conditional specifications. In the last decade, it has been successfully used in many real-size case studies from different areas (telecommunications [19], programming language platforms [3], collaborative editing systems [11], web services [18], etc)⁵.

Specifications and properties. The conditional specifications are sorted and consist in sets of axioms defining functions, represented as conditional equalities. We assume that a reduction ordering exists such that it can orient the axioms into rewrite rules. The specifications accepted by Spike should be coherent, i.e. any formula and its negation cannot be simultaneously consequences of the axioms, and complete, i.e. the functions are defined in any point of the domain.

A sufficient condition to achieve coherent conditional specifications is the *ground convergence*, i.e. the rewriting process of any ground term terminates and yields a unique result [5]. This property is easier to check for specifications based on free constructors: the lhs of the rewrite rules defining a function symbol f is basic, i.e. of the form $f(\vec{t})$ with $\vec{t} \equiv t_1, \dots, t_n$ a vector of n constructor terms, and there is no equality relation between two constructors terms starting with different constructor symbols.

Complete specifications may define only *operational sufficiently complete* function symbols f , i.e. for any ground basic term $f(\vec{s})$, i) there are matching axioms $a_i = b_i \Rightarrow l_i \rightarrow r_i$ such that $f(\vec{s}) \equiv l_i \sigma_i$, and ii) $\bigvee_i a_i \sigma_i = b_i \sigma_i$ is an inductive theorem and any two matching substitutions are equivalent modulo renaming. The test for inductive validity is generally undecidable, therefore we will restrict to the case when the specifications contain only conditional axioms with conditions having the form $a = b$, where b is either *true* or *false*.

The inference system. The Spike inference system is made of inference rules manipulating clauses, representing transitions between states $(E, H) \vdash (E', H')$, where E, E' are conjectures, and H, H' premises. In Fig. 2, we introduce a simplified version of it consisting only of 4 inference rules.

GENERATE applies on clauses having subterms that unify with some lhs of the axioms, referred to as unifying axioms. The soundness of the system is preserved if all the unifying axioms are considered [3]. TOTAL CASE REWRITING can apply on clauses with subterms that are matched by some lhs of conditional axioms. If the position of the subterm resides inside a maximal term of the treated clause C , then C cannot have minimal counterexamples and is added to the set

⁴ Notice that the two notions of induction hypothesis and premise are different.

⁵ For a more detailed list of publications, see [17].

GENERATE: $(E \cup \{C[t]_p\}, H) \vdash (E \cup (\cup_{\sigma} E_{\sigma}), H)$
 if E_{σ} is $\{a\sigma = b\sigma \Rightarrow C\sigma[r\sigma]_p\}$ and $a = b \Rightarrow l \rightarrow r \in Ax$ s.t. $t\sigma \equiv l\sigma$.

TOTAL CASE REWRITING: $(E \cup \{C[t]_p\}, H) \vdash (E \cup E', H \cup \{C\})$
 if E' is $\{a_1\sigma_1 = true \Rightarrow C[r_1\sigma_1]_p, a_2\sigma_2 = false \Rightarrow C[r_2\sigma_2]_p\}$ and
 $a_1 = true \Rightarrow l_1 \rightarrow r_1, a_2 = false \Rightarrow l_2 \rightarrow r_2 \in Ax$ s.t. $l_1\sigma_1 \equiv t$ and $l_2\sigma_2 \equiv t$.

(UNCONDITIONAL) REWRITING: $(E \cup \{C\}, H) \vdash (E \cup \{C'\}, H)$
 if $C \rightarrow_{Ax \cup \mathcal{L} \cup (H \cup E) \prec\prec_{rpo} C} C'$.

TAUTOLOGY: $(E \cup \{C\}, H) \vdash (E, H)$
 if C is a tautology.

Fig. 2. A simplified version of the Spike inference system

of premises. REWRITING rewrites clauses with unconditional orientable axioms Ax , lemmas \mathcal{L} , and smaller instances of premises and conjectures. TAUTOLOGY deletes tautologies.

