Gilles Barthe
Alberto Pardo
Gerardo Schneider (Eds.)

LNCS 7041

Software Engineering and Formal Methods

9th International Conference, SEFM 2011
Montevideo, Uruguay, November 2011
Proceedings

Springer

Lecture Notes in Computer Science 7041

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

Gilles Barthe Alberto Pardo
Gerardo Schneider (Eds.)

Software Engineering and Formal Methods

9th International Conference, SEFM 2011
Montevideo, Uruguay, November 14-18, 2011
Proceedings

Springer

Volume Editors

Gilles Barthe
Fundación IMDEA Software, Facultad de Informática (UPM)
Campus Montegancedo, 28660 Boadilla del Monte, Madrid, Spain
E-mail: gilles.barthe@imdea.org

Alberto Pardo
Universidad de la República
Facultad de Ingeniería, Instituto de Computación
Julio Herrera y Reissig 565 - Piso 5, 11300 Montevideo, Uruguay
E-mail: pardo@fing.edu.uy

Gerardo Schneider
Chalmers | University of Gothenburg
Department of Computer Science and Engineering
Kunskapsgatan 3, 41756 Gothenburg, Sweden
and
University of Oslo, Department of Informatics
PB 1080 Blindern, 0316 Oslo, Norway,
E-mail: gersch@chalmers.se

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24689-0 e-ISBN 978-3-642-24690-6
DOI 10.1007/978-3-642-24690-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011938201

CR Subject Classification (1998): D.2.4, D.2, F.3, D.3, D.1.5, C.2, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM 2011) held on November 14–18, 2011 in Montevideo, Uruguay, under the auspices of the Facultad de Ingeniería (InCo), Universidad de la República, Uruguay. The aim of SEFM is to bring together practitioners and researchers from academia, industry and government to advance the state of the art in formal methods, to scale up their application in software industry and to encourage their integration with practical engineering methods.

The Program Committee of SEFM 2011 received 105 abstracts and 85 full submissions from all over the world. We would like to thank all authors for submitting their papers. Each paper was reviewed by at least three reviewers. Based on the review reports and intensive discussions conducted electronically, the Program Committee selected 22 regular papers, 2 tool papers and 1 short paper (acceptance rate around 29%), for inclusion in this volume. We would like to thank the Program Committee members and all reviewers for their efforts in the selection process.

Besides the regular session, the conference held a special track devoted to "Modelling for Sustainable Development", organized by Antonio Cerone from UNU-IIST. The special track received 7 submissions and accepted 5 papers, which are included in this volume.

In addition to the contributed papers, the conference program included four keynote speakers: Holger Hermanns (Saarland University, Germany), Mike Hinchey (LERO, Ireland) and Daniel Le Métayer (INRIA, France) for the main track, and Matteo Pedercini (Millennium Institute, USA) for the special track.

As is the tradition with SEFM, the conference was preceded by a graduate school and tutorials. The school courses were offered by: Dave Clarke (Katholieke Universiteit Leuven, Belgium), Klaus Havelund (Jet Propulsion Laboratory, USA), Yassine Lakhnech (University Joseph Fourier / VERIMAG, France), Martin Leucker (University of Lübeck, Germany), Davide Sangiorgi (INRIA, France, and University of Bologna, Italy), and Tarmo Uustalu (Tallin University of Technology, Estonia).

The tutorials were offered by: Gustavo Betarte (Universidad de la República, Uruguay), Pedro D'Argenio (Universidad Nacional de Córdoba, and CONICET, Argentina), Dilian Gurov (KTH, Sweden), Sebastián Uchitel (Imperial College London, UK, and Universidad de Buenos Aires, Argentina), and Santiago Zanella (IMDEA Software, Spain).

We are grateful to the invited speakers, tutorialists, and lecturers for accepting our invitation to address the conference or lecture at the school.

We also would like to thank the members of the Steering Committee and the Organizing Committee as well as several other people whose efforts contributed to making the conference a success. In particular, we would like to thank Carlos Luna and Luis Sierra (Universidad de la República, Uruguay) for helping with the local organization.

August 2011

Gilles Barthe
Alberto Pardo
Gerardo Schneider

Conference Organization

Conference Committees

Conference Chair	Alberto Pardo, Universidad de la República (Uruguay)
Program Chairs	Gilles Barthe, IMDEA Software Institute (Spain) Gerardo Schneider, Chalmers | University of Gothenburg (Sweden), and University of Oslo (Norway)
Local Organization	Carlos Luna, Universidad de la República (Uruguay) Alberto Pardo, Universidad de la República (Uruguay) Luis Sierra, Universidad de la República (Uruguay)
Special Track Chair	Antonio Cerone, UNU-IIST (China)

Steering Committee

Manfred Broy	TU Munich (Germany)
Antonio Cerone	UNU-IIST (China)
Mike Hinchey	LERO (Ireland)
Mathai Joseph	TRDDC (India)
Zhiming Liu	UNU-IIST (China)
Andrea Maggiolo-Schettini	University of Pisa (Italy)

Program Committee

Bernhard K. Aichernig	TU Graz
Luis Barbosa	Universidade do Minho
Gilles Barthe	IMDEA Software Institute
Thomas Anung Basuki	Parahyangan Catholic University
Alexandre Bergel	University of Chile
Gustavo Betarte	Universidad de la República
Ana Cavalcanti	University of York
Pedro R. D'Argenio	Universidad Nacional de Córdoba - CONICET
Van Hung Dang	Vietnam National University
George Eleftherakis	University of Sheffield
José Luiz Fiadeiro	University of Leicester

Martin Fränzle	Carl von Ossietzky Universität Oldenburg
Stefania Gnesi	ISTI-CNR
Rob Hierons	Brunel University
Paola Inverardi	Università dell'Aquila
Jean-Marie Jacquet	University of Namur
Tomasz Janowski	UNU-IIST
Jean-Marc Jezequel	University of Rennes 1 and INRIA
Joseph Kiniry	IT University of Copenhagen
Paddy Krishnan	Bond University
Martin Leucker	University of Lübeck
Xuandong Li	Nanjing University
Peter Lindsay	The University of Queensland
Antónia Lopes	University of Lisbon
Nenad Medvidovic	University of Southern California
Mercedes G. Merayo	Universidad Complutense de Madrid
Stephan Merz	INRIA Nancy
Madhavan Mukund	Chennai Mathematical Institute
Martin Musicante	Universidade Federal do Rio Grande do Norte
César Muñoz	NASA
Mizuhito Ogawa	Japan Advanced Institute of Science and Technology
Olaf Owe	University of Oslo
Gordon Pace	University of Malta
Ernesto Pimentel	University of Malaga
Sanjiva Prasad	Indian Institute of Technology Delhi
Anders Ravn	Aalborg University
Leila Ribeiro	Universidade Federal do Rio Grande do Sul
Augusto Sampaio	Universidade Federal de Pernambuco
Gerardo Schneider	Chalmers | University of Gothenburg, and University of Oslo
Sebastian Uchitel	Imperial College London and Universidad de Buenos Aires
Willem Visser	Stellenbosch University
Sergio Yovine	CONICET - Universidad de Buenos Aires

Special Track Program Committee

Roberto Barbuti	University of Pisa (Italy)
Antonio Cerone	UNU-IIST (China)
Elsa Estevez	UNU-IIST (China)
Peter Haddawy	UNU-IIST (China)
Siu-Wai Leung	University of Macau (China)
Dora Marinova	Curtin University (Australia)
Paolo Milazzo	University of Pisa (Italy)
Ion Petre	Åbo Akademi University (Finland)

Weishuang Qu	Millennium Institute (USA)
Dave Robertson	University of Edinburgh (UK)
Siraj Shaikh	Coventry University (UK)
Michael Sonnenschein	University of Oldenburg (Germany)
Hefeng Tong	Institute of Scientific and Technical Information of China (China)
Jianhong Wu	York University (Canada)
Shaofa Yang	Chinese Academy of Sciences, SIAT (China)

Additional Reviewers

Abraham, Erika	Almeida, José Bacelar	Asirelli, Patrizia
Baliosian, Javier	Baltazar, Pedro	Bocchi, Laura
Boronat, Artur	Brandán Briones, Laura	Bu, Lei
Buntrock, Gerhard	Bøgholm, Thomas	Cadavid Gómez, Juan-José
Cajueiro, Adalberto	Calegari, Daniel	Castro, Pablo
Costa, Umberto	Crole, Roy	Cubo, Javier
Dang Duc, Hanh	Decker, Normann	Demange, Delphine
Dolques, Xavier	Fantechi, Alessandro	Filliâtre, Jean-Christophe
Fontaine, Pascal	Forejt, Vojtech	Francalanza, Adrian
Giménez, Eduardo	Goodloe, Alwyn	Hagen, George
Hauptmann, Benedikt	Hungar, Hardi	Iyoda, Juliano
Jöbstl, Elisabeth	Kromodimoeljo, Sentot	Kunz, César
Lal, Akash	Legay, Axel	Li, Xin
Martins Moreira, Anamaria	Martins, Francisco	Massoni, Tiago
Mehta, Farhad	Mori, Marco	Nakajima, Shin
Narkawicz, Anthony	Nguyen, Tang	Nowotka, Dirk
Ogata, Kazuhiro	Okikka, Joseph	Owens, Scott
Pelozo, Silvia	Pham Ngoc, Hung	Prisacariu, Cristian
Ramalingam, Ganesan	Ramanujam, R.	Rinetzky, Noam
Rodrigues, Nuno	Rosa, Cristián	Rossi, Matteo
Sannier, Nicolas	Schapachnik, Fernando	Schlatte, Rudolf
Seki, Hiroyuki	Shankar, Natarajan	Siminiceanu, Radu
Smith, Graeme	Spalazzese, Romina	Srba, Jiri
Sternagel, Christian	Stocks, Phil	Stümpel, Annette
Swaminathan, Mani	Teige, Tino	Thoma, Daniel
Thuong Tran, Thi Mai	Vallespir, Diego	Vicario, Enrico
Vighio, Saleem	Vorobyov, Kostyantyn	Wang, Linzhang
Winter, Kirsten	Zhao, Jianhua	

Sponsors

- ANII (Agencia Nacional de Investigación e Innovación, Uruguay)
- CSIC (Comisión Sectorial de Investigación Científica, Universidad de la República, Uruguay)
- PEDECIBA Informática (Programa de Desarrollo de las Ciencias Básicas, Uruguay)

Table of Contents

Keynote Talks

Regular Papers

Short Papers

Tool Papers

Special Track: "Modelling for Sustainable Development"

Formal Methods in Energy Informatics

Holger Hermanns

Dependable Systems and Software, Universität des Saarlandes, Saarbrücken, Germany
http://d.cs.uni-saarland.de/hermanns/

Abstract. The European electricity market is rapidly evolving towards a decentralized structure, not only because of climatical and political circumstances. With the foreseeable depletion of fossile energy sources this trend is expected to catch momentum also on other continents. The increase of production based on renewable energy implies drastically higher fluctuations in available electricity. The resulting mathematical problem, stochastic electricity balancing, has many facets where quantitative formal methods provide a promising foundation to develop IT-supported strategies to counteract this problem.

The European electricity supply systems are rapidly evolving towards a situation in which not only the consumer behavior, but also the producer behavior must be considered as a stochastic process. This is a direct consequence of small and medium scale renewable energy plants deployed massively, together with the fact that sun intensity and wind speed are uncontrollable. This asks for novel methods to control and manage electricity networks in a decentralized way. The core objective is to continuously match production and consumption of electricity across networks. If both do not match, this impacts the frequency of the supplied power, and this frequency skew in fact serves as a limited buffer for misbalanced production and consumption.

Traditionally, the balance is maintained by the electricity producers on the basis of periodic (weekly, daily, hourly, 15 min) predictions of the anticipated consumption. The real-time match of production and consumption is obtained by dedicated power plants and control loops, which continuously supervise and stabilize the frequency at 50Hz (in Europe). These mechanisms can buffer about 10% of the peak electricity consumption.

This traditional approach however is based on the assumption that the production is a determinstc and a controllable process. Both assumptions are invalid in the future. The resulting challenge, the stochastic energy balancing problem [1], is the problem of decision making to keep the consumption and production of electricity within very tight bounds, where both consumption and production exhibit stochastic behavior. This problem induces a set of principal requirements for future modeling and analysis techniques and supporting tools needed to study, predict and guarantee behavioral properties of electrical energy networks. Quanitative formal methods can play a distinguished role in attacks to solve the problem. They need to combine elements from concurrency theory,

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 1–2, 2011.

stochastic process theory, differential equations or inclusions, with methods from computer aided verification. The aim is to arrive at IT-supported approaches to counter the stochastic energy balancing problem. The current focus of major European electricity producer aims at making the consumption more controllable, to be able to buffer the uncontrollable fluctuations in production at the consumer side. These cooperative strategies of producers, consumers and infrastructure providers need advanced software engineering, modeling and analysis techniques for behavioral properties of electricity networks.

Reference

1. Hermanns, H., Wiechmann, H.: Future Design Challenges for Electric Energy Supply. In: Proceedings of the 14th IEEE International Conference on Emerging Technologies & Factory Automation, pp. 1310–1317 (2009)

Formal Methods as a Link between Software Code and Legal Rules

Daniel Le Métayer

INRIA Grenoble Rhône-Alpes
France
Daniel.Le-Metayer@inria.fr

Abstract. The rapid evolution of the technological landscape and the impact of information technologies on our everyday life raise new challenges which cannot be tackled by a purely technological approach. Generally speaking, legal and technical means should complement each other to reduce risks for citizens and consumers : on one side, laws (or contracts) can provide assurances which are out of reach of technical means (or cope with situations where technical means would be defeated); on the other side, technology can help enforce legal and contractual commitments. This synergy should not be taken for granted however, and if legal issues are not considered from the outset, technological decisions made during the design phase may very well hamper or make impossible the enforcement of legal rights. But the consideration of legal constraints in the design phase is a challenge in itself, not least because of the gap between the legal and technical communities and the difficulties to establish a common understanding of the concepts at hand. In this paper, we advocate the use of formal methods to reduce this gap, taking examples in areas such as privacy, liability and compliance.

Keywords: regulation, law, legal, liability, accountability, privacy, compliance, formal model, causality.

1 Motivation

The rapid evolution of the technological landscape and the impact of information and communication technologies (ICT) on our everyday life raise new challenges which cannot be tackled by a purely technological approach [Poullet - 2006]. For example, the protection of privacy rights on the Internet or in pervasive computing environments is by definition multidimensional and requires expertise from disciplines such as social sciences, economics, ethics, law and computer science [Rouvroy - 2008]. Other examples of the ever-growing intermingling of ICT and law include electronic commerce, digital rights management (DRM), software contracts, social networks, forensics, cybercrime, Internet regulation, e-government, and e-justice - and this list is far from limitative. As far as research is concerned however, there are still very few links between the ICT and law communities. This situation is unfortunate considering the importance of the interests at stake (not only in economic terms but also for society as a whole).

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 3–18, 2011.

Starting from this observation, the general goal of the research outlined here is to contribute, in partnership with lawyers, to the development of new approaches and methods for a better integration of technical and legal instruments.

In practice, the interactions between ICT and law take various forms and go in both directions [Le Métayer, Rouvroy - 2008]:

- ICT "objects" are, as any other objects, "objects of law": on one hand, there is no reason why new technologies and services should escape the realm of law; on the other hand, it may be the case that existing regulations need to be adapted to take into account the advent of new, unforeseen technological developments (e.g. certain provisions of privacy regulations become inapplicable in a pervasive computing context, intellectual property laws are challenged by the new distribution modes of electronic contents). Understanding precisely when this is the case and how regulations should evolve to cope with the new reality is a complex "technico-legal" issue with potential impact on both disciplines.
- ICT can also provide new enforcement mechanisms and tools for the benefit of the law: Privacy Enhancing Technologies (PETs) [Goldberg - 2007] help reduce privacy threats , certified tools can be provided to support electronic signature, DRM technologies are supposed to "implement" legal provisions and contractual commitments, computer logs can be used as evidence in courts, etc. At another level, data mining or knowledge management systems can be applied to the extraction of relevant legal cases or the formalization of legal reasoning.

Generally speaking, legal and technical means should complement each other to reduce risks and to increase citizens' and consumers' trust in ICT : on one side, laws (or contracts) can provide assurances which are out of reach of technical means (or cope with situations where technical means would be defeated); on the other side, technology can help enforce legal and contractual commitments [Le Métayer - 2010c]. These interactions are quite subtle however and this synergy should not be taken for granted: if legal issues are not considered from the outset, technological decisions made during the design phase may very well hamper or make impossible the enforcement of legal rights; similarly, new regulations or contracts drafted without proper consideration for the possibilities offered by the technology are bound to remain ineffective. But the consideration of legal constraints in the design phase of an IT system is a challenge in itself, not least because of the gap between the law and computer science communities and the difficulties to establish a common understanding of the concepts at hand. In this paper, we argue that formal methods, both for fundamental and practical reasons, can help reduce this gap (Section 2). We illustrate the feasibility and the interest of this approach through examples in software liability (Section 3), privacy (Section 4), and compliance (Section 5). We then identify further challenges for both disciplines (Section 6), showing that the link between ICT and law is a fruitful research area both for computer scientists and for lawyers, and we conclude with a discussion on a methodology for interdisciplinarity (Section 7).

2 Formal Methods as a Link between ICT and Law

Beyond their many differences, ICT and law share a strong emphasis on formalism. This commonality is not without reason: in both cases, formalism is a way to avoid ambiguity and to provide the required level of rigour, transparency, and security. As an illustration, L. Fuller in his book "The morality of law" [Fuller - 1964] puts forward the following distinctive features of a legal system: (1) a set of rules (2) without contradiction, (3) understandable, (4) applicable, (5) predictable, (6) publicized and (7) legitimate. Even though they were obviously not proposed with such a comparison in mind, it is interesting to note that, among these features, the first five are also often used in computer science to characterize a good software specification and the sixth one can be related to the notion of open access to source code. The last one, legitimacy, is actually a key distinctive feature of legal normativity with respect to technical normativity. We come back to this critical issue in the conclusion.

As far as software is concerned, the fact that both disciplines refer to the word "code" is not insignificant and the exploration of the commonalities can be very fruitful - and not only from a theoretical perspective. Indeed, there are many situations where the frontier between the two notions seems to be blurring[1]. Just to take a few examples:

- Software contracts typically incorporate references to technical requirements or specifications which can be used, for example, to decide upon acceptance of the software by the customer or validity of an error correction request. In case of litigation, these specifications can also be used by the judge as they are part of the contract executed by the parties. In this perspective, the contract can thus be seen as an extension of the technical specification including further legal provisions such as intellectual property rights, warranty, and liability.
- The DRM technologies are supposed to implement legal provisions and contractual commitments about the use of digital content such as music or video.
- More and more transactions are performed on the basis of electronic contracts (SLA, or "Service Level Agreements" for web services, electronic software licenses, e-commerce contracts, etc.).

In fact, the convergence has developed so much that lawyers have expressed worries that "machine code" might more and more frequently replace "legal code", with detrimental effects on individuals. This topic has stirred up discussions in the legal community (see, for example, [Lessig - 2001], [Lessig - 2007] and [Reidenberg - 1998]) and is bound to remain active for quite a long time. Indeed, the implementation of contractual commitments by computer code raises a number of issues such as the lack of flexibility of automated tools, the potential inconsistency between computer code and legal code, the potential errors or

[1] Lawrence Lessig refers to East Coast Code and West Coast Code to denote respectively law and software code [Lessig - 2007].

flaws in the computer code itself, not to mention the legitimacy issue pointed out above.

In any event, the reality is that software code and legal provisions are increasingly intermingled, sometimes with complementary roles, sometimes in a fuzzy or conflicting relationship. It is also the case that legal provisions, just like software code, are assumed to meet specific goals or requirements. Just like software specifications, these requirements can be defined precisely, even formally (at least to a certain extent, because legal provisions must usually leave some room for interpretation by the judge) using dedicated logics (see, for example, [Farrell et. al. - 2005] and [Prisacariu, Schneider - 2007]). Based on this double observation, we argue that the first step for a fruitful and useful exploration of the relationships between legal provisions and software code is the definition of a formal framework for expressing the notions at hand, understanding them without ambiguity, and eventually relating or combining them. Stated in so general terms, one may wonder whether such an approach can really be turned into practice and if it can have any impact beyond theoretical considerations. In the next three sections, we show the feasibility of the approach through its application to three areas in which the link between law and technology is of prime importance, namely software liabilities, privacy and compliance.

3 Liability Issues in Software Engineering

As mentioned above, software contracts between professionals ("B2B contracts") typically include references to technical requirements or specifications which can be used, for example, to decide upon acceptance of the software by the customer or liability in case of failure of the system. It is often the case that these requirements are not stated very precisely though, which may lead to misunderstandings between the parties or potential conflicts between them during the execution of the contract.

The legal situation is often simpler, at least apparently, in typical licenses for "off the shelf" software, which usually include strong liability limitations or even exemptions of the providers for damages caused by their products. This situation does not favour the development of high quality software though, because software vendors do not have sufficient economic incentives to apply stringent development and verification methods (see, for example, [Anderson, Moore - 2009], [Berry - 2007] and [Ryan - 2003]). Indeed, experience shows that products tend to be of higher quality and more secure when the actors in position to influence their development are also the actors bearing the liability for their defects. In addition, the validity of contractual liability limitations and exemptions can sometimes be questioned. For example, most regulations provide specific protections to consumers which make such clauses invalid in B2C contracts. Even in B2B contracts, liability limitations are usually considered null and void when the party claiming the benefit of the clause has committed acts of intentional fault, wilful misrepresentation or gross negligence. Another case is the situation where the limitation would undermine an essential obligation of a party and would thus

introduce an unacceptable imbalance in the contract [Steer et. al. - 2011]. This situation is more difficult to assess though, and left to the appraisal of the judge who may either accept the limitation, consider it null, or even fix a different liability level.

Whether liability clauses are defined too vaguely or unequally with risks of being invalidated in court, they result in contracts with high legal uncertainties, which is not a desirable situation, neither for business nor for society in general. The usual argument to justify this situation is the fact that software products are too complex and versatile objects whose expected features (and potential defects) cannot be characterised precisely, and which thus cannot be treated as traditional (tangible) goods. Admittedly, this argument is not without any ground: it is well known that defining in an unambiguous, comprehensive and understandable way the expected behaviour of software systems is quite a challenge, not to mention the use of such a definition as a basis for a liability agreement. But the fact that specifying entire software systems and all associated liabilities is usually out of reach does not mean that the most significant scenarios and sources of liabilities cannot be identified and formally specified. Actually, specifying formally all liabilities would not even be a desirable goal. Usually, the parties wish to express as precisely as possible certain aspects which are of prime importance to them and prefer to state other aspects less precisely (either because it is impossible to foresee, when signing the contract, all the events that may occur or because they do not want to be bound by overly precise commitments).

To address this need, we have proposed a framework providing different levels of services which can be used by the parties depending on factors such as the economic stakes and the timing constraints for the drafting of the contract [Le Métayer et. al. - 2010a]:

1. The first level is a systematic (but informal) definition of liabilities based on a library of (parameterized) legal clauses [Steer et. al. - 2011].
2. The second level is the formal definition of liabilities. This formal definition can be more or less detailed and does not have to encompass all the liability rules defined informally. In addition, it does not require a complete specification of the software but only the properties relevant for the targeted liability rules.
3. The third level is the implementation of a log infrastructure or the enhancement of existing logging facilities to ensure that all the information required to establish liabilities will be available if a claim is raised and will be trustable to be used as evidence for the case.
4. The fourth level is the implementation of a log analyser to assist human experts in the otherwise tedious and error-prone log inspection task.

Each level contributes to further reducing the uncertainties with respect to liabilities, and the parties can decide to choose the level commensurate with the risks linked to potential failures of the system.

The keystone of the formal specification of liabilities is the notion of "claim property". Basically, claim properties represent the grounds for the claims: they

correspond to failures of the system as experienced by the users. In practice, for them to give rise to liabilities, such failures should cause damages to the plaintiff, but damages are left out of the formal model. As an illustration, a claim property can express the fact that a signature application has sent to the server a message indicating that a given user has signed a specific document (identified by a stamp) when the user has never been presented any document with this stamp [Le Métayer et. al. - 2011]. Claims can be expressed as trace properties using temporal or predicate logics. The choice of the language of properties does not have any impact on the overall process but it may make some of the technical steps, such as the log analysis, more or less difficult.

The liabilities arising under a given contract can be expressed as a function mapping claims and traces onto sets of (liable) parties. One way to define the liability function is to specify typical faults in the execution of the components and to associate a set of liable parties with each claim and combination of faults. Faults can be expressed in the same trace property language as the claims. Another possibility is to define a causality relationship between the occurrences of certain types of faults and failures [Goessler et. al. - 2010]. Causality has been studied for a long time in computer science [Lamport - 1978], but with quite different perspectives and goals. In the distributed systems community, causality is seen essentially as a temporal property. In [Goessler et. al. - 2010], we have defined several variants of logical causality allowing us to express the fact that an event e_2 (e.g. a failure) would not have occurred if another event e_1 had not occurred ("necessary causality") or the fact that e_2 could not have been avoided as soon as e_1 had occurred ("sufficient causality"). We have shown that these causality properties are decidable and proposed trace analysis procedures to establish them.

Another key design choice is the distribution of the log files themselves. Indeed, recording log entries on a device controlled by an actor who may be involved in a claim for which this log would be used as evidence may not be acceptable to the other parties. In [Le Métayer et. al. - 2010b], we have introduced a framework for the specification of log architectures and proposed criteria to characterize "acceptable log architectures". These criteria depend on the functional architecture of the system itself and the potential claims between the parties. They can be used to check that a log architecture is appropriate for a given set of potential claims and to suggest improvements to derive an acceptable log architecture from a non-acceptable log architecture. On the formal side, we have shown that, for a given threat model, the logs produced by acceptable log architectures can be trusted as evidence for the determination of liabilities: technically speaking, any conclusive evaluation of a claim based on these logs produces the same verdict as the evaluation of the claim based on the sequence of real events.

As far as the log analysis itself is concerned, we have proposed a formal specification of the analyser using the B method in [Mazza et. al. - 2010] and we have shown the correctness of an incremental analysis process. This result makes it possible to build upon the output of a first analysis to improve it by considering additional logs or further properties.

The overall approach has been applied to several representative case studies: an electronic signature application on a mobile phone [Le Métayer et. al. - 2011], a distributed hotel booking service [Le Métayer et. al. - 2010b] and a cruise control system [Goessler et. al. - 2010].

4 Privacy

Another area where technical and legal issues become more and more entangled is privacy. Even in countries where they benefit from apparently strong legal protections, many citizens feel that information technologies have invaded so much of their life that they no longer have suitable guarantees about their privacy. Indeed, the fact that the massive use of information technologies is the source of new risks for privacy is unquestionable. Many data communications already take place nowadays on the Internet without the users' notice and the situation is going to get worse with the advent of "ambient intelligence" or "pervasive computing". One of the most challenging issues in this context is the compliance with the "informed consent" principle, which is a pillar of most data protection regulations. For example, Article 7 of the EU Directive 95/46/EC states that "personal data may be processed only if the data subject has unambiguously given his consent" (unless waiver conditions are satisfied, such as the protection of the vital interests of the subject). In addition, this consent must be informed in the sense that the controller must provide sufficient information to the data subject, including "the purposes of the processing for which the data are intended".

Technically speaking, the consent of the subject can be implemented through a "privacy policy" which should reflect his choices in terms of disclosure and use of personal data. We have proposed an implementation of privacy policies through "Privacy Agents", dedicated software components acting as "surrogates" of the subjects and managing their personal data on their behalf. The subject can define his privacy requirements once and for all, with all information and assistance required, and then rely on his Privacy Agent to implement these requirements faithfully. However, this technical solution raises a number of questions from the legal side: for example, to what extent should a consent delivered via a software agent be considered as legally valid? Are current regulations flexible enough to accept this kind of delegation to an automated system? Can the Privacy Agent be "intelligent" enough to deal with all possible situations? Should subjects really rely on their Privacy Agent and what would be the consequences of any error (bug, misunderstanding...) in the process?

In order to shed some light on these legal issues, we have focused on three main aspects of consent : its legal nature (unilateral versus contractual act), its essential features (qualities and defects) and its formal requirements. In a second stage, we have drawn the lessons learned from this legal analysis to put forward design choices ensuring that Privacy Agents can be used as valid means to deliver the consent of the data subject [Le Métayer, Monteleone - 2009]. Several kinds of Privacy Agents have been proposed (Subject Agents, Controller Agents and

Auditor Agents) and the roles of the different actors involved in the process have been defined precisely. Privacy policies themselves can be expressed in a restricted (pattern based) natural language. In order to avoid ambiguities in the expression of the policies, a mathematical semantics of the privacy language has been defined. This mathematical semantics characterizes precisely the expected behaviour of the Privacy Agents (based on the privacy policies defined by their users) in terms of compliant execution traces.

This work is an illustration of the privacy by design approach which is often praised by lawyers as well as computer scientists as an essential step towards a better privacy protection [Le Métayer - 2010d]. The general philosophy of privacy by design is that privacy should not be treated as an afterthought but rather as a first-class requirement during the design of IT systems; in other words, designers should have privacy in mind from the moment they define the features and architecture of a system and throughout its life cycle. The privacy by design approach has been applied in different areas such as electronic health record systems [Anciaux et. al. - 2008], location based services [Kosta et. al. - 2008], electronic traffic pricing ([De Jonge, Jacobs - 2008], [Balash et. al. - 2010]). More generally, it is possible to identify a number of core principles that are widely accepted and can form a basis for privacy by design. For example, the Organization for Economic Co-operation and Development (OECD) has put forward the following principles [OECD - 1980]:

- The collection limitation principle: lawful collection of data with the "knowledge or consent" of the data subject.
- The purpose specification and use limitation principles: specification of the purposes, collection and use limited to those purposes.
- The data quality principle: accuracy of the data, relevance for the purpose and minimality.
- The security principle: implementation of reasonable security safeguards to avoid "unauthorised access, destruction, use, modification or disclosure of data".
- The openness and individual participation principles: right to obtain information about the personal data collected, "to challenge" the data and, if the challenge is successful, to have the data "erased, rectified, completed or amended".
- The accountability principle: data controllers should be accountable for complying with these principles.

These principles have inspired a number of privacy regulations. They are also very much in line with the European Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data[2].

One must admit however that the take-up of privacy by design in the ICT industry is still rather limited. This situation is partly due to legal and

[2] The latter however puts more emphasis on the explicit consent of the subject.

economic reasons: as long as the law does not impose binding commitments[3], ICT providers and controllers do not have sufficient incentives to invest into privacy by design. But part of the reason is also technical: computer scientists have devised a number of privacy enhancing tools, but no general methodology is available to integrate them in a consistent way to ensure suitable privacy properties. In the same way as the use of cryptography is by no means a guarantee of security, the use of privacy enhancing tools does not bring by itself a guarantee of privacy. The next challenge in this area is thus to go beyond individual cases and to establish sound foundations and methodologies for privacy by design [Le Métayer - 2010d].

As a first step in this direction, we have proposed a formal framework for the implementation of the data minimization principle which stipulates that the collection should be limited to the data strictly necessary for the purpose. This framework allows us to define:

- The service to be performed, expressed as a set of equations characterizing the values to be computed.
- The actors involved.
- The requirements of each actor, defined as constraints on the variables used in the equations. Typical requirements may express the fact that a given value should not be collected or that it should be collected only in a specific form (aggregated, sampled, ciphered, etc.).

An operational semantics defines the effect of each action on the state of the actors and the underlying threat model (possibilities of tampering with variables, properties of cryptographic commitments or secure components, etc.). An inference system based on this operational semantics allows us to derive properties of the variables such as, for example, the fact that an actor can obtain enough knowledge to identify an error (or potential fraud) in the computation of a variable. This inference system can be used to explore the design space systematically, for example to infer architectures meeting the requirements of the parties (e.g. limited disclosure on side the data subject and ability to discover certain types of frauds on the side of the data controller) or to detect conflicting requirements.

Even if much work has still to be done in this area, as suggested in Section 6, we believe that the added value of the formal approach for privacy by design can be decisive: in addition to the usual benefits (precise definitions of assumptions and requirements, detection of inconsistencies, verification), it can be used to provide designers with practical means for the systematic exploration of the available options and for the justification of their architectural choices.

5 Compliance

Compliance is yet another legal area where the use of formal methods can be very beneficial. Nowadays, organizations have to comply with a growing

[3] This situation might change in Europe though, with the revision of the European Directive 95/46/EC which is currently under discussion.

number of legal rules stemming from law, regulations, corporate policies or contractual agreements. These rules have a potential impact on all their activities and breaches may lead to different types of damages, including financial losses, lawsuits, competitive disadvantages and disrepute. But manual compliance enforcement or verification are error prone and tend to exceed the capacity of most organizations. IT systems, even if they cannot provide the full answer to this complex issue, can help organizations in the management and monitoring of their obligations.

To address this need, we have proposed a framework based on a formalism called FLAVOR[4] [Thion - 2011] which provides the following combination of features:

- Contrary to duty obligations [Prakken - 1996]: a contrary to duty obligation consists of a primary obligation and an alternative obligation which becomes effective when (and if) the primary obligation is breached. Contrary to duty obligations are useful to express penalty clauses in contracts as well as compensations and sanctions for breaches of legal rules.
- Combinations of temporal and deontic modalities: one of the most pervasive characteristics of legal rules is the interaction between temporal ("always", "eventually") and deontic ("obligatory", "prohibited") modalities [Pace, Schneider - 2009]. This interaction clearly appears in constructions such as "shall ... within ... days after ...", or "must ... within ...". Actually most obligations or prohibitions come with a deadline which may be defined by a fixed date, by a delay or by a specific event.
- Conditions and contexts: legal rules are generally expressed as abstract and general statements intended to be applied in a variety of circumstances. To this aim, the wording of a legal rule generally distinguishes the effect of the rule (action to be performed or prevented) and its context of application. The context of application typically involves parameters and data (e.g. price, reference number, time, ...) related to specific events.

We have defined a semantics for the language which is suitable for the implementation of an auditing tool and which avoids the paradoxes and counter-intuitive meanings often arising in modal logics. Based on this semantics, we have provided criteria for analysing obligations and defined a strength ordering which can be used to reason on contractual clauses. The framework has been illustrated with typical business contracts and privacy policy rules.

6 Further Challenges

The contributions sketched in the previous sections have been presented here only for illustrative purposes, to show that the use of formal methods as a link between law and software code is not a purely speculative idea. Needless to say, much work remains to be done, not only in the application areas mentioned here, but also more generally on the interactions between law and ICT.

[4] Formal Language for A posteriori Verification Of legal Rules.

The notion of causality, for example, is extremely rich and complex, and it represents in itself a very fruitful area for further research. First, it would be interesting to express causality in a more abstract way, independently of the underlying computation and communication models, and to establish precise links with related notions in dependability, diagnosis and security. The study of the correspondence between formal characterisations and legal definitions of causality is obviously another area for further work. To this respect, it would also be interesting to introduce probabilities in the formal framework in order to reflect certain interpretations of causality in the legal sense, the differences between several causes being often considered with respect to their effects on the likeliness of the occurrence of the damage [Busnelli et. al. - 2005].

As far as compliance is concerned, a number of key issues have already been investigated but still require further work [Pace, Schneider - 2009], especially to ensure that formal models are consistent both with the legal views and with the practical constraints that organizations have to face [Governatori et. al. - 2006]. Among these issues, we should mention the possibility to detect conflicts between obligations [Fenech at. al. - 2009], to verify statically the compliance of a system or to monitor its actions in order to ensure that no obligation can be violated. There are also other significant aspects of the problems faced by organizations that are not fully taken into consideration by previous work:

1. The first aspect is the dynamic nature of contracts. Most companies execute new contracts on a daily basis and these contracts usually have termination provisions. The execution of new contracts and their termination represent a substantial part of the difficulty and must be integrated in formal frameworks for obligations.
2. The second aspect is the fact that organizations have to cope with events which are not within their control and must take them into account before deciding to enter into new legal agreements.
3. The third aspect is the observation that, in practice, conflicts between obligations do not necessarily take the form of sheer contradictions: the situation is often more subtle, for example the consequences of a breach can be more or less significant; sometimes the conjunction of obligations does not lead to a contradiction but to a detrimental reduction of the choice space of the organization. Last but not least, following point 2 above, potential breaches may or may not be under the control of the organization.

Needless to say, privacy is also an area where a lot of difficult problems remain to be solved (and many others are bound to arise in the future). The main challenges in this area concern both privacy by design and privacy evaluation. First, much work remains to be done to turn privacy by design into practice, both from a formal point of view and from a methodological perspective. The work sketched in Section 4 is a first step in this direction, addressing the minimization principle, but other principles such as, for example, transparency or accountability require more attention. Indeed, Transparency Enhancing Tools (TETs) have been called for by lawyers (see, for example, [Hildebrandt - 2008] and [Hildebrandt - 2006]) but they have not yet become a reality. These tools

should provide ways for individuals to understand how their personal data (and, ideally, any data that can be used in a processing with potential effects on them) are collected, generated, managed, transferred, etc. The transparency requirement is of upmost importance in a context where information flows are growing dramatically and the data mining and inference techniques become more and more powerful.

The concept of accountability is already applied in certain areas such as the finance and public governance and it is likely to be included in the future version of the European Directive on Data Protection 95/46/EC currently under discussion. Accountability puts emphasis on "how responsibility is exercised and making it verifiable". Technically, it involves at least two dimensions: transparency (making processing visible) and security (in the sense of integrity and non repudiation of the accountability data). More generally, it is a multi-faceted notion, involving social, legal and political aspects. The relationship between accountability and privacy is also rather complex: accountability can be used to strengthen privacy rights (when it applies to the data controllers) but it can also represent a threat to privacy (when it applies to the data subjects, e.g. within financial transactions, or when it requires to record excessive amounts of personal data). More research is needed to clarify the technical definition of accountability and associated requirements (in line with the legal view), to ensure that accountability can go hand in hand with privacy, and to provide practical and trustworthy implementation methods and tools helping organizations to comply with the transparency and accountability requirements.