Proof example. The above inference system can prove $e_1 : even(plus(x, x)) = true$ and $e_2 : odd(S(plus(y, y))) = true$ using the lemma $plus(x, S(y)) = S(plus(x, y))$ and the \prec_{rpo}, \ll_{rpo} orderings from the introductory example. The initial state of the proof is $(\{e_1, e_2\}, \emptyset)$. TOTAL CASE REWRITE (TCR) is applied on $odd(S(plus(y, y)))$ of e_2 with axioms (5) and (6) to yield $e_3 : even(plus(y, y)) = true \Rightarrow true = true$ and $e_4 : even(plus(y, y)) = false \Rightarrow false = true$. From the current state $(\{e_1, e_3, e_4\}, \{e_2\})$, e_3 is deleted by TAUTOLOGY (T). REWRITING (R) simplifies e_4 with the conjecture e_1 to give the tautology $e_5 : true = false \Rightarrow false = true$, which is deleted in the next step. GENERATE (G) can be applied on the remaining conjecture, e_1 . The term $plus(x, x)$ is unified with the lhs of the axioms defining $plus$, to yield $e_6 : even(0) = true$ and $e_7 : even(S(plus(z, S(z)))) = true$, where z is a new variable. e_6 is reduced to the tautology $e_8 : true = true$ by REWRITING with axiom (II), then deleted using TAUTOLOGY. e_7 is reduced to $e_9 : even(S(S(plus(z, z)))) = true$ by REWRITING with the lemma. A second TOTAL CASE REWRITING on the term $even(S(S(plus(z, z))))$ of e_9 yields $e_{10} : odd(S(plus(z, z))) = true \Rightarrow true = true$ and $e_{11} : odd(S(plus(z, z))) = false \Rightarrow false = true$. From the new current proof state $(\{e_{10}, e_{11}\}, \{e_2, e_9\})$, e_{10} is deleted by TAUTOLOGY and e_{11} is reduced to the tautology $e_{12} : true = false \Rightarrow false = true$ by REWRITING with the premise e_2 . The proof finishes after the application of TAUTOLOGY on the last conjecture e_{12} .

The proof is schematised as follows: $(\{e_1, e_2\}, \emptyset) \vdash^{(TCR)} (\{e_1, \underline{e_3}, e_4\}, \{e_2\}) \vdash^{(T)} (\{e_1, e_4\}, \{e_2\}) \vdash^{(R)} (\{e_1, \underline{e_5}\}, \{e_2\}) \vdash^{(T)} (\{\underline{e_1}\}, \{e_2\}) \vdash^{(G)} (\{e_6, e_7\}, \{e_2\}) \vdash^{(R)} (\{e_8, e_7\}, \{e_2\}) \vdash^{(T)} (\{e_7\}, \{e_2\}) \vdash^{(R)} (\{e_9\}, \{e_2\}) \vdash^{(TCR)} (\{e_{10}, e_{11}\}, \{e_2, e_9\}) \vdash^{(T)} (\{\underline{e_{11}}\}, \{e_2, e_9\}) \vdash^{(R)} (\{\underline{e_{12}}\}, \{e_2, e_9\}) \vdash^{(T)} (\emptyset, \{e_2, e_9\})$. The underlined formula from a proof state is the conjecture to which the corresponding inference rule is applied.

4 Translating and Checking Spike Specifications

Spike specifications and proofs can be directly translated into Coq scripts. Each Spike datatype and function definition is translated into an equivalent Coq representation. In addition, the underlying 'Descente Infinie' induction principle is explicitly defined. In order to do this, Coq formulas have to be syntactically represented and compared as Spike does. The idea is to use a term algebra that abstracts the Coq datatypes and function symbols such that each Coq term is weighted with an abstract term. In this way, comparing two Coq formulas reduces to comparing the weights of their built-in terms.

The Coq scripts consist in the specification and proof parts. The specification part defines the abstract term algebra and the RPO ordering built from abstractions of Spike function symbols and their precedence. Then, the Coq datatypes and functions are introduced, together with translation functions for each datatype, which show how to abstract constructor terms. In the proof part, firstly a weight is associated to each conjecture encountered in the Spike proof, then lemmas about weight comparisons are defined. The 'Descente Infinie' principle is explicitly stated in the main lemma which proves that for each false instance from the set of conjectures, there is another one with a smaller weight. Another lemma states that all conjectures are true. The last lemma trivially concludes that the initial conjectures are true, too.

Datatypes and the function definitions. The Coq datatypes, function definitions and translation functions have to be defined manually by the user. Spike specifications can be annotated with Coq code that is inserted into the generated Coq script during the translation process. As example, here is the code for the introductory example, consisting of a set of computing functions :

```

Fixpoint model_nat (v: nat): term :=
match v with
| 0 => (Term id_0 nil)
| S x => let r := model_nat x in (Term
id_S (r::nil))
end.
Fixpoint model_bool (v: bool): term
:=
match v with
| true => (Term id_true nil)
| false => (Term id_false nil)
end.
Fixpoint plus (x y:nat): nat :=
match x with
| 0 => y
| S x' => S (plus x' y)
end.

Fixpoint even (v:nat): bool :=
match v with
| 0 => true
| S x => match odd x with
| true => true
| false => false
end
end
with odd (v:nat): bool :=
match v with
| 0 => false
| S x => match even x with
| true => true
| false => false
end
end.

```

where `id_x` is the abstraction of a function symbol `x`, `model_sort` is the translation function for `sort` and **term** (recursively defined as `Inductive term : Set := | Var : variable → term | Term : symbol → list term → term.`) is the type of the abstracted terms provided by COCCINELLE [8], a Coq library well suited for modelling mathematical notions needed for rewriting, such as term algebras and RPO. This is the checking step for the Spike specification and its properties, since any computing function written as a structurally recursive function is guaranteed to be complete and ground convergent, if accepted by Coq. The termination and ground confluence properties result from the fact that every recursive call is executed on a structurally smaller argument and that the inserted Coq script is based on free constructors, respectively.

The ordering over conditional equalities. An important part of the Coq script concerns the implementation of the induction ordering involved in the ‘well-foundedness’ requirement. Its definition and the computation of comparisons between conjectures are based on computable functions and inductive predicates provided by COCCINELLE. COCCINELLE formalises RPO in a generic way using a precedence and a status (multiset/lexicographic) for each function symbol. Spike automatically generates a term algebra starting from the abstract function symbols which preserve the precedence of the original symbols. Then, the algebra is applied as argument to the functor of the generic RPO module which establishes fundamental properties about RPO orderings, for example, any RPO ordering is a reduction ordering. Also, the well-foundedness of the induction ordering, denoted below by `less`, is provided.

In order to deal with mutually recursive functions, the RPO definition from the generic module has been extended to take into account precedence relations with equivalent symbols. Also, even if many interesting properties about the RPO orderings have been already provided by COCCINELLE, some about the multiset extension of RPO were missing. A new function computing weight comparisons was defined and its equivalence with `less` was proved as a soundness lemma. This guarantees that any terminating weight comparison operation is sound. The termination property is ensured if the size of the terms is limited by a global maximal value. For a given proof, it has to be greater than the double of the maximal size of the terms encountered in the proof. The stability under substitutions of `less` was also proved.