The definition of realistic and formally grounded measures of privacy is also a challenging task. Several proposals have been made to define relevant privacy metrics such as k-anonymity [Sweeney - 2002] or differential privacy [Dwork - 2006] but the problem remains open : some of these metrics do not necessarily measure a true protection level because they are vulnerable to certain types of attacks, while others provide guarantees which are difficult to reach in practice because they would result in unacceptable reductions of data utility. Also, it is not clear whether a single type of metric can be suitable for different application areas corresponding to varied needs and expectations in terms of privacy.

Needless to say, the above challenges concern the lawyers as well as the computer scientists. As an illustration, key notions of European data protection laws such as "personal data", "informed consent", "subject" or "controller" are challenged, if not made ineffective, by new technologies. Another illustration is the role of the consent of the subject in current data protection regulations. Some lawyers have expressed the view that putting too much stress on consent can lead to an exclusively individualistic view of privacy disregarding the collective value of privacy as a fundamental right. To avoid this drift, clear limitations should be placed on the legitimacy of consent: for example, certain data should be considered as inalienable and, when consent is authorized, it should come with strong requirements in terms of transparency to ensure that the subject really understands the consequences of his consent. But where to place the red

line and on which grounds are difficult questions, and, as suggested above, the effective implementation of transparency and consent delivery is also a challenge for computer scientists. In certain cases, the implementation of transparency can even create conflicts with the legal protection of intellectual property rights (e.g. with respect to profiling algorithms). Legal, social and technical dimensions are thus strongly intermingled and an interdisciplinary approach is required to make any progress on these topics.

7 Conclusion: Interdisciplinarity in Practice

In this paper, we have argued that the development of the new information society raises a number of challenges which require stronger collaboration between lawyers and computer scientists. But setting up this kind of interdisciplinary collaboration also represents a challenge in itself, especially when it concerns disciplines which have very different histories and cultures and have built very different modes of functioning (research development, assessment, collaborations, etc.). On one hand, each discipline should keep its criteria of excellence; on the other hand, disciplines should find together new ways of creating, communicating and evaluating research results. Needless to say, researchers in each discipline have also to overcome any misconception about the other discipline and accept points of views from "outsiders" questioning their own discipline. As shown by the pieces of work sketched in Sections 3, 4 and 5, this objective is not out of reach though. Drawing on the lessons of these projects, we believe that such an inderdisciplinary collaboration should be based on a precise methodology and it should include at least the following steps:

- The comparison of the terminologies and notions used in the different disciplines: often the same term is used in two disciplines with different meanings or intentions; vice versa, it also happens that the same notion is named in different ways in different disciplines. Indeed, there is no shortage of terms which may lead to confusion in discussions between lawyers and computer scientists (e.g. "causality", "accountability", "effectiveness", "proof", "security" , etc). The analysis of these shifts is a pre-requisite for mutual understanding; in addition it can shed new light on each discipline and help refining the underlying concepts.
- The comparison of the procedures, modes of operation in the different disciplines: for example how are the instruments conceived, how are they accepted, monitored, revised? How is their effectiveness defined and measured? Such a comparison, in addition to enhancing mutual understanding, can be a source of inspiration and improvement in each discipline. For example, the legal procedures can be a source of inspiration to provide a more transparent or democratic process for the development of new technologies, to devise technologies with "contradiction means" (possibility to bypass the procedure implemented by the tools). Vice versa, new ideas can come from the technology concerning criteria such as evolutivity or effectiveness.

– The study of the problems at hand in an iterative way where each discipline can bring its own analysis, views and findings before confronting them to the findings of the other disciplines and, based on this enlarged view, proposing a refined solution, which can be confronted again to the other ones.

Beyond research collaborations, the complex issues raised in this paper also question the relationships between the legal and technological normativities: how can the law face the "over-effectiveness" of technological norms and their opaque dissemination mode? How can the stability required by the legal systems adapt to the fast evolution of technologies? At what stage should the legal dimension be taken into account in the deployment of new technical infrastructures? How to introduce a mode of contestation or democratic debate in the elaboration of technological choices? Needless to say, these issues go beyond law and technology, they are by essence political, which should not come as a surprise considering the tremendous (and still growing) impact of information technologies on our everyday life [Jacobs - 2009].

Acknowledgments. This work was partly supported by the French ANR projects LISE under the grant ANR-07-SESU-007 and FLUOR under the grant ANR-07-SESU-005.

References

[Anciaux et. al. - 2008] Anciaux, N., Benzine, M., Bouganim, L., Jacquemin, K., Pucheral, P., Yin, S.: Restoring the patient control over her medical history. In: 21st IEEE International Symposium on Computer-Based Medical Systems, pp. 132–137. IEEE Computer Society, Los Alamitos (2008)

[Anderson, Moore - 2009] Anderson, R., Moore, T.: Information security economics – and beyond. Information Security Summit (IS2) (2009)

[Balash et. al. - 2010] Balasch, J., Rial, A., Troncoso, C., Geuens, C., Preneel, B., Verbauwhede, I.: PrETP: privacy-preserving electronic toll pricing. In: Proc. 19th USENIX Security Symposium (2010)

[Berry - 2007] Berry, D.M.: Abstract appliances and software: the importance of the buyer's warranty and the developer's liability in promoting the use of systematic quality assurance and formal methods. Scientific Literature Digital Library and Search Engine (2007), http://www.scientificcommons.org/42749418

[Busnelli et. al. - 2005] Busnelli, F.D., et al.: Principles of European tort law. Springer, Heidelberg (2005)

[Dwork - 2006] Dwork, C.: Differential privacy. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 1–12. Springer, Heidelberg (2006)

[Farrell et. al. - 2005] 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 (IJCIS) 14(2-3), 99–129 (2005)

[Fenech at. al. - 2009] Fenech, S., Pace, G., Schneider, G.: Automatic Conflict Detection on Contracts. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 200–214. Springer, Heidelberg (2009)

[Fuller - 1964] Fuller, L.L.: The morality of law. Yale University Press, New Haven (1964)

[Goessler et. al. - 2010] Gössler, G., Le Métayer, D., Raclet, J.-B.: Causality analysis in contract violation. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 270–284. Springer, Heidelberg (2010)

[Goldberg - 2007] Goldberg, I.: Privacy-enhancing technologies for the Internet III: Ten years later. In: Digital Privacy: Theory, Technologies, and Practices (2007)

[Governatori et. al. - 2006] Governatori, G., Milosevic, Z., Sadiq, S.W.: Compliance checking between business processes and business contracts. In: EDOC, pp. 221–232. IEEE, Los Alamitos (2006)

[Hildebrandt - 2006] Hildebrandt, M.: Profiling: from data to knowledge. DuD: Datenschutz und Datensicherheit 30(9), 548–552 (2006)

[Hildebrandt - 2008] Hildebrandt, M.: Profiling and the rule of law. Identity in the Information Society 1(1), 55–70 (2008)

[Jacobs - 2009] Jacobs, B.: Architecture is politics: security and privacy issues in transport and beyond. Data Protection in a Profiled World. Springer, Heidelberg (2010)

[De Jonge, Jacobs - 2008] De Jonge, W., Jacobs, B.: Privacy-friendly electronic traffic pricing via commits. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 143–161. Springer, Heidelberg (2009)

[Kosta et. al. - 2008] Kosta, E., Zibuschka, J., Scherner, T., Dumortier, J.: Legal considerations on privacy-enhancing location based services using PRIME technology. Computer Law and Security Report 24, 139–146 (2008)

[Lamport - 1978] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)

[Le Métayer, Rouvroy - 2008] Le Métayer, D., Rouvroy, A.: STIC et droit: défis, conflits et complémentarités. Interstices (2008), http://interstices.info/jcms/c_34521/stic--et--droit--defis--conflits--et--complementarites

[Le Métayer, Monteleone - 2009] Le Métayer, D., Monteleone, S.: Automated consent through privacy agents: legal requirements and technical architecture. The Computer Law and Security Review 25(2) (2009)

[Le Métayer et. al. - 2010a] Le Métayer, D., Maarek, M., Mazza, E., Potet, M.-L., Frenot, S., Viet Triem Tong, V., Crépeau, N., Hardouin, R.: Liability in software engineering: overview of the LISE approach and application on a case study. In: International Conference on Software Engineering, ICSE 2010, pp. 135–144. ACM/IEEE (2010)

[Le Métayer et. al. - 2010b] Le Métayer, D., Mazza, E., Potet, M.-L.: Designing log architectures for legal evidence. In: 8th International Conference on Software Engineering and Formal Methods, SEFM 2010, pp. 156–165. IEEE, Los Alamitos (2010)

[Le Métayer - 2010c] Le Métayer, D.: (ed.) Les technologies au service des droits, opportunités, défis, limites. Bruylant, Cahiers du CRID 32 (2010)

[Le Métayer - 2010d] Le Métayer, D.: Privacy by design: a matter of choice. Data Protection in a Profiled World, pp. 323–334. Springer, Heidelberg (2010)

[Le Métayer et. al. - 2011] Le Métayer, D., Maarek, M., Mazza, E., Potet, M.-L., Frenot, S., Viet Triem Tong, V., Crépeau, N., Hardouin, R.: Liability issues in software engineering. The use of formal methods to reduce legal uncertainties. Communications of the ACM (2011)

[Mazza et. al. - 2010] Mazza, E., Potet, M.-L., Le Métayer, D.: A formal framework for specifying and analyzing logs as electronic evidence. In: Davies, J. (ed.) SBMF 2010. LNCS, vol. 6527, pp. 194–209. Springer, Heidelberg (2011)

[Lessig - 2001] Lessig, L.: The future of ideas: the fate of the commons in a connected world. Random House (2001)

[Lessig - 2007] Lessig, L.: Code and other laws of cyberspace, Version 2.0. Basic Books, New York (2007)

[OECD - 1980] OECD guidelines on the protection of privacy and transborder flows of personal data. Organization for Economic Co-operation and Development (1980)

[Pace, Schneider - 2009] Pace, G.J., Schneider, G.: Challenges in the specification of full contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 292–306. Springer, Heidelberg (2009)

[Poullet - 2006] Poullet, Y.: The Directive 95/46/EC: ten years after. Computer Law and Security Report 22, 206–217 (2006)

[Prakken - 1996] Prakken, H., Sergot, M.J.: Contrary-to-duty obligations. Studia Logica 57, 91–115 (1996)

[Prisacariu, Schneider - 2007] Prisacariu, C., Schneider, G.: A formal language for electronic contracts. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 174–189. Springer, Heidelberg (2007)

[Reidenberg - 1998] Reidenberg, J.: Lex informatica: the formulation of information policy rules through technology. Texas Law Review 76, 3 (1998)

[Rouvroy - 2008] Rouvroy, A.: Privacy, data protection and the unprecedented challenges of ambient intelligence. Studies in Ethics, Law and Technology. Berkley Electronic Press (2008)

[Ryan - 2003] Ryan, D.J.: Two views on security and software liability. Let the legal system decide. IEEE Security and Privacy (2003)

[Steer et. al. - 2011] Steer, S., Craipeau, N., Le Métayer, D., Maarek, M., Potet, M.-L., Viet Triem Tong, V.: Définition des responsabilités pour les dysfonctionnements de logiciels : cadre contractuel et outils de mise en oeuvre. Actes du colloque Droit, sciences et techniques: quelles responsabilités, LITEC, collection Colloques et Débats (2011)

[Sweeney - 2002] Sweeney, L.: k-anonymity: a model for protecting privacy. Int. J. Uncertain. Fuzziness Knowl.-Based Syst. 10(5), 557–570 (2002)

[Thion - 2011] Thion, R., Le Métayer, D.: FLAVOR: a formal language for a posteriori verification of legal rules. In: IEEE International Symposium on Policies for Distributed Systems and Networks. IEEE, Los Alamitos (2011)

Developing Model-Checking Mechanisms for ASSL: An Experience Report

Emil Vassev and Mike Hinchey

Lero—The Irish Software Engineering Research Centre
University of Limerick, Limerick, Ireland
{Emil.Vassev,Mike.Hinchey}@lero.ie

Abstract. The Autonomic System Specification Language (ASSL) is a formal method dedicated to autonomic computing, and as such, assists developers with *formal specification*, *validation* and *code generation* of autonomic systems. Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. Moreover, one of the major objectives of the framework is to assure the correctness of autonomic systems via the inclusion of tools targeting model checking. In this paper, we report our experience in developing model-checking mechanisms for ASSL.

Keywords: model checking, formal methods, ASSL, autonomic computing.

1 Introduction

The Autonomic System Specification Language (ASSL) [1, 2] is an initiative for self-management of complex systems where we approach the problem of formal specification, validation, and code generation of autonomic systems (ASs) within a framework. Being dedicated to autonomic computing (AC) [3], ASSL helps AC researchers with problem formation, system design, system analysis and evaluation, and system implementation. The framework provides tools that allow ASSL specifications to be edited and validated. From any valid specification, ASSL can generate an operational Java application skeleton. The ASSL formal validation is addressed by multiple model checking mechanisms, some fully implemented and some still under development.

In general, model checking advocates for formal verification whereby software programs are automatically checked for specific flaws by considering correctness properties. In ASSL, some of those properties are defined as semantic definitions forming a theory that aids in the construction of correct AS specifications. For the purpose of developing flawless ASs, we are considering four distinct model checking mechanisms for ASSL: 1) a consistency checker; 2) a built-in model checker; 3) a mechanism for mapping ASSL specifications to a formal notation with provided tool support for model checking; and 4) a post-implementation model checker. Whereas the first three model-checking methods check ASSL specifications, the fourth one verifies the generated Java code. Note that despite careful specification and the existence of ASSL-level model checking, it is possible to generate ASs containing

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 19–34, 2010.

fatal errors (e.g., deadlocks). This is mainly due to the so-called state-explosion problem. Moreover, with the post-implementation model checker we may verify not only the newly-generated code but also all consecutively updated versions of the same. In this paper, we report our experience in developing the ASSL model checking mechanisms in the course of research projects at Lero–the Irish Software Engineering Research Centre.

The rest of this paper is organized as follows: In Section 2, we briefly present the ASSL formal specification model. In Section 3, we present our experience in the development of the four ASSL model checking mechanisms. Section 4 briefly outlines a case study where the built-in model checking approach is applied. Finally, Section 5 provides brief concluding remarks and a summary of future research goals.

2 ASSL

ASSL [1, 2] is based on a specification model exposed over hierarchically organized formalization tiers (see Table 1). This specification model provides both infrastructure elements and mechanisms needed by an AS (autonomic system).

Table 1. ASSL multi-tier specification model

AS	**AS Service-level Objectives**	
	AS Self-management Policies	
	AS Architecture	
	AS Actions	
	AS Events	
	AS Metrics	
ASIP	**AS Messages**	
	AS Channels	
	AS Functions	
AE	**AE Service-level Objectives**	
	AE Self-management Policies	
	AE Friends	
	AEIP	**AE Messages**
		AE Channels
		AE Functions
		AE Managed Elements
	AE Recovery Protocols	
	AE Behavior Models	
	AE Outcomes	
	AE Actions	
	AE Events	
	AE Metrics	

Each tier of the ASSL specification model is intended to describe different aspects of the AS in question, such as *service-level objectives, policies, interaction protocols, events, actions*, *autonomic elements*, etc. This allows us to specify an AS at different levels of abstraction (imposed by the ASSL tiers) where the AS in question is composed of special autonomic elements (AEs) interacting over interaction protocols (IPs). As shown in Table 1, the ASSL specification model decomposes an AS in two directions: 1) into levels of functional abstraction; and 2) into functionally related *sub-tiers*. The first decomposition presents the system at three different tiers [1, 2]:

1) *a general and global AS perspective* – we define the general system rules (providing autonomic behavior), architecture, and global actions, events, and metrics applied in these rules;
2) *an interaction protocol (IP) perspective* – we define the means of communication between AEs within an AS;
3) *a unit-level perspective* – we define interacting sets of individual computing elements (AEs) with their own autonomic behavior rules, actions, events, metrics, etc.

The second decomposition presents the major tiers, AS and ASIP, as composed of functionally related sub-tiers, where new AS properties emerge at each sub-tier. The AS Tier specifies an AS in terms of *service-level objectives* (AS SLOs), *self-management policies*, *architecture topology*, *actions*, *events*, and *metrics* (see Table 1). The AS SLOs are a high-level form of behavioral specification that helps developers establish system objectives such as performance. The *self-management policies* are driven by *events* and trigger the execution of *actions* driving an AS in critical situations. The metrics constitute a set of parameters and observables controllable by an AS. With the ASIP Tier, the ASSL framework helps developers specify an AS-level interaction protocol as a public communication interface expressed with special *communication channels*, *communication functions*, and *communication messages*. At the AE Tier, the ASSL formal model exposes specification constructs for the specification of the system's AEs. Note that AEs are considered to be analogous to software agents able to manage their own behavior and their relationships with other AEs. An AE may also specify a private *AE interaction protocol* (AEIP) shared with special AE considered as "friends" (AE Friends tier).

It is important to mention that the ASSL tiers are intended to specify different aspects of the AS in question, but it is not necessary to employ all of them in order to develop an AS. Conceptually, it is sufficient to specify self-management policies only, because those provide self-management behavior at the level of AS (the AS tier) and at the level of AE (AE tier). These policies are specified within the AS/AE Self-management Policies sub-tier (the ASSL construct is **AS[AE]SELF_MANAGEMENT**) with special ASSL constructs termed *fluents* and *mappings* [1, 2]. A fluent is a state where an AS enters with fluent-activating events and exits with fluent-terminating events. A mapping connects fluents with particular actions to be undertaken. Usually, an ASSL specification is built around one or more self-management policies, which make that specification AS-driven. Self-management policies are driven by events and actions determined deterministically. The following ASSL code presents a sample specification of a self-healing policy.

```
ASSELF_MANAGEMENT {
    SELF_HEALING {
        FLUENT inLosingSpacecraft {
            INITIATED_BY { EVENTS.spaceCraftLost }
            TERMINATED_BY { EVENTS.earthNotified } }
        MAPPING {
            CONDITIONS { inLosingSpacecraft  }
            DO_ACTIONS { ACTIONS.notifyEarth } }
    }
} // ASSELF_MANAGEMENT
```

As shown, fluents are expressed with *fluent-activating* and *fluent-terminating* events. In order to express mappings, conditions and actions are considered, where the former determine the latter in a deterministic manner.

Once a specification is complete, it can be validated with the ASSL built-in verification mechanisms (e.g., consistency checking) and a functional application skeleton can be generated automatically. The application skeletons generated with the ASSL framework are fully-operational multithreaded event-driven applications with embedded messaging.

3 Model Checking with ASSL

The ASSL framework helps developers edit and validate ASSL specifications and generate Java code, i.e., the ASSL toolset provides powerful tools needed to formally process an ASSL specification and automatically generate the corresponding implementation. The following subsections present the ASSL model checking mechanisms, used to validate the ASSL specifications.