The ‘Descente Infinie’ induction principle is implemented in three steps. Firstly, the existence of minimal elements in any non-empty set of weights, represented as lists of abstract terms, is guaranteed:

$$\forall Y: \mathcal{P}(\mathbf{list\ term}), (\exists y, y \in Y) \rightarrow \exists n \in Y, \forall m \in Y, \neg(\mathbf{less\ } m\ n).$$

Then, the ‘counterexample non-minimality’ requirement is implemented such that, for any list of pairs (`coq_formula`, `weight`), whenever a formula instance is false there is another one with a smaller weight:

$$\forall F \in \mathbf{F}, \forall \vec{x}, \neg\pi_1(F\ \vec{x}) \rightarrow (\exists F_1 \in \mathbf{F}, \exists \vec{x}_1, \neg\pi_1(F_1\ \vec{x}_1) \wedge \mathbf{less}\ \pi_2(F_1\ \vec{x}_1)\ \pi_2(F\ \vec{x}))$$

where π_1 and π_2 are the first and second projections of a pair, respectively. \mathbf{F} is the set of functors that associate a pair (formula, weight) to a vector of terms.

Finally, we prove that $\forall F \in \mathbf{F}, \forall \vec{x}, \pi_1(F \vec{x})$. The first and the third step require classical reasoning. The third step is valid for any set \mathbf{F} satisfying the ‘counterexample non-minimality’ requirement.

Satisfying the ‘counterexample non-minimality’ requirement. All crucial information for satisfying this requirement can be extracted from the Spike proof, in particular the set \mathbf{F} , the conjectures containing smaller counterexamples and the comparisons between the weights of the conjecture instances. For example, the set \mathbf{F} for validating the introductory example consists of functors associated to each of the twelve conjectures e_1 to e_{12} : $\{(fun\ x \Rightarrow (even(plus\ x\ x) = true, weight_{e_1})), \dots, (fun\ _ \Rightarrow (true = false \Rightarrow false = true, weight_{e_{12}}))\}$. The proof consists of a case analysis on the conjectures that may have counterexamples following the ‘quest for smaller counterexample’ reasoning represented in Fig. 1. Each of the cases can be treated independently and in any order. For example, if e_1 has a counterexample, by performing a case analysis on x instantiated with 0 and $(S\ z)$, a smaller counterexample should exist either in e_6 or e_7 .

An important part of the proof is spent on verifying weight comparisons. The comparison proofs can be automatically generated and consist in i) the replacement of all terms of the form `(model_sort x)` with COCCINELLE abstraction variables of the form `(Var i)`, where i is a natural, ii) the use of the ‘stability under substitutions’ property of `less` which allows to perform the comparison tests on weights with abstraction variables instead of using the original weights, iii) computing the comparison result of weights with abstraction variables, and iv) validating the result using the soundness lemma. In order to perform iii) for the case when the weights of two compared terms, $weight_1$ and $weight_2$, consist of m and n abstracted terms, respectively, we have to check that the size of any term from $weight_1$ added with the size of any term from $weight_2$ does not exceed the maximal value. The total number of size comparisons is therefore $m * n$.

One-to-one translation of Spike inference steps. Automatic translators have been implemented for the inference rules from Fig. 2. Deductive steps like rewriting, case analysis and tautology elimination operations are directly translated into Coq proof commands.

The variable instantiation schemas of GENERATE are controlled by Coq functional schemas [2]. This is the checking step for complete instantiation schemas. For example, to show that x from e_1 is replaced by 0 and $(S\ z)$, we define a function `f` with all the instantiation cases:

```
Fixpoint f (x: nat) {struct x} : nat :=
  match x with
  | 0 => 0
  | (S z) => 0
  end.
```

Functional Scheme `f_ind` := Induction for f Sort Prop.

The instances are generated by the Coq script `pattern x, (f x). apply f_ind`. The idea is that, for each instance $HFabs$, we choose the functor from \mathbf{F} corresponding to the appropriate conjecture from the Spike proof script, and show that $HFabs$ is logically equivalent with a smaller instance. For the case when x is 0, here is the generated Coq script: `exists (fun _ => ((even 0) = true, weighte6)). eexists. split. contradict HFabs. auto. apply less_HFabs_e6`. The first two commands instantiate e_6 , then the proof is split in two: the ‘logically equivalence’ and comparison parts consisting of the application of `auto`, a tactic enough powerful to show the equivalence between `plus 0 0` and `0`, and of the comparison lemma `less_HFabs_e6`, respectively.

The translation of TOTAL CASE ANALYSIS is similar, excepting that the case analysis on whether a condition a is either *true* or *false* is performed by `destruct a`. This is the checking step for the inductive validity of the disjunction of rewrite rules conditions. This translation offers a better control of the rewritten term than `auto`. For example, the fact that $C[f(t)]$ is rewritten with a rule of the form $f(x) \rightarrow \dots$ can be simulated by `pattern t. simpl f. cbv beta. pattern t` isolates t from C , `simpl f` rewrites $f(t)$ and `cbv beta` puts back the resulted term in C . REWRITING is translated using the same trick for rewriting.

There is no need to reduce or compare tautologies with other conjectures. Tautologies can be eliminated by Coq using `intros. auto. intros` separates the conditions from the conclusion of a conditional equality. `auto` checks either that the conclusion is of the form $t = t$ or that the conditions contain the conclusion.

Experimental results. Table II displays some statistics about the execution time and size of the Coq scripts generated with our implementation for several Spike specifications and conjectures: properties about *plus* and different definitions of *even* and *odd* (conjectures 1] to 10]), about other recursive data structures like trees and lists (conjectures 11] and 12]) and conjectures stating the soundness of a simple insertion sorting algorithm (conjectures 13] to 16]).

The third and fourth columns show the number of comparison lemmas and the time needed for their validation, respectively. The fifth and sixth columns display the cardinality of \mathbf{F} and the validation time of the corresponding formulas, respectively. The last column gives the total execution time, including the overhead time needed for checking the algebra and the ordering. The overhead time for the first ten conjectures was 45.5s, for the next two conjectures was 1m10s and for the last conjectures 59.6s. The statistics for 5], 6] and 7] take into account that 4] was executed before being used as lemma. On the other hand, 16] requires a lemma whose proof in Spike needs arithmetic reasoning. In Coq, the lemma was represented as a hypothesis. Finally, the execution time of each of the Spike proofs and the translation operations lasted less than one second.

The experiments have been done on a MacBook Air featuring a 1.6 GHz Intel Core 2 Duo processor and 2 GB RAM.

Table 1. Statistics about the Coq validation process of some implicit induction proofs

#	conjecture(s)	# less	less time	# F	proof time	total time
1]	$evenr(plus(x, y)) = true \wedge$ $evenr(plus(y, z)) = true$ $\Rightarrow evenr(plus(x, z)) = true$	27	3m15s	22	0m50.5s	4m51s
2]	$evenm(x) = evenr(x)$	9	0m10s	8	0m00.5s	0m56s
3]	$plus(x, 0) = x$	4	0m02s	3	0m02s	0m48s
4]	$plus(x, S(y)) = S(plus(x, y))$	7	0m01.5s	8	0m01.5s	0m56s
5]	$even(plus(x, x)) = true$ and $odd(S(plus(x, x))) = true$ (needs 4)]	17	0m44s	20	0m06.5s	1m34s
6]	$plus(x, y) = plus(y, x)$ (needs 4)]	26	0m39s	24	0m05.5s	1m30s
7]	$evenm(plus(x, x)) = true$ and $oddm(plus(x, x)) = false$ (needs 4)]	21	1m13s	20	0m13.5s	2m12s
8]	$evenr(S(x)) = true \Rightarrow$ $true = oddm(x)$	11	0m49s	10	0m10.5s	1m45s
9]	$oddc(x) = oddm(x)$	19	1m09s	16	0m29.5s	2m24s
10]	$plus(x, plus(y, z)) = plus(plus(x, y), z)$	7	0m46s	6	0m23.5s	1m55s
11]	$flat(ins(x, t)) = Cons(x, flat(t))$	14	0m53s	15	0m38.4s	2m21s
12]	$app(x, app(y, z)) = app(app(x, y), z)$	8	0m46s	17	0m22.4s	1m58s
13]	$sorted(Cons(x, y)) = true \Rightarrow$ $sorted(y) = true$	5	0m07s	6	0m30s	1m47s
14]	$length(insert(x, y)) = S(length(y))$	9	0m28s	10	0m27s	2m05s
15]	$length(isort(x)) = length(x)$	14	0m36s	16	0m32s	2m18s
16]	$sorted(isort(x)) = true$ (needs lemma)	5	0m02s	4	0m27s	1m39s