3.1 Consistency Checking

The ASSL tiers can be classified as *declarative* (or imperative) and *operational* tiers [1, 2]. Whereas the former simply describe definitions in the AS under consideration, the latter not only describe definitions but also focus on the operational behavior of that AS. The ASSL framework evaluates an AS specification formally to construct a special *declarative specification tree* needed to perform both consistency checking (and eventually model checking) and code generation. The declarative specification tree is created by the framework when parsing an AS specification and copes with the hierarchical tier structure of that specification. Each specified tier/sub-tier is presented as a *tier instance*. Consistency checking (see Fig. 1) is a framework mechanism for verifying specifications by performing exhaustive traversing of the declarative specification tree. In general, the framework performs two kinds of consistency-checking: 1) *light* – checks for type consistency, ambiguous definitions, etc.; and 2) *heavy* – checks whether the specification model conforms to special *correctness properties*. The "*heavy*" consistency checking can be considered as a form of model checking, where the model is verified against predefined correctness properties.

The correctness properties are *ASSL semantic definitions* [1, 2] defined per tier. Although, they are expressed in First-Order Linear Temporal Logic (FOLTL)[1] [5],

[1] In general, FOLTL can be seen as a quantified version of linear temporal logic. FOLTL is obtained by taking propositional linear temporal logic and adding a first order language to it.

currently ASSL does not incorporate a FOLTL engine, and thus, the consistency checking mechanism implements the correctness properties as Java statements. Here, the FOLTL operators ∀(forall) and ∃(exists) work over sets of ASSL tier instances. It is important to mention that the consistency checking mechanism generates *consistency errors* and *consistency warnings*. Warnings are specific situations where the specification does not contradict the correctness properties, but rather introduces uncertainty as to how the code generator will handle it.

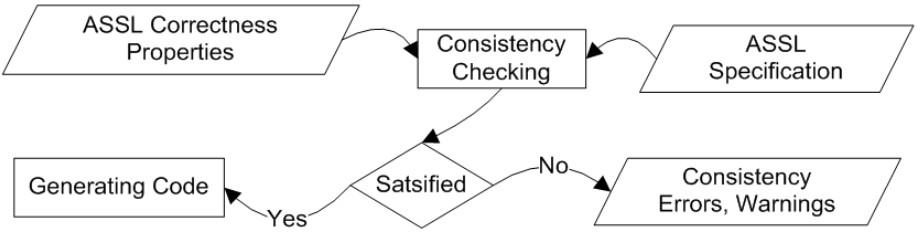


Fig. 1. Consistency Checking with ASSL

As mentioned above, a variety of predefined correctness properties are subject of consistency checking. One of those correctness properties is the so-called *autonomicity rule* [1, 2]. According to that rule, every autonomic system specified with ASSL must have specified at least one self-management policy. Fig. 2 shows an error reported by the ASSL consistency checker, because the processed ASSL specification violates the autonomicity rule (the entire ASSELF_MANAGEMENT sub-tier comprising the self-management policies is commented).

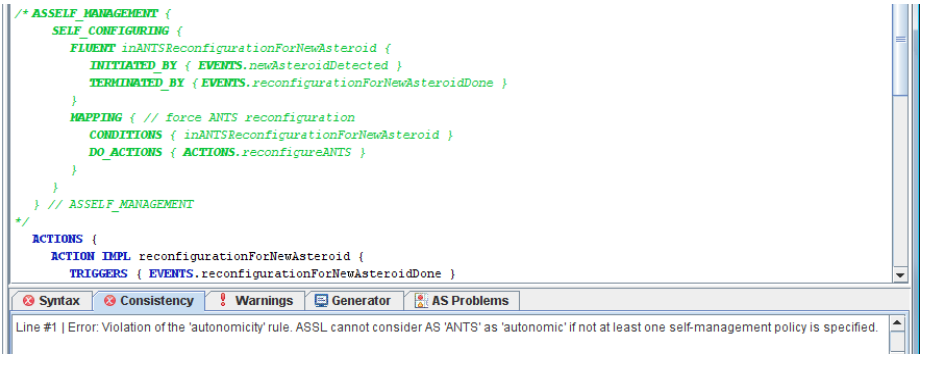


Fig. 2. Checking for "Autonomicity" with the ASSL Consistency Checker

3.2 Built-in Model Checker

In this approach, an ASSL specification is translated into a *state-transition graph*, over which model checking is performed to verify whether an ASSL specification satisfies *correctness properties*. Here, the model-checking problem is: given the AS A and its ASSL specification a, determine in the AS's state graph g (called ASG)

whether the behavior of A, expressed with the correctness properties p, meets the specification a [4]. An ASG formally stems from the concept of Kripke Structure [5]. The latter is basically a graph having the reachable states of an ASSL-specified system as nodes and the state transitions of the system as edges. In addition, to allow for formal verification, each system state must be labeled with properties (called atomic propositions AP) that hold in that state and each state transition must be associated with one or more state transition operations Op. The notion of state in ASSL is related to the ASSL specification constructs called *ASSL tier instances* [1, 2] (specified tiers and sub-tiers). The ASSL operational semantics [1, 2] considers a state-transition model where *tier instances* can be in different *tier states*, e.g., instances of the SLO (Service-Level Objectives) tier can be evaluated as *satisfied* or *not satisfied*. Here, an ASSL-developed AS transits from one state to another when a particular tier instance *evolves* from a tier state to another tier state. Here, transition operations Op cause tier instances to evolve.

3.2.1 Building the Autonomic System Graph

In order to build the ASSL model checker, we had to do some preliminary theoretical work to prepare the program structures holding an ASG. Here, we had to define:

1) the reference state model for ASSL-specified ASs, which appeared to be a product machine that consists of *high-level tier states* composed of multilevel *nested tier states*, and the global system state is a product of all nested states (we had to identify an initial state and all the possible tier states S);
2) a set of all atomic propositions AP, which denote the properties of individual states S, and present the S-AP relationship as tuples of the form (S_n, AP_1, …. AP_n);
3) all possible transition relations R as tuples of the form (S_1, Op, S_2).

Next, we had to implement structures holding the S-AP and R tuples. Note that those are recorded in two flat files (one per tuple type) and are loaded into the implemented program structures at the time of ASSL loading. This helps the model-checker tool cope with future extensions to ASSL. To implement the tuple structures, we used a distinct token class per tuple type (S-AP and R) and used vectors of tuple tokens. In addition, a generic algorithm is implemented to traverse those vectors and return a sub-vector of tuple tokens refined by *state*, by *operation*, or by *atomic proposition*. Thus, at runtime, the model-checking tool can obtain all the atomic propositions and related transition operations for a particular state. Here,

- tier states S are recorded with tier instance name and state name;
  Example: **tier** { *SLO* } **name** { *performance* } **state** { *unsatisfied* }
- transition operations Op are recorded with their ASSL predefined names [1];
- atomic propositions AP are recorded with "if" and "then" sections and optional "temporal" operators (a temporal logic operator).
  Example: **if** { *event prompted* } **then** { **tempOperator** { *eventually* } *fluent initiated* }

In the next step, we had to develop a mechanism constructing the ASG from an ASSL specification. Here, the ASG is constructed by the ASSL framework by using a special *declarative specification tree* created by the framework when parsing an AS

specification [1, 2]. The declarative specification tree contains the hierarchical tier structure of the actual specification. Thus, enriched with the tier states S, it can be used to derive the composite multilevel structure of the ASG by taking into consideration that all the tier instances run concurrently as state machines. Thus, the tier states S are derived from the declarative specification tree and enriched with the appropriate atomic propositions AP. The latter are retrieved per state.

In addition, the so-called *operational evaluation* [1, 2] performed on the ASSL specification is used to derive all the transition relations R (S_1, Op, S_2) needed to connect the states S and thus, to construct the ASG. Here, an ASG is composed of nodes that can be presented formally as a tuple (s, R, AP) where: s is the tier state; R is a set of transition relations connecting the state s to other states via system operations; AP is a set of atomic propositions held in s. Similar to the declarative specification tree, the generated ASG is hierarchical, i.e., composed of multilevel composite tier states. Note that the generated ASG is stored in a flat file, which helps us trace the graph. Fig. 3 depicts the transformation of the declarative specification tree into an ASG, where the latter is presented at the highest possible level of abstraction comprising a single composite state "AS Active", which is a product machine consisting of product states.

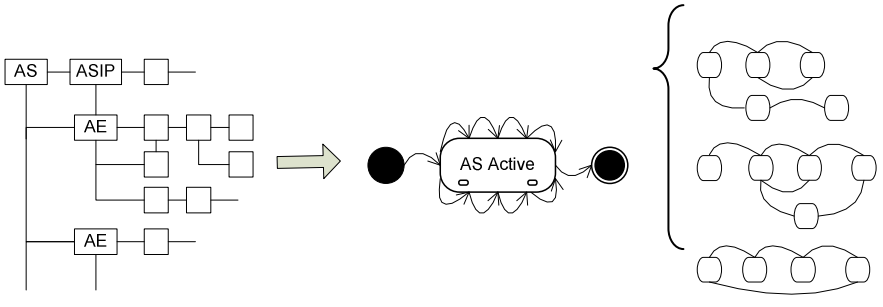


Fig. 3. Transformation of the Declarative Specification Tree into an ASG

3.2.2 Building the Model-Checking Engine

Next, we had to implement the model checking engine that should work over the following algorithm: *given that Φ is a correctness property expressed in a temporal logic formula, determine whether the "AS Active" tier state* (see Fig. 3) *satisfies Φ, which implies that all possible compositions of nested tier states satisfy Φ.*

Thus, the model-checking engine traverses all the possible paths in an ASG to check whether special correctness properties Φ (expressed in a temporal logic) are satisfied. In case such a property is not satisfied, the ASSL framework produces a counterexample. The latter is an execution path of the ASG for which the desired correctness property is not true.

At the time of writing, the model-checking engine is still under development. We are currently examining two possible solutions: 1) developing our own engine; or 2) integrating an already existing engine that can process the generated ASG file. Engines of current interest are SPIN [6] and GEAR [7]. In all approaches though, we need to consider the so-called *state-explosion problem*. In general, the size of an ASG is at least exponential in the number of ASSL tier instances running concurrently in

the system (recall that an ASG is a product machine). We are currently working on two possible solutions to that problem - *abstraction* and *prioritized tiers*. The first solution is to use composite tier states to abstract their nested tier states. Thus, given an original state graph G (derived from an ASSL specification) an abstraction is obtained by suppressing low-level tier states yielding a simpler and likely smaller state graph G_a. This reduces the total amount of states to be considered but is likely to introduce a sort of conservative view of the system where the abstraction ensures only that correctness of G_a implies correctness of G. The other possible solution is to prioritize ASSL tiers by giving their tier states a special probability weight pw. This can be used as a state-reduction factor to derive probability graphs G_{pw} with a specific level of probability weight, e.g., $pw > 0{,}5$. However, this approach is likely to introduce probability to the model-checking results, which correlates with the probability level of the graph G_{pw}.

3.3 External Model Checker

Another research direction of ours is towards mapping ASSL specifications to special *service logic graphs* supporting the so-called r*everse model checking* [8]. In this approach, to complement the original textual view of an ASSL specification, and in perspective to visualize and reify certain aspects of the operational semantics of ASSL, we map selected ASSL-specified behavioral elements to GEAR's behavioral models. These can be visualized as special Service Logic Graphs (SLGs) in the jABC framework [9] (of which GEAR is the model checking plugin) and analyzed, guiding the user through the processes and workflows of the specified autonomic system. Note that these models are directly amenable to model checking. SLGs themselves are composed of reusable building blocks that are called Service Independent Building Blocks, and may represent a single atomic service or a whole subgraph (i.e., another SLG). Thus SLGs can be hierarchical, which grants a high reusability not only of the building blocks, but also of the models themselves, within larger systems. SLGs formally stem from the concept ofa Kripke Transition System (KTS) [5]. Similar to any KTS, SLGs are graph structures with labeled branches and nodes that are enriched with atomic propositional properties, thus sufficing to adopt the established model checking technologies for SLGs.

From the point of view of model generation, AS and AE specifications are structurally identical with reference to events, self-management policies and actions, but differ in terms of scoping - while the AS specification has a global scope, the AE specification is only valid for the element in question (see Section 2). Due to the similarities, we focus on the description of the AE tier. The AS tier is captured similarly, by means of hierarchy (where single nodes of the AS-level KTS are expandable to AE-level models).

Fig. 4 shows a specification fragment of an ASSL specification of the Voyager spacecraft [10] (right) and the corresponding section of the behavioral model (left). In the textual specification (right), we have two events, one fluent with a mapping, and one action. Dashed arrows illustrate a trace of an event within the specs. Arrows indicate the correspondence between elements of the ASSL-specification and of the behavioral. The InTakingPicture cloud defines the current state of the system (an atomic proposition).

Event is the central language element in ASSL. It specifies fluents, actions, and policies globally in the AS tier and locally in the AE tier. Events can be activated by messages, other events, actions or metrics [1, 2]. In our behavioral model, events are mapped to homonymous Branches. In Fig. 4, the behavioral model starts with the event timeToTakePicture. It initiates the self-management policy (fluent) inTakingPicture.

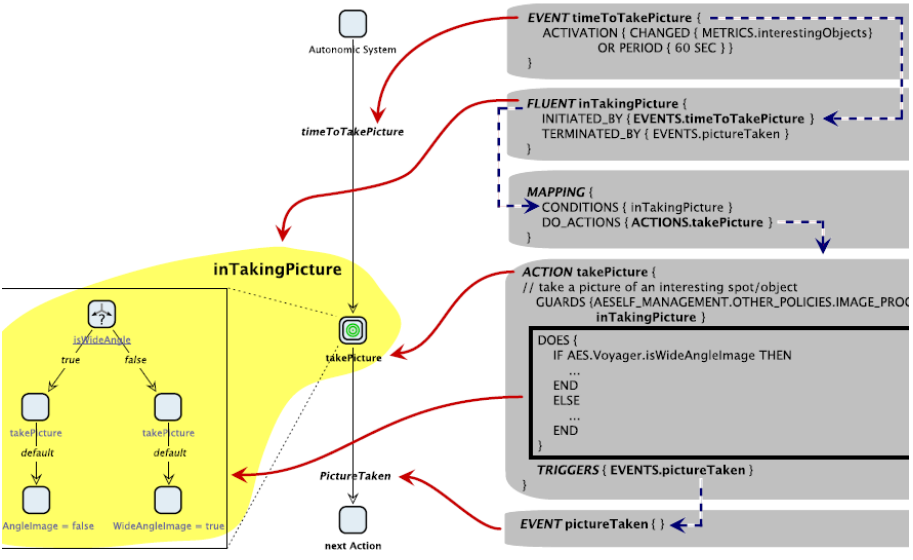


Fig. 4. Action, Event, Fluent, and Mapping in KTS behavioral model representation

An AE self-management policy defines the behavior of the AE by connecting specific system states (expressed with fluents) with the intended (re)action (expressed with mappings) (see Section 2). Fluents and mappings are central to the model extraction: the information contained in a self-management policy is used and useful both for model construction and for verification. Together, fluent and mappings define the control flow, i.e. create branches with the name of the initiating event. They define all possible incoming branches of an action. The specific condition that activates the fluent is stored in the context of the system's model. The context represents the current global state of the system, like a global Blackboard or shared memory-mechanism. For model checking purposes, the fluent is additionally associated as atomic proposition to the corresponding node(s) of the behavioral model. This enables global model checking. The fluent can be used as preconditions of actions. They hold on all states in the region between initiation and termination.

The fluent in our example is activated by the *timeToTakePicture* event and the overall status of the AS is changed to *intakingPicture*. This change activates an action: *takePicture* which is specified in the *mapping* section of the self-management policy. The self-management policy which connects the event to actions is additionally used to annotate the nodes in the behavioral model with atomic

propositions (APs). The name of the AP is equal to the name of the fluent. They can later be used for model checking.

Actions are routines performed by an AE or AS (global and local) [1,2]. In our behavioral model, they are the second essential element. The different elements of an action are used to describe the nodes and for verification purposes. Action parameters become parameters of a node; the *DOES* part [1, 2] (see the ASSL specification in Fig.4) represents the body of a node. It can be a single action (then the node is an atomic node), or a more complex structure where the latter is represented as an entire behavioral model. We then model them as a SLG hierarchy, as shown in Figure 4: the node *takePicture* has a corresponding submodel presented on the left.

The action's *guards*, *returns* and *outcomes* [1, 2] are used for verification. We offer two possibilities for verification:

- The GEAR's Localchecker mechanism uses the *guard* to verify if an action could be executed within the current system state (defined by the fluents and stored in a global context).
- We can use a model checker to verify relations of nodes and actions expressed as *temporal logic* [5] constraints. Internally, GEAR uses the modal μ - calculus [11] enriched with forward and backward modalities, so it is best equipped, for example, to express dataflow properties, or other behavioral constraints such as temporal logic formulas.

The specified action in Fig. 4 contains a guard, which must conform to the AP annotated at the node.

3.4 Post-Implementation Model Checker

In this approach, we rely on the Java PathFinder [12] tool to perform model checking on the ASSL-generated Java code.

3.4.1 Java PathFinder

Java PathFinder is a post-implementation model checker tool written in Java and targeted at Java programs [12, 13]. It can check Java programs for deadlocks, invariants and user-defined assertions in the code. Moreover, properties expressed in Linear Temporal Logic [14] can be checked. In general, it is claimed that Java PathFinder is capable of checking any Java program that does not rely on native methods. However, it is important to mention that the state-explosion problem limits the size of the applications that can be checked effectively up to 10,000 lines of code. Similar to any regular model checking tool, Java PathFinder performs exhaustive testing. The difference is that it works on the real Java code instead of on a state graph. Here, the basic technique is multiple execution of the program under consideration to check all the possible executions for paths that can lead to property violations, such as deadlocks or unhandled exceptions. If an error is found, Java PathFinder reports the execution path that leads to it. Note that every execution step is recorded, so we can trace the execution path that gets to property violation.

Fig. 5 depicts the operational model of Java PathFinder. As depicted, different components (tools) work by accompanying the execution of the compiled Java program (in Java bytecode), e.g., an ASSL-generated AS compiled to Java bytecode

with a regular Java Compiler. As shown in Fig. 5, special *configurable search strategies* are provided to solve the problem of state explosion. Because for large (more than 10,000 lines of code) applications the whole state space cannot be searched effectively, these search strategies are used to direct the search.

In addition, different *state-reduction techniques* can help to reduce the number of states that have to be stored:

- Special *heuristic choice generators* are provided to set possible choices where a certain state does not have to be complete. These generators have the form of Java PathFinder APIs that can be embedded in the tested applications.
- A special *library abstraction* per state reduces the overhead coming from tracking the run-time data changes taking place in the checked Java application. Note that all the heap, stack, and thread changes are stored by default. This can cause a big overhead if abstraction is not provided.

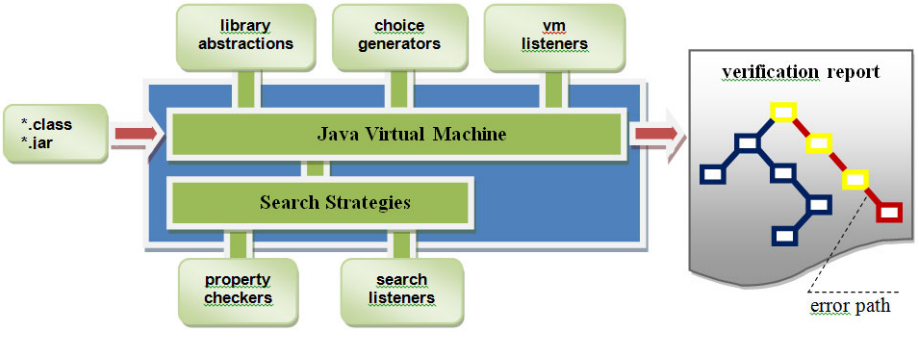


Fig. 5. Java PathFinder operational model (elaborated from [12])

3.4.2 Embedding Java PathFinder in ASSL

In general, Java PathFinder provides capabilities for non-deterministic testing via random input data generators [12] that can be embedded in the tested Java application. Special APIs are provided, which can significantly ease the creation of test drivers. Hence, the ASSL framework can automatically generate such test drivers based on the Java PathFinder API. ASSL could generate these special test drives as *non-deterministic choices* implemented in the generated code. Here, to simulate non-deterministic testing we rely on two Java PathFinder capabilities – *backtracking* and *state matching*.

With *backtracking*, we use the Java PathFinder tool to restore previous execution states, which helps to determine whether there are unexplored choices left. Therefore, if an end state is reached, backward steps can be performed to find execution paths that are still not executed, and thus, the program does not have to be re-executed from the very beginning.

With *state matching*, the Java PathFinder checks whether a specific execution path has already been explored any time when an ASSL-generated non-deterministic choice is reached. In such a case, model checking does not continue along the current execution path, but does backtracking to reach the *nearest non-explored path* that

starts from the nearest non-deterministic choice. For example, the following *run()* method could be generated by the ASSL framework for an autonomic element.

```
public class AE_WORKER {
   ...
   public void run () {
      boolean cond = Verify.getBoolean();
      if (cond)   {  ... }
      else  {  ... }
   }
   ...
}
```

Note that autonomic elements are generated by ASSL as Java Threads [1, 2]. Here, a non-deterministic PathFinder choice point will be generated (see `cond = Verify.getBoolean`) to test two different paths of execution of the autonomic element.Both *backtracking* and *state matching* techniques will be used to trace the two possible execution path – when `cond = true` and when `cond = false`.

4 Case Study: Checking Liveness Properties with ASSL

This section demonstrates how the ASSL built-in model-checking mechanism can perform formal verification to check *liveness* properties of an AS specified and generated with ASSL. Our example is the ASSL specification model for the NASA Voyager Mission [10]. In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs (autonomic elements) that follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. In this section, we use a sample from this specification to demonstrate how a liveness property such as "*a picture taken by the Voyager spacecraft will eventually result in sending a message to antennas on Earth*" can be checked with the ASSL model-checking mechanism. Note that the ASSL specification model for the NASA Voyager Mission is relatively large (over 1000 lines of specification code). Thus, we do not present the entire specification but a specification sample. For more details on that specification, please refer to [10].

```
POLICY IMAGE_PROCESSING {
   ....
   FLUENT inProcessingPicturePixels {
      INITIATED_BY { EVENTS.pictureTaken }
      TERMINATED_BY { EVENTS.pictureProcessed }
   }
   ....
   MAPPING {
      CONDITIONS { inProcessingPicturePixels }
      DO_ACTIONS { ACTIONS.processPicture }
   }
}

ACTION processPicture {
   ....
   DOES {
      ....
      call AEIP.FUNCTIONS.sendBeginSessionMsgs
      ....
   }
}
```

Fig. 6. The IMAGE_PROCESSING policy

Fig. 6 presents a partial ASSL specification of the **IMAGE_PROCESSING** self-management policy of the Voyager AE. Here the **pictureTaken** event will be prompted when a picture has been taken. This event initiates the **inProcessingPicturePixels** fluent. The same fluent is mapped to a **processPicture** action, which will be executed once the fluent gets initiated. As it is specified, the **processPicture** action prompts the execution of the **sendBeginSessionMsgs** communication function (see Fig. 6), which puts a special message **x** on a special communication channel [10] (message **x** is sent over that channel). Note that the specification of both the **pictureTaken** event and the **sendBeginSessionMsgs** function is not presented here. As we have already mentioned in Section 3.2, the ASSL model-checking mechanism builds the ASG (autonomic system graph) from the ASSL specification. Here both the *declarative specification tree* and the *ASSL operational semantics* [1, 2] are used to derive tier states S and transition relations R, and to associate those tier states via the ASSL transition operations Op. Next the labeling function $L(s)$ (integrated in the model-checking mechanism) labels each tier state s with appropriate atomic propositions AP.

Fig. 7 presents a partial ASSL ASG of the sub-tiers of the Voyager AE. These sub-tiers are derived from the declarative specification tree constructed for the Voyager AE. Note that this ASG is a result of our analytical approach and for reasons of clarity it is simplified, i.e., not all the possible tier states are presented here.

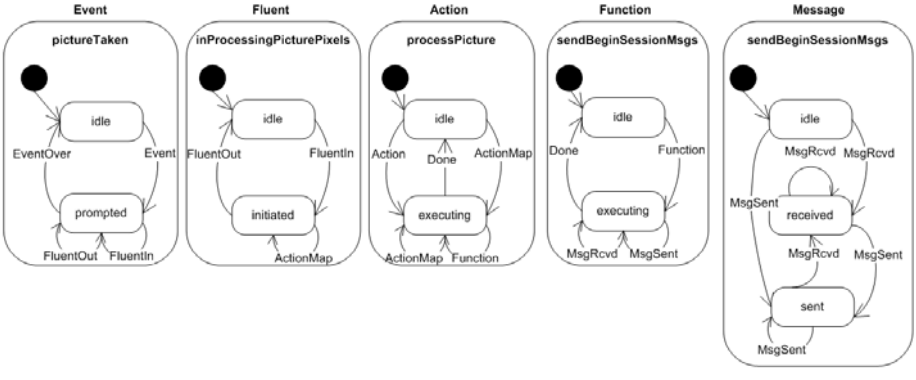


Fig. 7. State machines of the Voyager AE sub-tiers

As shown, each sub-tier instance forms a distinct *state machine* (basic machine) within the AE state machine and the AE state machine is a *Cartesian product* of the state machines of its sub-tiers. It is important to mention that by taking the Cartesian product of a set of basic sub-tier machines, we form a product machine consisting of product states. The latter are tuples of concurrent basic sub-tier states. Moreover, in the AE product machine, the ASSL state-transition operations Op are considered product transitions that move from one product state to another. Note that the states in the state machine of the whole AS product machine can be obtained by the Cartesian product of all the AE product machines. Thus, by considering the sub-tier state machines we construct the Voyager AE product machine (see Fig. 8). Note that this is again a simplified model where not all the possible product states are shown.

Fig. 8 presents the AE product states as large circles embedding the sub-tier states (depicted as smaller circles). Here we use the following aliases: *e* states for *Event state machine*; *f* states for *Fluent state machine*; *a* states for *Action state machine*; *y* states for *Communication Function state machine*; *x* states for *Message state machine*. Moreover, white circles present "idle" state and gray circles present the corresponding "active" state of the sub-tier state machine under consideration (such as: prompted for events, initiated for fluents, etc.; see Fig. 7).

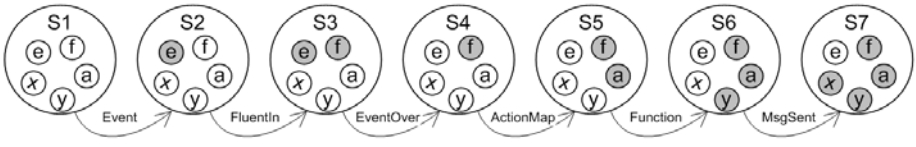


Fig. 8. Voyager AE product machine

Therefore, the formal presentation *(S; Op; R; S_0; AP; L)* (see Section 4.1) of the Voyager AE ASG is:

- *S = {S_1; S_2; S_3; S_4; S_5; S_6; S_7}*
- *Op = {Event; FluentIn; EventOver; ActionMap; Function; MsgSent}*
- *R = {(S_1;S_2;Event); (S_2;S_3;FluentIn); (S_3;S_4;EventOver); (S_4;S_5;ActionMap); (S_5;S_6;Function); (S_6;S_7;MsgSent)}*
- *S_0 = S_1 (initial state)*
- *AP = {* event *pictureTaken* occurs*,* event *pictureTaken* terminates*,* action *processPicture* starts*,* fluent *inProcessingPicturePixels* initiates*,* function *sendBeginSessionMsgs* starts*,* sends message *x }*
- *L(S):*
  - *L(S_1) = {* event *pictureTaken* occurs *};*
  - *L(S_2) = {* fluent *inProcessingPicturePixels* initiates *};*
  - *L(S_3) = {* event *pictureTaken* terminates *};*
  - *L(S_4) = {* action *processPicture* starts *};*
  - *L(S_5) = {* function *sendBeginSessionMsgs* starts *};*
  - *L(S_6) = {* sends message *x };*

Moreover, we consider the following correctness properties applicable to our case:

- *If an event occurs eventually a fluent initiates.*
- *If an event occurs next eventually it terminates.*
- *If a fluent initiates next actions start.*
- *If an action starts eventually a function starts.*
- *If a function starts eventually it sends a message.*

The ASSL model-checking mechanism uses the correctness property formulae to check if these are held over product states considering the atomic propositions *AP* true for every state. Thus, the ASSL framework is able to trace the state path shown in Fig. 6 and to validate the *liveness property* stated above. Note that in this example, we intentionally presented a limited set of atomic propositions *AP* and correctness

properties. The former are derivable, that is, deduced from the operational evaluation of the ASSL specification. Moreover, the Voyager AE product machine presents only product states relevant to our case study.

5 Conclusion and Future Work

We have presented our experience to-date in developing model-checking software verification mechanisms for the ASSL framework. Currently ASSL supports a family of software-verification framework tools (implemented or still under implementation) including a *consistency checker*, a *built-in model checker*, an *ASSL-to-SLG specification mapper* to support external model checking with the GEAR model checker and an *integration of the Java PathFinder* model checker to support post-implementation model checking. Currently, the ASSL consistency checker is the only fully implemented tool. It automatically checks ASSL specifications for consistency errors and some design flaws. The latter are verified against special consistency rules implemented as semantic definitions.

The other model mechanisms for ASSL require different implementation approaches. For example, to implement the built-in model checker, we developed program structures and algorithms that help an ASSL specification be transformed into a state-transition graph composed of special tier states with associated atomic propositions and transition relations connecting those states. We are currently developing a model-checking engine that works on the state transition graph. In addition, possible solutions to the so-called state-explosion problem are considered.

Our plans for future work are mainly concerned with further development of the model checker and test-case generator tools for ASSL. Moreover, in addition to the model-checking mechanisms, we are currently working on a special *test-case generator*, which aims at automatic generation of test suites for self-management policies. A test case is generated with a policy-execution path and test attributes that come in the form of inputs and special replacement ASSL constructs ensuring the execution of a tested policy. The test attributes are determined by change-impact analysis of the effect of a change in particular events or particular actions employed by an execution path. It is our understanding that such a testing mechanism will have a great impact on the development of prototype models for current and future space-exploration missions. Properly tested prototypes, eventually, will lead to the construction of more reliable spacecraft systems. Note that traditional methods, such as analyzing each requirement and developing test cases to verify the correctness of ASSL-implemented ASs, are not effective, because they require complete understanding of the overall complex system's self-management behavior.

Acknowledgment. This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre.

References

1. Vassev, E.: Towards a Framework for Specification and Code Generation of Autonomic Systems. PhD Thesis, Computer Science and Software Engineering Department, Concordia University, Quebec, Canada (2008)

2. Vassev, E.: ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems. LAP Lambert Academic Publishing (2009)
3. Murch, R.: Autonomic Computing: On Demand Series. IBM Press (2004)
4. Vassev, E., Hinchey, M., Quigley, A.: Model Checking for Autonomic Systems Specified with ASSL. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA, pp. 16–25 (2009)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
6. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2003)
7. Bakera, M., Renner, C.: GEAR - Game-based, Easy and Reverse Model Checking (2008), http://jabc.cs.tu-dortmund.de/modelchecking/
8. Bakera, M., Wagner, C., Margaria, T., Vassev, E., Hinchey, M., Steffen, B.: Component-oriented Behavior Extraction for Autonomic System Design. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA, pp. 66–75 (2009)
9. Nagel, R.: jABC, http://www.jabc.de
10. Vassev, E., Hinchey, M.: Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. In: Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), pp. 246–253. IEEE Computer Society, Los Alamitos (2009)
11. Kozen, D.: Results on the propositional μ-calculus. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 348–359. Springer, Heidelberg (1982)
12. Java PathFinder, http://javapathfinder.sourceforge.net/
13. Visser, W., Havelund, K., Brat, G., Park, S.-J.: Model Checking Programs. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000). IEEE Computer Society, Los Alamitos (2000)
14. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)

Models and Communication in the Policy Process

Matteo Pedercini

Millennium Institute
Washington D.C., U.S.A.
mp@millennium-institute.org

Policy-making is a complex process that involves a variety of actors. Several difficulties of various nature intervene in such process, making the identification and implementation of successful policies especially difficult. The usefulness of models in addressing technical obstacles related to the incorrect understanding of the issues and inferring of policy impacts have been broadly investigated [1,2]. Beyond facilitating technical aspects of the policy process, models can also facilitate communication among actors involved in such process.

From a high level perspective, the fundamental categories of actors involved in the policy process include: the broad public (holding a set of issues); politicians (elected to solve such issues); researchers (producing relevant knowledge and tools); and planners (applying such knowledge and tool to define policy options). When communication among actors does not work, the process should not be expected to identify the best solutions to the issues at stake. In such case one could expect, at best, some "Muddling Through" [3]: policy problems are never solved comprehensively and thus policy solutions advance toward better states by an ongoing process of iteration. At worst, the implemented policies can lead to undesired results [4]. Correct communication among actors is therefore essential for effective policy-making, and modeling constitute a fundamental connecting mechanism in the process.

One of the reasons of the difficulty in communication among actors has to do with the fact that different actors have different mental models, i.e. different informal theories, about present issues. In the words of Meadows, "Even in the modern age of science and industrialization social policy decisions are based on incompletely-communicated mental models" [5]. The use of formal modeling methods can help addressing this problem, potentially providing a solid, shared platform for discussion and policy analysis [1]. Specifically, models can be useful in facilitating transfer of key information about: (1) the identification of the key issues, by facilitating input from multiple stake-holders; (2) their causes, by illustrating underlying causal mechanisms; and (3) the possible policies to address them, by producing results in friendly formats.

In summary, models can help moving from a "Muddling Through" type of process, towards Saeed calls "Effective Policy Design" [6]. Nevertheless, to fully exploit their potential, modeling tools are to fulfill specific characteristics, most importantly a high degree of transparency [7, 8, 9, 10]. With the broad term "transparency" we consider here the following four characteristics: (1) based on clear assumptions; (2) broadly accessible; (3) user friendly; (4) produce clear and readily understandable outcomes.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 35–37, 2010.

Beyond model transparency, the characteristics of the modeling process are also fundamental for models to facilitate communication among actors in the policy process. Specifically, open modeling processes such as Group Model Building (GMB) can effectively increase understanding and confidence of model users and stakeholders in the results produced [11, 12, 13, 14]. Relevant aspects of such modeling approaches when applied to public policy-making include: (1) involving stake holders in definition of key issues; (2) involving experts in definition of key assumptions; (3) understanding how results will be used in the policy process; (4) and considering implications for policy-makers when formulating policy scenarios.

Based on the principles of transparency and GMB illustrated above, the Millennium Institute has, over the last two decades, developed a variety of modeling tools to support national development planning. Two cases illustrate especially well the application of such principles in different settings. A first such case is the Multy-Entity Gaming (MEG) tool developed in the early 2000's to facilitate regional policy discussion among the recently independent Balkan countries. MEG included a simulator enabling country representatives to introduce simultaneously their policy choices, and observe their impacts for each country and for the region as a whole. The relevance of the issues being addressed for the future of the region called for an effective way of communicating to the broad public results from the various scenarios. The characteristic of the modeling method used (System Dynamics) made it possible to trace results over time for key indicators, explain their relationship, and produce effective graphs and tables. Such results were then transformed into articles for a virtual newspaper (dated October 2015), illustrating what the future might look like depending on the different policy choices implemented. Such use of model results proven useful to facilitate communication among policy-makers and the broad public, laying the foundations for a successful policy-process.

A second case of application designed to facilitate communication among policy actors is the Bergen Learning Environment for National Development (BLEND), jointly developed with the University of Bergen. The first version of BLEND (as several versions have followed [15]) was designed to involve policy-makers (playing the role of different ministers) in simultaneous decision-making for a virtual country. The tool illustrates how decisions implemented by a given minister (say of education) affect performance of other ministers (say of health, infrastructure...) and thus the importance of communication among policy-makers. Allowing for different degrees of communication in different simulation sessions, policy-makers could appreciate how policy coordination and agreement on a shared development plan is essential for an effective policy process.

In summary, the usefulness of a modeling tool for actual policy analysis is strictly related to the possibility for policy makers to understand and use its results; and to the ability of involving the relevant actors of the policy process in the modeling. The transparency of the modeling tool and the opennes of the modeling process are therefore essential aspects for successful model-based policy analysis. The millennium Institute has, over the years, developed a vairety of applications following such principles, in order to support effective policy-making. Better software and better knowledge about the policy process are being continuously produced, faciliting further progress towards a type of modeling that is aware and sensitive to communication issues, and thus more useful to support policy-making. Despite

computer modeling is a highly technical discipline, it is fundamental that such soft aspects are properly considered: as Meadows puts it "The main problem is learning to communicate from each world to the other: we are not talking about a tool, we are talking about a subtle process of human communication" [5].