5 Conclusions and Future Work

We have proposed a methodology for directly checking potentially any implicit induction proofs using certified proof environments. By the means of the Coq proof assistant, the methodology was applied to check non-trivial proofs done with a restricted version of the Spike system. The ‘Descente Infinie’ induction principle underlying the Spike proofs was explicitly defined and every single Spike inference step has been translated into equivalent Coq script using automated translators.

One of our long-term goals is to automatically check large Spike proofs. As shown by the experimental results, the checking time is some orders of magnitude longer than for producing a Spike proof, so the current implementation has to be optimised. To meet this objective, the fixed part of the scripts (i.e. the specification and the RPO ordering definition) can be validated in a separate Coq module to be imported, instead of being (re)validated each time a new conjecture is proved. Also, a lot of time is spent validating comparison lemmas. Computing the size of all terms in advance would linearize the complexity of comparison proofs. Last but not least, since the translated inference steps can be performed independently, it would be interesting to check them concurrently.

Some other Spike proofs require more sophisticated inference rules to deal with arithmetic reasoning, parametrized specifications and existential variables [3], or more general versions of the presented inference rules, for example TOTAL CASE REWRITING with conditional axioms having more complex conditions. A challenge would be to implement automatic translators for each of these cases.

In other direction, we intend to define a tactic that performs implicit induction reasoning as an alternative to the existing explicit induction techniques for validating inductive properties. In this way, Coq (and other similar proof assistants) would be able to automatically execute multiple induction steps and manage more conveniently mutually defined functions.

References

1. Baader, F., Nipkow, T.: *Term Rewriting and All That*. Cambridge University Press, Cambridge (1998)
2. Barthe, G., Courtieu, P.: Efficient reasoning about executable specifications in Coq. In: *Theorem Proving in Higher Order Logics*, p. 64 (2002)
3. Barthe, G., Stratulat, S.: Validation of the JavaCard platform with implicit induction techniques. In: Nieuwenhuis, R. (ed.) *RTA 2003*. LNCS, vol. 2706, pp. 337–351. Springer, Heidelberg (2003)
4. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) *LPAR 2007*. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
5. Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. *Journal of Automated Reasoning* 14(2), 189–235 (1995)
6. Brotherston, J.: *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh (November 2006)
7. Bundy, A., van Harmelen, F., Horn, C., Smaill, A.: The Oyster-Clam system. In: Stickel, M.E. (ed.) *CADE 1990*. LNCS, vol. 449, pp. 647–648. Springer, Heidelberg (1990)
8. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Certification of automated termination proofs. In: *Frontiers of Combining Systems*, pp. 148–162 (2007)
9. Courant, J.: *Proof reconstruction*. Research Report RR96-26, LIP, Preliminary version (1996)
10. Dixon, L.: *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh (2005)
11. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science* 351(2), 167–183 (2006)
12. Kaliszzyk, C.: *Validation des preuves par récurrence implicite avec des outils basés sur le calcul des constructions inductives*. Master’s thesis, Université Paul Verlaine - Metz (2005)
13. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: *seL4: Formal verification of an operating-system kernel*. *Communications of the ACM* 53(6), 107–115 (2010)
14. Lindblad, F., Benke, M.: A tool for automated theorem proving in Agda. In: Filiâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) *TYPES 2004*. LNCS, vol. 3839, pp. 154–169. Springer, Heidelberg (2006)