References

1. Sterman, J.D.: Business Dynamics: Systems Thinking and Modelling for a Complex World. McGraw-Hill Higher Education, New York (2000)
2. Tversky, A., Kahneman, D.: Judgment under uncertainty: Heuristics and biases. Science, New Series 185(4157), 1124–1131 (1982)
3. Lindblom, C.E.: The Science of "Muddling Through". Public Administration Review 19(2), 79–88 (1959)
4. Saeed, K.: A Re-evaluation of the Effort to Alleviate Poverty and Hunger. Socio Economic Planning Sciences 21(5), 291–304 (1987)
5. Meadows, D., Robinson, J.: The electronic oracle: computer models and social decisions. System Dynamics Review 18(2), 271–308 (2002)
6. Saeed, K.: Development planning and policy design, a System Dynamics approach. Ashgate, Londo (1994)
7. Senge, P., Sterman, J.D.: Systems thinking and organizational learning: Acting locally and thinking globally in the organization of the future. European Journal of Operational Research 59(1), 137–150 (1992)
8. Sterman, J.D.: Learning in and about complex systems. System Dynamics Review 10(2-3), 291–330 (1994)
9. Davidsen, P.I.: Educational Features of the System Dynamics Approach to Modelling and Learning. Journal of Structural Learning 12(4), 269–290 (1996)
10. Größler, A., Maier, F.H., et al.: Enhancing Learning Capabilities by Providing Transparency in Business Simulators. Simulation & Gaming 31(2), 257–278 (2000)
11. Vennix, J.A.M.: Building consensus in strategic decision making: system dynamics as a group support system. Group Decision and Negotiation 4(4), 335–355 (1995)
12. Andersen, D.F., Richardson, G.P., et al.: Group Model Building: Adding More Science to the Craft. System Dynamics Review 13(2), 187–201 (1997)
13. Ford, D.N., Sterman, J.D.: Expert Knowledge Elicitation to Improve Formal and Mental Models. System Dynamics Review 14(4), 309–340 (1998)
14. Fiddaman, D.: Dynamics of Climate Policy. Syst. Dyn. Rev. 23, 21–34 (2007)
15. Kopainsky, B., Pedercini, M., Davidsen, P.I., Alessi, S.M.: Blending planning and learning for national development. Simulation & Gaming 41(5), 641–642

Distributed Implementation of Systems with Multiparty Interactions and Priorities

Imene Ben-Hafaiedh[1], Susanne Graf[1], and Nejla Mazouz[2]

[1] VERIMAG. 2, avenue de Vignate
38610 Gieres, France
[2] Tunisia Polytechnic School
{benhfaie,graf,mazouz}@imag.fr

Abstract. Rich interaction models are a powerful mechanism allowing to synchronize several entities in order to achieve some common goal and to specify global properties in an abstract manner. In this paper we focus on two types of interaction models, namely *multiparty* interactions and *priorities* where priorities may be used to specify different scheduling policies. We propose a protocol for building distributed implementation of component-based models with multiparty interactions and priorities. We also present a set of experiments providing a performance analysis of the protocol.

Keywords: priorities, multiparty interaction, distributed systems.

1 Introduction

Providing a distributed implementation of component-based systems while preserving global properties is a very challenging task [5,18], as we cannot determine exactly the global state of distributed systems, but we can only approximate it [11]. Interaction models in component-based systems are a means for abstracting global properties of these systems. In this paper we focus on two types of interaction models, namely *multiparty interactions* and *priorities*.

Multiparty interactions provide a convenient means for describing the global behavior of a distributed system. Thus they can be later refined into efficient low-level protocols with respect to the platform in use. A multiparty interaction consists of a set of actions that need to be executed jointly by a number a components.

Priorities between interactions in component-based systems are widely used in system design as a way of defining different scheduling policies. They are expressive enough to enforce safety properties without inducing any deadlock in the system [13]. In fact, enforcing priorities means that when two interactions can be fired simultaneously, the one with higher priority must be executed. Thus, they restrict the behavior of the initial system which means that they preserve deadlock freedom if the initial system is deadlock-free.

The main challenge in enforcing priorities in a distributed setting is that components need to obtain a common and precise knowledge about the enabledness of interactions so the interaction with higher priority can be executed.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 38–57, 2011.

In [5], a partially distributed implementation has been proposed for component-based systems with priorities, but where a centralized engine manages interactions and enforce priorities. Existing distributed protocols implementing multiparty interactions [1,17] do not handle priorities. In [6], we have proposed a protocol for distributed implementation of systems with *binary* interactions and priorities. In a binary interaction exactly two components are involved. Which means that it is sufficient that each of the involved components gets information about the other to decide the execution of an interaction. Thus, the protocol presented in [6], is completely symmetric, which means that it does not distinguish between the two participants of an interaction. In the case of a multiparty interactions a completely symmetric protocol may, in the case where all partners of an n-ary interaction initiate the protocol almost simultaneously, lead to a huge number of messages. For this reason, most existing solutions for multi-party interactions (e.g. [17]) are asymmetric and predefine an "initiator" for each interaction; here we also choose this asymmetric approach.

In this paper, we propose a protocol providing a distributed implementation of component-based systems with multiparty interactions and priorities.

The paper is organized as follows. In Section 2, we present the basic semantics of a distributed system with multiparty interactions and priorities. Section 3 is dedicated to a thorough description of the protocol, and we prove its correctness in Section 4. We discuss in Section 5 some experimental results. In Section 6, we compare our proposal with other existing approaches.

2 Distributed System with Priorities

In this section, we present our notion of *distributed systems* defined by a set of *components*.

Definition 1 (component). *A* component K *is a* labeled transition system *(LTS)* $(Q, q^0, \mathcal{I}_K, \delta)$*:* Q *is a set of states with initial state* $q^0 \in Q$*,* $\mathcal{I}_K$ *is the label set.* $\delta \subseteq Q \times \mathcal{I} \times Q$ *is a transition relation.* $(q, a, q') \in \delta$ *is denoted* $q \xrightarrow{a} q'$ *and we denote by* $q \xrightarrow{a}$ *the fact that* $\exists\, q' \in Q$ *such that* $q \xrightarrow{a} q'$*.*

Definition 2 (distributed system). *A distributed system* $DS^{\mathcal{K}}$ *is defined by a set of components* $\mathcal{K} = \{K_i\}_{i=1}^n$*. It defines an LTS* $(\mathcal{S}, \mathcal{I}, \Delta)$ *where:*

- $\mathcal{I}$ *is a set of* multiparty interactions*. An interaction* a *may belong to more than one component and* $\mathcal{K}_a = \{K_i \in \mathcal{K} | a \in \mathcal{I}_{K_i}\}$*.*
- $\mathcal{S}$ *is the set of* global states *where* $S_0 \in \mathcal{S}$ *is the initial state. If* $S \in \mathcal{S}$*, then* $S = (q_1, \ldots, q_n)$ *where* $q_i \in Q_i$*. A* local state $S_{K_i} = q_i$ *is the restriction of the global state* S *to the states of* K_i*.*
- $\Delta \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ *is a transition relation.*

If a transition $\tau \in \Delta$, $\tau = (S, a, S')$ is fired, which means that the interaction a is executed, then the new global state S' is reached and we denote this by $S \xrightarrow{a} S'$. Δ is the least set of transitions satisfying the following rule:

$$\frac{a \in \mathcal{I} \qquad \forall i \text{ s.t. } K_i \in \mathcal{K}_a,\ q_i \xrightarrow{a} q_i' \qquad \forall i \text{ s.t. } K_i \notin \mathcal{K}_a,\ q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

Intuitively this rule means that a distributed system DS can execute an interaction in a global state S, if all components involved in this interaction can execute it in their local state corresponding to S. This means that firing a corresponds to synchronously firing the corresponding interactions a of the components in $\mathcal{K}_a$.

Priorities. Given a set of interactions $\mathcal{I}$, a priority between two interactions specifies which one is preferred over the other when both can be executed. Priorities are defined as partial orders $<\subseteq \mathcal{I} \times \mathcal{I}$ and we write $a < b$ means that a has less priority than b.

Definition 3 (distributed system with priorities). *A distributed system $DS^{\mathcal{K}} = (\mathcal{S}, \mathcal{I}, \Delta)$ with a priority order $<\subseteq \mathcal{I} \times \mathcal{I}$ is a distributed transition system $DS^{\mathcal{K}}_{<} = (\mathcal{S}, \mathcal{I}, \Delta_{<})$ where: $\Delta_{<} \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ is the largest transition relation satisfying the following rule:*

$$\frac{S \xrightarrow{a} S',\ \nexists b \in \mathcal{I}\ s.t.\ a < b \text{ and } S \xrightarrow{b}}{S \xrightarrow{a}_{<} S'}$$

Definition 4 (locally ready, globally ready, enabled interaction). *Let $DS^{\mathcal{K}}_{<}$ be a distributed system with priorities as above. Consider a global state $S = (q_1, \ldots, q_n) \in \mathcal{S}$, and an interaction $a \in \mathcal{I}$.*

- *a is* locally ready *in the* local state q_i *iff* $q_i \xrightarrow{a}_i$
- *a is* globally ready *in S iff* $\forall i\ s.t. K_i \in \mathcal{K}_a$, $q_i \xrightarrow{a}_i$
- *a is* enabled *in S iff a is globally ready in S and no interaction with higher priority is also globally ready in S.*

Definition 5 (global and local priorities). *Consider a system $DS^{\mathcal{K}}_{<} = (\mathcal{K}, \mathcal{S}, \mathcal{I}, \Delta_{<})$. A priority rule $a < b$ is called* local *if the interactions a and b have a common component, i.e., $\mathcal{K}_a \cap \mathcal{K}_b \neq \emptyset$. Otherwise, we call this priority rule* global.

We can now define the usual notions of *concurrency* and *conflict* of interactions, where in a distributed setting we want to allow the independent execution of concurrent interactions (so as to avoid global sequencing). We distinguish explicitly between the usual notion of conflict which we call structural conflict, and a conflict due to priorities.

Definition 6 (concurrent interactions, conflicting interactions). *Let a, b be interactions of $\mathcal{I}$ and $S \in \mathcal{S}$ a global state in which a and b are globally ready.*

- *a and b are called* concurrent *in S iff $K_a \cap K_b = \emptyset$. That is, when a is executed then b is still globally ready afterward, and vice versa, and if executed, both interleavings lead to the same global state.*
- *a and b are called* in structural conflict *in S iff they are not concurrent in S, that is a and b are alternatives disabling each other.*
- *a and b are in (purely)* prioritized conflict *in S iff a and b are concurrent in S but $a < b$ or $b < a$ holds.*

Note that in case of prioritized conflict, it is known which interaction cannot be executed, whereas in case of structural conflict, the situation is symmetric. We use the notations $Concurrent_S(a)$, $Conflict_S(a)$, $PrioConflict_S(a)$ *to denote the set of interactions that in state* q *are concurrent with* a*, respectively in structural or prioritized conflict with* a*.*

In distributed systems [9,8] the detection of some situations is important for designing correct protocols. Confusion is such a situation occurring when concurrency and conflict are mixed. More precisely, confusion arises in a state where two interactions a_1 and a_2 may fire concurrently, but firing one modifies the set of interactions in conflict with the other (see Definition 7). In presence of priorities, confusion situations may compromise the correctness of a distributed implementation of a specification.

Definition 7 (confusion). *Let* a *and* b *be interactions, and* S *a global state of* $DS_<$*. We suppose that* a *and* b *are concurrent — and thus globally ready — in* S*.*

- a *is in* structural confusion *with* b *iff* $\exists S' \in \mathcal{S}, S \xrightarrow{b} S'$ *implies* $Conflict_S(a) \neq Conflict_{S'}(a)$
- a *is in* prioritized confusion *with* b *iff* $\exists S' \in \mathcal{S}, S \xrightarrow{b} S'$ *implies* $PrioConflict_S(a) \neq PrioConflict_{S'}(a)$

In Section 3, we propose a distributed implementation of systems $DS_<$ in which concurrent interactions are executed independently, based on the notion of concurrency of Definition 6 and our implementation does not support systems $DS_<$ with prioritized confusion situations. To realize a distributed implementation of $DS^{\mathcal{K}}_<$ we use a *distributed controller* obtained by a set of local controllers exchanging messages with each other. By a *local controller* of a component K we understand a component that may allow or disallow interactions b of K. b is allowed by offering an interaction synchronizing with b and b is forbidden by not offering it.

Definition 8 (distributed controller). *A finite set of* local controllers *for a distributed system* $DS^{\mathcal{K}}$*, is a set of labeled transition systems* $\{LC_{K_i}\}_{i=1}^{n}$ *each associated to one component* K_i *of* DS*.* LC_{K_i} *restricts the behavior of* K_i*. In a state* $q_i \in Q_i$ *of* K_i*,* LC_{K_i} *decides which interaction to execute together with other local controllers, and synchronizes with* K_i *on this interaction. The set of local controllers* $\{LC_{K_i}\}_{i=1}^{n}$ *communicate amongst each others, by message passing, to decide which interaction to execute (see Figure 1).*

This definition of controller ensures a part of a safety property stating that only interactions specified by the local behavior of components can be executed as each local controller and its corresponding controlled component synchronize on the interaction a chosen to be executed. Moreover all components in $\mathcal{K}_a$ must also synchronize and execute a and this is what we will prove in Section 4. Thus interactions not specified by $DS_<$ cannot be executed.

The behavior of a given local controller is described by the protocol proposed in Section 3, where we describe how local controllers communicate using messages exchange to schedule an interaction for execution.

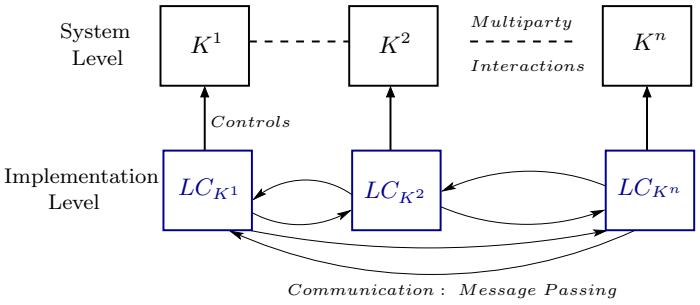


Fig. 1. Local Controllers and their Controlled Components

3 Protocol Description

In this section, we provide an overall picture of the distributed controller that is a *protocol* describing the global behavior of the set of local controllers. The behavior of the controller LC_K depends in each state q of K on the set of interactions locally ready in this state. LC_K exchanges messages with other local controllers of to decide when and which interaction to fire. Once an interaction a is chosen, LC_K synchronizes with K, and thus implicitly also with all other components involved in a to execute it.

In the following, we refer to K instead of LC_K whenever this is not misleading.

To decide when and which interaction to fire, local controllers communicate using *message passing*. We assume that the message passing mechanism, used by the controllers, ensures the following basic properties: (1) any message is received at the destination within a finite delay; (2) messages sent are received in the order in which they have been sent; (3) there is no duplication nor spontaneous creation of messages.

Table 1. Messages used by the Protocol

Message	Description
$POSSIBLE(a)$	If a is locally ready, the controller uses this message to inform the negotiator of a.
$NOTPOSSIBLE(a)$	Respond a negotiator that a is not locally ready.
$COMMIT(a)$	Message sent by the negotiator of a to its peers to lock them or sent back by a peer to respond the negotiator.
$READY(a)$	Sent by a negotiator to ask about the global readiness of a.
$NOTREADY(a)$	Sent by a negotiator to inform that a is not globally ready.
$START(a)$	Message sent by the negotiator to all the peers of a to order the execution of a.
$REFUSE(a)$	Sent by a controller to inform that it cannot commit to a.

The behavior of each local controller LC_K is given by a labeled transition system (see Figure 2) where the states of this LTS represent the different *phases* of the protocol. Transitions represent reactions to messages received by peer components which may consist in a local phase change and/or transmission of messages to some (other) peers. Thus, every transition of Figure 2 is specified by a *message* received, a *guard* and an *action* (see Table 3). Here *message* denotes the message that triggers a transition if the *guard* holds. If there is no message, the transition depends only on its *guard*. The role of each message is described in Table 1 where we use expressions of the form MESSAGE(interaction, sender) to denote the message triggering a given transition. The *action* of a transition denotes the list of statements to be executed if the transition occurs and may include message sending expressions of the form Send(message, recipient).

Some parts of the behavior of LC_K can be performed independently, thus we choose to describe it as a set of labeled transitions systems running in parallel and called *activities* (see Figure 3). These activities share the set of variables depicted in Table 2 and treat a set of disjoint messages. The transitions of Table 3 are performed by these activities where actions of the transitions may describe variable assignments, message sending or creating and killing new *activities*.

For each interaction of the to be controlled system $DS^{\mathcal{K}}_{<}$, we associate the role of *negotiator* to a local controller of one of the components involved in this interaction. Thus, a local controller may be the negotiator of a set of interactions in which its involved and each interaction has exactly one negotiator. The choice of negotiators of interactions and how it may affect the performance of the protocol is discussed in Section 5.1. This particular role of negotiator corresponds to the notion of *coordinator* defined in [17] and to the notion of *manager* presented in [3,2]. Note that comparing to the protocol for binary interactions, presented in [6], here negotiators are assigned to all interactions and not only to interactions involved in some priority rule as is the case in [6]. The reason is that in addition to the role of checking the enabledness of an interaction, the negotiator here checks also its global readiness. Thus, a controller presents a two phase behavior. A first phase is collecting knowledge about the global readiness of possible interactions and a second negotiating the enabledness of ready interactions. The phase *Active* is first entered by the transition 0, providing the set of interactions locally ready ($possibleSet(q_0)$) of the initial state of the controlled component. The controller LC_K looks for a next interaction to fire by proceeding as follows:

■ Once in phase *Active*, the activities $Main$ and $WaitingForCommit$ are created and run in parallel. $Main$ starts by checking its locally ready interactions ($possibleSet$) for interactions that are globally ready and for which it is the *negotiator*. For interactions in $possibleSet$ for which LC_K is not the negotiator, it only informs their corresponding negotiators about the local readiness of the interaction.

To check the global readiness of an interaction a, messages of the form $POSSIBLE(a)$ are exchanged (Transition 1 of Figure 2), and peers in which

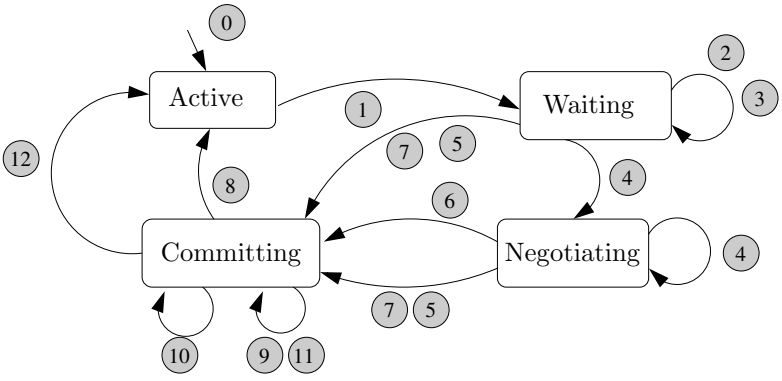


Fig. 2. The Phases of the Local Controller behavior (see Table 3 for the transitions)

a is currently not locally enabled respond with $NOTPOSSIBLE(a)$ after which the requesting controller abandons a.

If LC_K is the negotiator of a ($a \in toNegotiate$) and if it collects $POSSIBLE(a)$ from all the participants of a, then a is detected to be *globally ready*. If, in addition, a is involved in a priority rule, $Main$ creates a new activity $Negotiate(a)$ which checks whether a is enabled and the LC_K goes to phase $Negotiating$ through transition 4.

■ $Negotiate(a)$ activity checks the enabledness of an interaction a by sending a $READY(b)$ message to all negotiators of interactions b with higher priority than a ($b \in higherPrio(a)$), it checks whether their interactions are globally ready (and thus a cannot be executed now).

In turn the negotiators of b, as soon as they are not executing an interaction and have found out whether b is globally ready, respond positively or negatively as soon as they have the information available. In fact, it is sufficient that $NOTREADY(b)$ messages are sent as a is blocked anyway as long as it does not have a response concerning b.

■ If an interaction with maximal priority is globally ready, it is immediately known to be enabled and a $Committing$ phase is entered through transitions 5 or 6 by killing activities $Main$, $WaitingForCommit$ and $Negotiate$ and creating $TryToStart$ (see further down).

■ The $Main$ activity, which is running while LC_K is in the phases $Active$, $Negotiating$ and $Waiting$, handles local priorities locally. Whenever an interaction b is known to be globally ready, it kills all activities $Negotiate(a)$ if $a < b$.

■ Concurrently to $Main$, the activity $WaitingForCommit$ handles incoming $COMMIT$ messages. Whenever a $COMMIT(a)$ is received from the negotiator of a, which means that a is checked to be enabled by the negotiator of a. All other negotiation activities are terminated and a $TryToStart$ activity is created (Transition 7). The existence of the activity $WaitingForCommit$ means that no other actions is in its $Committing$ phase yet.

■ To avoid multiple commits for different interactions, *COMMIT* message is only sent by $TryToStart$ activity as it is created when all other activities terminate.
■ If LC_K is the negotiator of a, then $TryToStart(a)$ sends a $COMMIT(a)$ message to all participants of a and waits for a *COMMIT* response from all of them. Once, all participants send back a *COMMIT*, the negotiator orders the execution of a by sending a *START* message and it executes a together with the controlled component. Note that if $TryToStart$ fails committing to a because it receives a *REFUSE* message from at least one of the participants — in that case the peer has committed to a conflicting interaction — the controller starts again by checking the global readiness of its locally ready interactions (transition 8).
■ If LC_K is not the negotiator of a, then $TryToStart(a)$ sends a $COMMIT(a)$ message to the negotiator of a and waits for a *START* or a *REFUSE*. If it receives *START*, the controller executes a together with the controlled component which corresponds to the transition 12 of Table 3. If a *REFUSE* message is received, then activity $TryToStart(a)$ terminates and new $Main$ and $WaitingForCommit$ activities are created.
■ Finally, an activity $AnswerNegotiators$ (not represented in Figure 3) is always running in all states of the state diagram of Figure 2, if LC_K is the negotiator for at least one interaction a that dominates some other interaction. This activity receives messages of the form $READY(a)$. It returns $NOTREADY(a)$ if a is in the $notReadySet$, returns $READY(a)$ if a is in the $readySet$, and otherwise defers the answer until the status of a is known.

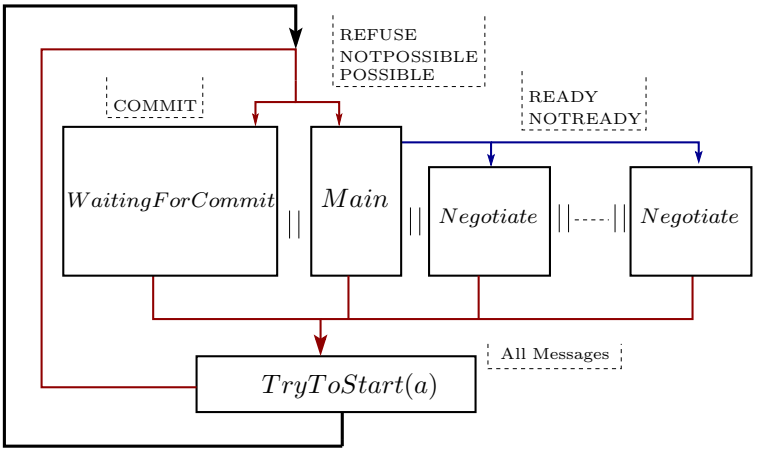


Fig. 3. Structure of the Protocol for a Local Controller

Avoiding Deadlocks Due to Decision Cycles

The protocol as described above may lead to a deadlock or a livelock in a situation where a set of components are all ready to execute at least 2 from a set of conflicting enabled interactions. Such a situation may occur in *Committing*

Table 2. Variables used by the Protocol

Variable	Description
$possibleSet(q)$	The set of interactions locally ready in q.
$readySet$	The set of interactions detected to be globally ready.
$notReadySet$	The set of interactions which are locally ready but detected to be not globally ready.
$toNegotiate$	The set of interactions for which the local controller is the negotiator.
$lessPrio(a)$	Interactions locally ready with less priority than a
$higherPrio(a)$	Interactions locally ready with higher priority than a.
$Peers(a)$	The set of participants in the interaction a.
$Neg(a)$	The negotiator of the interaction a.
$ReadyPeers(a)$	The set of participants in a for which a is locally ready.

phase of the protocol, where a negotiator has sent a $COMMIT$ message to all participants and waits for their response. Similarly one of these participants could be a negotiator of a different interaction and being in a *Committing* phase. This may lead to a deadlock if the set of interactions for which the negotiators are committing forms what we call a *cycle*.

Definition 9. *A cycle is a set of interactions* $A = \{a_i\}_{i=1}^{n}$ *involving a set of components* $\{K_i\}_{i=1}^{n}$ *for which the following holds: For all* $i \in [1, n]$, a_i *is an interaction involving the two components* K_i *and* $K_{\{i+1 mod n\}}$ *and there exists at least one global state in which all these interactions are enabled. We denote the fact that* A *is a cycle by* $Cycle(A)$*. Note that in such a global state each* K_i *has at least two enabled interactions, in the corresponding local state, one interaction with* K_{i-1} *and the other with* K_{i+1}*.*

To avoid deadlocks due to such cycles, we use a solution that we have already proposed for the binary version of our protocol described in [6]. The idea is to detect statically the set of potential cycles of the system. Then, we define for each cycle statically one of the components involved as a *Cyclebreaker*. Whenever a potential deadlock may be reached (from the point of view of the Cyclebreaker), the *Cyclebreaker* will commit to one of the interactions and refuse the other. This approach avoids defining a total order over all interactions or components, as proposed in [1,17], which may lead to systematically avoiding certain interactions and which may compromise liveness. Our solution is more faithful to the initial description of the system as it does not exclude any interactions unless an actual cycle occurs. Thus to avoid deadlocks due to cycles, if a given controller sends a $COMMIT$ message and then it receives another $COMMIT$ message for a different interaction, then either there is no cycle involving these two interactions or there exists at least one. In the second case, if the received $COMMIT$ concerns the interaction committed by a *Cyclebreaker*, then the controller cannot send back a $REFUSE$ and thus if this interaction is not the one committed by the

Cyclebreaker, the controller will send back a *REFUSE* which breaks the cycle. To perform this solution locally, we define pairs of interactions representing the local view of LC_K about a given cycle.

Notation 1. *We denote by* $cyclesof(K)$*, the set of pairs of interactions of* K *involved in some cycle.* $(a, b) \in cyclesof(K)$ *implies that* a *and* b *are interactions of* K *and* $\exists\ A$ *such that* $Cycle(A) \wedge \{a, b\} \subseteq A$.
We denote by $Cyclebreaker(A, K)$ *the predicate which holds if the component* K *is the* Cyclebreaker *of a cycle* A.
We denote also by $notRefuse(K)$ *the set of pairs of interactions of the form* (a, b) *such that* $(a, b) \in notRefuse(K)$ *implies:*

1. $(a, b) \in cyclesof(K)$
2. $\forall$ *cycles* A *such that* $\{a, b\} \subseteq A$, $Cyclebreaker(A, K_a)$ *holds, where* K_a *is a participant in the interaction* a*. This means that whenever* K *sends* $COMMIT(b)$ *message, and then it receives* $COMMIT(a)$*, it will not send back* $REFUSE(a)$.

Note that the order of interactions of a pair in $notRefuse(K)$ is relevant as the first interaction is the one that cannot be refused by K. Note that a pair of interactions $(a, b) \notin notRefuse(K)$ means that either there is no cycles involving these two interactions ($(a, b) \notin cyclesof(K)$) or that there exist such cycles ($(a, b) \in cyclesof(K)$) but if K commit for b and receives a *COMMIT* message for a then it can send back a $REFUSE(a)$ to its peer K_a because the latter is not the *Cyclebreaker* of these cycles. Theorem 3 proves that this way to deal with cycles allows indeed to avoid deadlocks.

4 Correctness

We now prove that the proposed protocol satisfies the following properties [2]:

- Correctness: Only interactions allowed by $DS^K_<$ can be executed:
  1. only locally ready interactions can be executed.
  2. if a component executes an interaction, the remaining components participating in that interaction will execute it (Synchronization).
  3. interactions in conflict (structural or prioritized conflict) cannot be committed simultaneously (Safety).
- Progress: when an interaction is enabled it will eventually be executed or one of its participants executes a conflicting interaction.

The first item of the correctness property is guaranteed by Definition 8, as each local controller and its corresponding controlled component synchronize on the interaction chosen to be executed. Thus only interactions which are locally ready for components can be executed.

Theorem 1 (Synchronization). *Our protocol guarantees that if a participant executes interaction* b*, then all of the components participating in* b *will execute it.*

Table 3. Transitions of the Local Controller State Diagram

Tr	*message*	*guard*	*action*
0		possibleSet(q_0)$\neq \emptyset$, q_0: initial state of K	Create($Main$), Create($WaitingForCommit$)
1		possibleSet$\neq \emptyset$	$\forall a \in$(possibleSet$\cap$toNegotiate), send(POSSIBLE(a),Neg(a)) $\forall a \in$(possibleSet\toNegotiate), send(POSSIBLE(a), Peers(a))
2	POSSIBLE(a)	($a \in$possibleSet\toNegotiate)	send(POSSIBLE(a),Neg(a))
2	NOTPOSSIBLE(a)	($a \in$possibleSet$\cap$toNegotiate)	$notReadySet$:= $notReadySet \cup a$
3	POSSIBLE(a, K)	($a \in$possibleSet$\cap$toNegotiate) $\wedge$(Peers(a)$\neq$readyPeers(a)$\cup K$)	readyPeers(a):= readyPeers(a)$\cup K$
4	POSSIBLE(a, K)	($a \in$possibleSet$\cap$toNegotiate) $\wedge$(Peers(a)==readyPeers(a)$\cup K$) $\wedge$ $a \notin$ prioFree	create(Negotiate(a)), readySet:=readySet$\cup a$, ($\forall$ $b \in$lessPrio(a),kill(Negotiate(b))
5	POSSIBLE(a, K)	($a \in$possibleSet$\cap$toNegotiate) $\wedge$(Peers(a)==readyPeers(a)$\cup K$) $\wedge$ $a \in$ prioFree	Kill(WaitingForCommit), Kill(Main), Send(COMMIT(a),Peers(a))
6		$Negotiate(a)$==OK	Kill(WaitingForCommit) Kill(Main), Send(COMMIT(a),Peers(a))
7	COMMIT(a)	($a \in$possibleSet\toNegotiate)	Kill(WaitingForCommit) Kill(Main), Send(COMMIT(a),Neg(a)
8	REFUSE(a)	Committed(a)	Goto(Active),Reset(readySet), Keep(possibleSet), Create($Main$), Create($WaitingForCommit$).
9	COMMIT(b)	Committed(a),($a \neq b$) (a, b) $\notin cyclesof(K)$ or (a, b) $\in notRefuse(K)$	waitingSet:=waitingSet$\cup\{b\}$
10	COMMIT(b)	Committed(a),($a \neq b$)and((a, b) $\in$ $cyclesof(K)$ and (a, b) $\notin notRefuse(K)$)	Send(REFUSE(b), Peers(b))$\wedge$readySet:=readySet\{b}
11	COMMIT(a, K)	Peers(a)$\neq$ readyToCommit(a)$\cup$K, $a \in$ possibleSet$\cap$toNegotiate	readyToCommit(a):= readyToCommit(a)$\cup K$
12	COMMIT(a, K)	Peers(a)==readyToCommit(a)$\cup$K, $a \in$ possibleSet$\cap$toNegotiate	Send(START(a), Peers(a)) and $\forall$ $b \in$possibleSet, Send(REFUSE(b), Peers(b))$\wedge$ Execute(a)
12	START(a)	$a \in$ possibleSet\toNegotiate	$\forall b \in$possibleSet,$b \neq a$, Send(REFUSE(b),Peers(b)), Execute(a),Update($possibleSet(q)$), Create($Main$), Create($WaitingForCommit$).

Proof. What we have to prove is that if a participant K_i of the interaction b executes it, then the rest of participants will also do so. If a component K_i which is the negotiator of b, executes b then it must according to transition 12 of Table 3 have received $COMMIT$ messages from all the participants of b. Note that $COMMIT$ is a blocking message, which means that all participants will stay waiting for a response from the negotiator and once they receive the $START$ message they will also execute b. If a component K_j which is not the negotiator

of b, executes b then it must according to transition 12 of Table 3 have received a $START$ message from the negotiator K_b of b. This means that similarly, K_b has sent a $START$ message to all participants of b, and as previously detailed all these participants are in a blocking state waiting for the message of the negotiator.

Theorem 2 (Safety). *Let be S a global state, b_1 an interaction and denote A the set $Conflict_S(b_1) \cup PrioConflict_S(b_1)$ of interactions that are in conflict with b_1 in the global state S. Our protocol guarantees that if b_1 is fired in state S, no interaction in A is fired in S.*

Proof. Suppose for $b_2 \in A$ that:
First case: $b_2 \in Conflict_S(b_1)$, that is b_1 and b_2 share a common component K. First of all, only interactions, for which the corresponding negotiator has sent a $START$ message to all participants, are executed. If the common component participating in b_1 and b_2 is the negotiator of both interactions, then only one interaction can be executed as according to transition 12 of Figure 2 a negotiator can send $START$ only for one interaction at a time and the property is satisfied. If b_1 and b_2 have different negotiators. Suppose that both negotiators have sent a $START$ message to execute b_1 and b_2. This means that all participants involved in b_1 and b_2 have sent a $COMMIT$ message to their negotiators (according to transition 12 of Table 3). As K is a common component, then this means that K has sent two $COMMIT$ messages one for the negotiator of b_1 and one for those of b_2 which is impossible as only one $COMMIT$ message can be sent at a time. In fact, a $COMMIT$ message can only be sent by the $TryToStart$ activity which does not have any other concurrent activity (see Figure 3).
Second case: $b_2 \in PrioConflict_S(b_1)$, that is b_1 and b_2 are concurrent (and thus belong to different components) and either $b_1 < b_2$ or $b_2 < b_1$. Suppose that K_{b_1} is the negotiator for b_1 and K_{b_2} is the negotiator for b_2.

If $b_2 < b_1$, then b_2 should not be executed before the execution of b_1 — which has started — has been completed and K_{b_1} enters *Active* phase for the successor state of S. We have now to prove that from that moment on K_{b_2} cannot "believe that b_1 is *not ready*" which is the condition for committing to b_2.

Indeed, if K_{b_2} does not yet know about the readiness of b_1, before committing b_2, it will send a $READY(b_1)$ message to K_{b_1}, but as b_1 is already engaged for execution, K_{b_1} will not send any response before the execution of b_1 is terminated the next state reached, and the readiness of b_1 evaluated in the new state; and K_{b_2} remains blocked for b_2 during this time.

Now, we must prove that K_{b_2} cannot have old, depreciated knowledge that b_1 is not ready. This can only be the case, if at some point b_1 was not ready and K_{b_1} has sent $NOTREADY(b_1)$ to K_{b_2}, and then transitions concurrent to b_2 have been executed leading to the current state S in which b_1 is ready and executed, and K_{b_2} may use incorrect knowledge and execute b_2. This corresponds exactly to a situation of confusion, which we have excluded (see Section 2). If $b_1 < b_2$, the situation is almost symmetric.

Lemma 1. *If a negotiator K_1 of an interaction a_0 sends a $COMMIT$ message to a participant K_2, then K_1 will receive a $REFUSE(a_0)$ or a $COMMIT(a_0)$ message from K_2 within a finite delay.*

Proof. We assume that the actual execution of an interaction a_0 as well as all the basic functions used in our protocol terminate and every message reaches its recipient within a finite delay. If K_1 waits for a response, after sending a $COMMIT(a_0)$ message to K_2, this means that it exists a global state S of the system in which a_0 is enabled and that K_2 is in the phase $Committing(a_1)$ ($a_0 \neq a_1$) (see the diagram of Figure 2). Indeed K_2 cannot be in any of the rest of the phases $Waiting$, $Active$ or $Negotiating$ as the activity $WaitingForCommit$ running in these phases (see Figure 3) will catch this $COMMIT(a_0)$ message and will send back a $COMMIT(a_0)$ to K_1. Thus, K_2 does not respond because it is trying to commit to another interaction $a_1 \neq a$. Which means that K_2 is in the phase $Committing(a_1)$ and that in the same global state S, a_1 is enabled. According to the Table 3 one of the following cases holds:

1- $(a_0, a_1) \in cyclesof(K_2)$ and $(a_0, a_1) \notin notRefuse(K_1)$ (according to the guard of transition 10 of Table 3), in this case K_2 sends back a $REFUSE(a_0)$ to K_1 within a finite delay.

2- $(a_0, a_1) \notin cyclesof(K_2) \vee (a_0, a_1) \in notRefuse(K_2)$, in this case K_2 is also waiting for an answer from K_3 about a_1. Similarly, if K_3 does not answer with a $REFUSE(a_1)$, then it exists an interaction a_2 enabled in S such that $(a_1, a_2) \notin cyclesof(K_3) \vee (a_1, a_2) \in notRefuse(K_3)$. As there exists a finite number n of components in the system, this means that there exists some cycle of size k in S for which the following holds:

$$\begin{gathered}(a_0, a_1) \notin cyclesof(K_2) \vee (a_0, a_1) \in notRefuse(K_2)\\ (a_1, a_2) \notin cyclesof(K_3) \vee (a_1, a_2) \in notRefuse(K_3)\\ \cdots\\ (a_{k-2}, a_{k-1}) \notin cyclesof(K_k) \vee (a_{k-2}, a_{k-1}) \in notRefuse(K_k)\end{gathered}$$

This is a contradiction. Indeed, the first part of each property means that there is no cycle containing these interactions, which is not true as we have a circular sequence which means a cycle. The second part does not hold as we assume that each cycle has exactly one $Cyclebreaker$ which may try to commit to one of the interactions only.

Theorem 3 (Progress). *Let b be an* enabled *interaction. Our protocol guarantees that b will eventually be executed or a component participating in it executes another interaction.*

Proof. The enabledness of an interaction is first detected by the negotiator of this interaction. An interaction b is enabled for its negotiator K_b, when COMMIT(b) messages are sent from all the participants of b. When it is detected to be enabled the negotiator of b goes to phase $committing(b)$. b becomes disabled when its negotiator leaves this state either by executing b (through transition 12 according to Table 3) or because one of the participants of b executes another interaction

(through transition 8 according to Table 3). In other words what we have to prove is that the negotiator K_b of b cannot stay in this state eternally. When K_b is in phase *committing*(b), then it has send $COMMIT$ messages to all participants of b, and according to Lemma 1, K will receive eventually a $COMMIT$ messages from all participants or at least one $REFUSE$ from one of the participants and thus will leave the phase *committing*(b) through transition 12 or 8.

5 Experimental Results

In this section, we report the results of experiments undertaken using an implementation of our protocol. Our implementation uses JAVA 1.6 to ensure the different algorithm computations and Message Passing Interfaces (MPIs) [15,19] to ensure the communication layer between components. Two metrics have been used to evaluate the performance of the protocol, namely *response-time* and *message-count*. The metric *message-count* computes the average number of messages needed to schedule one interaction for execution. The *response-time* is measured from the instant at which an interaction becomes locally ready (as viewed by its negotiator) to the instant at which it is selected for execution by the protocol. This metric is defined as the sum of two other metrics: *sync-time* and *selection-time*: *sync-time* measures the (mean) time taken by the algorithm to ensure that a given interaction is globally *ready*, starting from the moment when it is locally ready in its negotiator. *selection-time* measures the (mean) time taken by the algorithm to select an interaction for execution once it has been found globally ready. Note that the enabledness of an interaction is checked during the *selection-time*. *sync-time* is independent of priorities between interactions.