15. Nahon, F., Kirchner, C., Kirchner, H., Brauner, P.: Inductive proof search modulo. *Annals of Mathematics and Artificial Intelligence* 55(1-2), 123–154 (2009)
16. Pientka, B., Kreitz, C.: Automating inductive specification proofs in NuPrL. *Fundamenta Informaticae* 1(2), 182–209 (1998)
17. The Spike prover, <http://code.google.com/p/spike-prover>
18. Rouached, M., Godart, C.: Reasoning about events to specify authorization policies for web services composition. In: *ICWS, IEEE International Conference on Web Services*, pp. 481–488. IEEE Computer Society, Los Alamitos (2007)
19. Rusinowitch, M., Stratulat, S., Klay, F.: Mechanical verification of an ideal incremental ABR conformance algorithm. *J. Autom. Reasoning* 30(2), 53–177 (2003)
20. Stratulat, S.: A general framework to build contextual cover set induction provers. *J. Symb. Comput.* 32(4), 403–445 (2001)
21. Stratulat, S.: Automatic ‘Descente Infinie’ induction reasoning. In: Beckert, B. (ed.) *TABLEAUX 2005. LNCS (LNAI)*, vol. 3702, pp. 262–276. Springer, Heidelberg (2005)
22. Stratulat, S.: Combining rewriting with Noetherian induction to reason on non-orientable equalities. In: Voronkov, A. (ed.) *RTA 2008. LNCS*, vol. 5117, pp. 351–365. Springer, Heidelberg (2008)
23. Stratulat, S., Demange, V.: Validating implicit induction proofs using certified proof environments. In: *Poster Session of 2010 Grande Region Security and Reliability Day, Saarbrücken (March 2010)*
24. The Coq Development Team. *The Coq reference manual - version 8.2* (2009), <http://coq.inria.fr/doc>
25. Wilson, S., Fleuriot, J., Smalls, A.: Inductive proof automation for Coq. In: *Coq Workshop (to appear 2010)*

Author Index

- Arenas, Alvaro E. 28
Asirelli, Patrizia 43
Autexier, Serge 59
Aziz, Benjamin 28
- Baier, Christel 1
Beek, Maurice H. ter 43
Bicarregui, Juan 28
Boström, Pontus 74
Boucheneb, Hanifa 89
- Daum, Matthias 105
Dimovski, Aleksandar 121
Dios, Javier de 305
- Eggers, Andreas 168
- Faber, Johannes 136, 152
Fantechi, Alessandro 43
Fitzgerald, John 12
Fränzle, Martin 168
Frappier, Marc 245
- Gervais, Frédéric 245
Gnesi, Stefania 43
- Hülsbusch, Mathias 183
- Ihlemann, Carsten 152
Imine, Abdessamad 89
- Jacobs, Swen 152
Joshi, Rajeev 27
- Kolahdouz-Rahimi, Shekoufeh 199
König, Barbara 183
- Laibinis, Linas 275
Laleau, Régine 245
- Lano, Kevin 199
Larsen, Peter Gorm 12
Lüth, Christoph 59
- Marrone, Stefano 215
Mateescu, Radu 229
Milhau, Jérémy 245
Montenegro, Manuel 305
- Najem, Manal 89
- Papa, Camilla 215
Peña, Ricardo 305
Pierce, Ken 12
- Rensink, Arend 183
- Salaün, Gwen 229
Schirmer, Norbert W. 105
Schmidt, Mareike 105
Schneider, Steve 260
Semenyak, Maria 183
Sofronie-Stokkermans, Viorica 152
Soltenborn, Christian 183
Steffen, Martin 290
Stratulat, Sorin 320
- Tarasyuk, Anton 275
Teige, Tino 168
Tran, Thi Mai Thuong 290
Treharne, Helen 260
Troubitsyna, Elena 275
- Verhoef, Marcel 12
Vittorini, Valeria 215
- Wehrheim, Heike 183, 260
Wilson, Michael D. 28
Wolff, Sune 12