5.1 Sensitivity to the Choice of Negotiators

We illustrate the sensitivity of our protocol to the choice of negotiators by means of the well-known Dining Philosophers problem [12]. As proposed in [17], using multiparty interactions, a simple solution to this problem can be provided as each philosopher could pickup both two forks at a time by means of a three-party interaction (see Figure 5). However, using only binary interactions, the solutions to this problem must rely on some distinction amongst the behaviors of the philosophers, which makes such solutions not scalable nor reusable [16]. This problem models any situation where any entity needs to access a set of resources in mutual exclusion.

Using this example, we study how the choice of negotiators in a system may affect the performance of the protocol. We have carried out a series of experiments for the system of dining philosophers depicted in Figure 5 in the case of 2, 3 and 4 philosophers.

For each case, we have measured the already described metrics (message-count, sync-time and selection-time) to execute one interaction and we have focused on two configurations depending on the choice of the negotiators for

interactions. In a first configuration, we have assigned as negotiator of the interaction the component *Philosopher* involved in this interaction. Then, in a second configuration we have assigned the *Fork* component as negotiator for interactions in which it is involved. Figure 4, shows that the *message-count* when the *Philosophers* are the negotiators is higher than the case when *Forks* components are chosen as negotiators.

This is expected as the component *Fork* is involved in more than one interaction and thus it has more knowledge about the state of the participants of these interactions. However, the component *Philosopher* is involved in only one interaction and so, when it has to schedule an interaction for execution, it needs to communicate with other components to get knowledge about their states and thus exchanges more messages. More precisely, in the case of the system with two Philosophers, when the component $Philosopher_i$ is the negotiator of the interaction $getForks_i$, then according to the description of the protocol provided in Section 3, each *Philosopher* tries to commit for its interaction by sending to both *Forks* a $COMMIT$ message making a total of 4 $COMMIT$ messages. When a component $Fork_1$, for example, is the negotiator of both interactions in the system namely, $getFork_1$ and $getFork_2$, then $Fork_1$ tries to commit to only one interaction which means that only 2 $COMMIT$ messages will be sent by the negotiator. Consequently, when assigning negotiators in a system, the designer has to take into account the number of interactions in which negotiators are involved. Thus, the more the interactions in which a negotiator is involved, the less it needs to exchange messages. Similarly, the response-time metric is also affected by the choice of negotiators as it can be observed in the right-hand side of Figure 4.

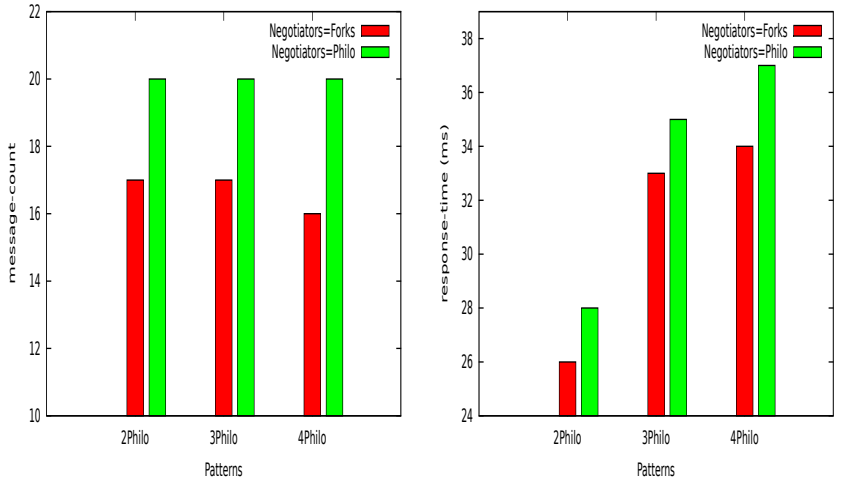


Fig. 4. Sensitivity to the choice of negotiators

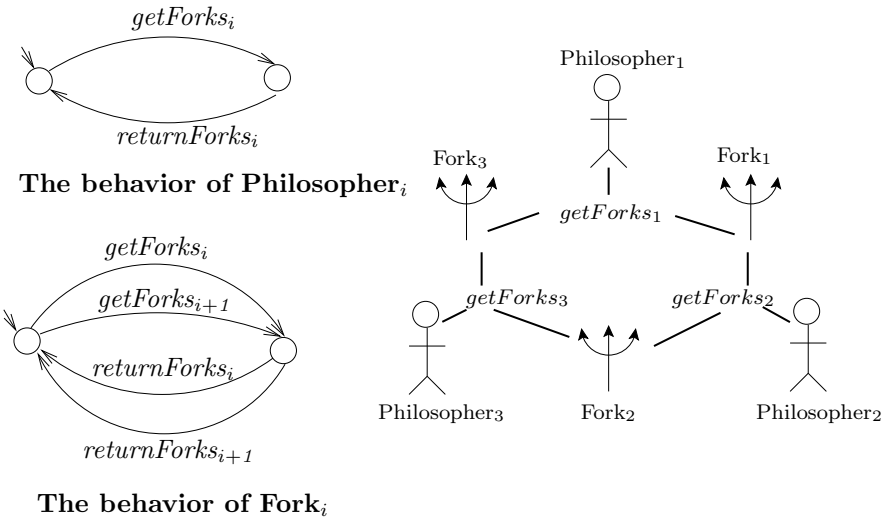


Fig. 5. The Dining Philosophers with multiparty interactions

5.2 Example with Priorities: Jukebox System

We use a Jukebox example to illustrate the use of priorities and to study how our algorithm performs, in particular, when global priorities are defined. The system is defined by a set of readers (R_1, ...,R_4) which need to access data located on Discs (D_1,D_2). Access to discs is handled by Jukebox components (J_1,J_2). Each Jukebox can load any disc to one of the readers it manages. Figure 6, represents the structure of the Jukebox system, where each *load* interaction corresponds to a $load_{J_i} D_i R_i$ interaction with J_i, D_i and R_i are respectively the connected Jukebox, Disc and Reader. The Jukebox J_1 manages the access of the readers R_1 and R_2, and J_2 the access of the readers R_3 and R_4. The behavior of each component is depicted in Figure 7. The interaction $load_{J_i} D_i R_i$ is a three-party interaction between the Jukebox J_i, The Disc D_i and the Reader R_i allowing, J_i to load D_i for the reader R_i. The interaction $unload_{j_i} D_i$ of unloading the disc is a binary interaction as it does not involve the reader. In Figure 7, the jukebox system is modeled without priorities. However, two types of priorities could be defined:

– Priorities to enforce termination: We give priority to *load* interactions over the *unload* ones. Formally, it defines the following priorities: $\{unload_{j_j} D_i < load_{J_j} D_i R_i\}_{j \in \{1,2\}}$. Note that this set of priorities defines *local priorities* as they include interactions of a common component namely the jukebox. Table 4, depicts the different results obtained when running the Jukebox system and measures the time taken and the *message-count* for the execution of two *load* interactions. Note that with priorities, the time taken to satisfy two readers is considerably lower than for the case without priorities. However, as introducing priorities needs more communication a main drawback is the *message-count*.

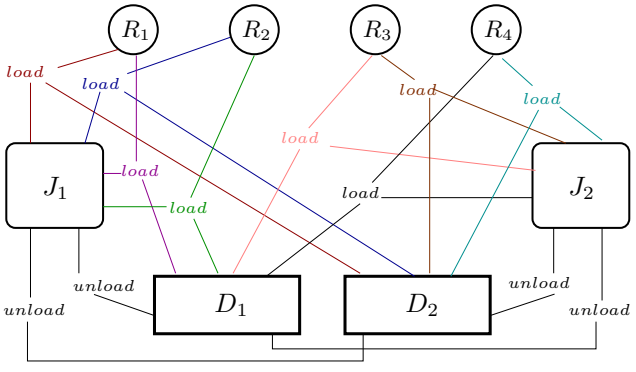


Fig. 6. The Jukebox System

– Priorities to manage resource access: We define priorities between Readers when accessing to a given Disc. We define for example the priority rule: $load_{J_1} D_1 R_1 < load_{J_2} D_1 R_3$ which means that whenever the Readers R_1 and R_3 want to load the Disc D_1, the priority is given to the Reader R_3. Such a priority rule involves the Disc as a common component which can consequently ensure this rule locally. However, in general, resources are represented by passive components which are not designed to manage or to make decisions. As managing access to resources is handled by the Jukebox component, we assign J_1 and J_2 as negotiators of the *load* interactions. Thus such a priority will be handled as a *global* priority rule, as components J_1 and J_2 have to communicate to decide which interaction to fire. Without this priority rule, for 20 executions of *load* interactions, which may involve any of the 4 Readers, we obtain an average of 915 messages exchanged. However, with $load_{J_1} D_1 R_1 < load_{J_2} D_1 R_3$, no interaction *load* for Reader 1 takes place and the average of the messages exchanged is about 1050, which is expected as global priorities are handled by the exchange of additional messages, namely $READY$ and $NOTREADY$, between the negotiators.

Table 4. Enforcing Termination using Priorities

	elapsed-time (ms)	message-count
Without priorities	580	65
With priorities	300	75

6 Related Work

This paper handles two important issues in the context of distributed control.

The first one concerns the enforcement of global properties, which are here priorities between interactions, in a distributed setting which is challenging to implement [5,11].

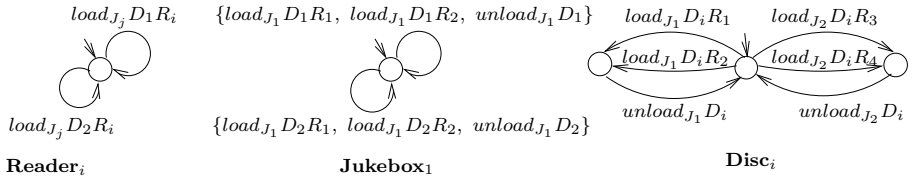


Fig. 7. Components of the Jukebox System

In [14,4,7], model-checking and *knowledge* are used to transform the system with priorities into a new system without priorities by restricting the possible choices in order to impose priorities. This means that they reduce concurrency of the initial system, whereas in our approach we guarantee *maximal progress*, which means that we allow all possible interactions of the initial description.

Similarly, to enforce priorities the approach proposed in [10] codes the priorities in the behavior of the initial system, so as to obtain a new model without priorities, then implements the so-obtained model in a distributed setting. Such approach adds to the system particular components called *managers* associated to each interaction which increase considerably the size of the studied system. Moreover, this makes their implementation less flexible to any change on the set of priorities as it means changing the system structure whenever the set of priorities is changed. Whereas in our approach the implementation of the protocol is still the same to any set of priorities.

Second, we propose a new protocol implementing *multiparty* interactions in a distributed setting. In [2,1,17], similar algorithms have been proposed but no *global* priorities are handled. In [3] *local* priorities have been defined to deal with deadlocks due to decision cycles as each controller explores its *possibleSet* in a decreasing order of priority which means that exactly one interaction, which has the highest priority, will be always executed if it is enabled. Similarly, to deal with such problem, the α-core algorithm proposed in [17] defines a total order between components which also means that some interactions can never be executed if they are in conflict with the interactions of the component with the highest priority, which limits the variability of the executions of the initial system. However, the solution we propose to deal with deadlocks due to decision cycles, uses some static *knowledge* about the structure of the system to define beforehand the set of potential cycles and then defines, only when needed, a priority given to the interaction chosen by the *Cyclebreaker*.

The α-core algorithm defines for each interaction a particular component called *Coordinator* managing the corresponding interaction and collecting information from all its participants to decide about its execution. In our approach the coordinator, i.e. the negotiator is one of the participant which allows to reduce the number of messages exchanged as the negotiator exploits already some local knowledge.

Similarly, in [1], *managers* are associated to interactions. However, a given manager is responsible for managing a subset of interactions and thus managing

conflicts between managers is achieved by means of a circulating token allowing the manager having it to execute its corresponding non-conflicting interactions. This solution based on a circulating token may lead to a situation in which a manager can never execute its interactions as it never gets the token at the right moment.

7 Conclusion

In this paper, we have focused on component-based systems with multiparty interactions and priorities. We have provided a protocol to implement such systems in a distributed setting. A variety of protocols for implementing multiparty interactions exist in the literature, but our approach is innovative as it handles in addition global priorities between interactions. An implementation of the proposed protocol is also provided with a set of experimental results allowing to analyze its performance. There are several research directions for future work. First, more experimentation is needed in particular to compare the performance of the protocol to existing approaches. Second, we are interested in combining our approach with *knowledge*-based methods, as it is proposed in [4], in order to optimize the performance of our protocol by taking into account a pre-calculated *knowledge* and thus reducing communication between local controllers.

References

1. Bagrodia, R.: A distributed algorithm to implement n-party rendezvous. In: Nori, K. (ed.) FSTTCS 1987. LNCS, vol. 287, pp. 138–152. Springer, Heidelberg (1987)
2. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Trans. Software Eng. 15(9), 1053–1065 (1989)
3. Bagrodia, R.: Synchronization of asynchronous processes in CSP. ACM Trans. Program. Lang. Syst. 11(4), 585–597 (1989)
4. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority Scheduling of Distributed Systems Based on Model Checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
5. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed Semantics and Implementation for Systems with Interaction and Priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008)
6. Ben-Hafaiedh, I., Graf, S., Quinton, S.: Building distributed controllers for systems with priorities. Journal of Logic and Algebraic Programming (2010)
7. Bensalem, S., Peled, D., Sifakis, J.: Knowledge based scheduling of distributed systems. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 26–41. Springer, Heidelberg (2010)
8. Bolton, C.: Adding Conflict and Confusion to CSP. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 205–220. Springer, Heidelberg (2005)
9. Bolton, C.M.: Capturing Conflict and Confusion in CSP. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 413–438. Springer, Heidelberg (2007)

10. Quilbeuf, J., Bonakdarpour, B., Bozga, M.: Automated distributed implementation of component-based models with priorities. Technical Report TR-2011-3, Verimag Research Report
11. Mani Chandy, K., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3(1), 63–75 (1985)
12. Dijkstra, E.W.: Hierarchical ordering of sequential processes, pp. 198–227. Springer-Verlag New York, Inc., New York (2002)
13. Gößler, G., Sifakis, J.: Priority Systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2003. LNCS, vol. 3188, pp. 314–329. Springer, Heidelberg (2004)
14. Graf, S., Peled, D., Quinton, S.: Achieving Distributed Control through Model Checking. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 396–409. Springer, Heidelberg (2010)
15. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, 2nd edn. MIT Press, Cambridge (1999)
16. Lynch, N.A., Merritt, M., Weihl, W.E., Fekete, A.: Atomic Transactions. Morgan Kaufmann, San Francisco (1993)
17. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience 16, 1173–1206 (2004)
18. Rudie, K., Murray Wonham, W.: Think globally, act locally: decentralized supervisory control. IEEE Transactions on Automatic Control 37(11), 1692–1708 (1992)
19. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The complete reference. MIT Press, Cambridge (1996)

Verification of PLC Properties Based on Formal Semantics in Coq

Jan Olaf Blech[1] and Sidi Ould Biha[2]

[1] fortiss GmbH, Munich
joblech@gmail.com
[2] INRIA and Tsinghua University, Beijing
Sidi.Ould_Biha@inria.fr

Abstract. Programmable Logic Controllers (PLC) are widely used in embedded systems for the industrial automation domain. We propose a formal semantics of two languages defined in the IEC 61131-3 standard for PLC programming. The first one is the *Instruction List* (IL) language, an assembly like language. The second one is the *Sequential Function Charts* (SFC) language, a graphical high-level language that allows to describe the main control-flow of the system. A PLC system description may comprise SFC and IL code. We formalized the semantics in the proof assistant Coq. Furthermore, we present an associated tool for automatically generating SFC representations from a graphical description – the text based IL code can be handled in Coq directly – and its usage for verification purposes. We demonstrate our approach to prove safety properties of a PLC in a real industrial demonstrator.

1 Introduction

Discovering and validating properties of Programmable Logic Controllers (PLC), is a prerequisite for the development of safety critical embedded systems. Tools and techniques for different kinds of systems and analysis scenarios have been developed. These comprise techniques aimed for distinct usage scenarios based on model checking and abstract interpretation.

In this work, we describe a general purpose way for the verification of PLC that are modeled using the Instruction List (IL) and Sequential Function Chart (SFC) languages of the IEC 61131–3 [15] standard. The standard is mainly used for modeling PLC functionality in the development of embedded systems for the industrial automation domain. We describe a tool set and method: For a given PLC description given in the graphical SFC language we automatically generate a Coq [9] description and some basic theorems and their proofs. In addition to the SFC language, text based IL programs are used in our PLC descriptions. We have formalized a syntactic representation of IL, thus, IL programs can be imported directly into our Coq environment. We present some standard techniques to reason about our PLC descriptions and verify properties. Furthermore, we present a case study of a PLC used inside a sorting machine.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 58–73, 2011.

The formalization of the IL and SFC semantics is done in the formal proof system Coq and its extension SSReflect [10]. Choosing Coq enables us to use its extraction mechanisms later and produce a certified compiler or interpreter for PLC based on our semantics. In this development, we also use some SSReflect libraries. In particular we use the libraries on booleans, natural numbers, lists and generic interface for types with decidable equality. The most important contributions of this paper comprise:

- Formal Coq semantics of the IL and SFC languages which are reusable for other projects.
- On overview on a tool to automatically generate SFC representations and some proofs.
- A case study on the verification of PLC properties using an IL and SFC description of a PLC.

Overview

This paper is organized as follows. We give an overview on PLC in Section 2. The IL and SFC language are presented in Section 3 and Section 4. A tool for generating Coq readable SFC representations and related proofs is described in Section 5. A case study is presented in Section 6. Section 7 discusses related work and a conclusion is featured in Section 8.

2 Programmable Logic Controller

A PLC is composed of a microprocessor, a memory, input and output devices where signals can be received from sensors or switches and sent to actuators. Figure 1 shows the architecture of a PLC system. A main characteristic of PLC is their execution mode. A PLC program is typically executed in a permanent loop. PLC program execution can be structured into *scan cycles* which are associated with a cycle time, the inputs are read, the program instructions are executed and the outputs are updated. The cycle time is often fixed or has an upper bound limit. Therefore the instructions which are scheduled to be executed in the cycle should terminate during the cycle time interval.

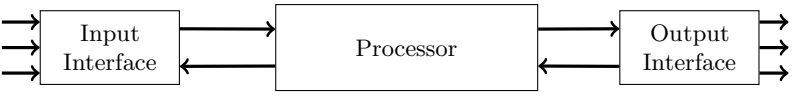


Fig. 1. PLC system

Since the introduction of PLC in the industry, each manufacturer has developed its own PLC programming languages. In 1993, the *International Electrotechnical Committee* (IEC) published the *IEC 1131 International Standard* for PLC. The third volume of this standard defines the programming languages for PLC. It defines 5 languages :

- *Ladder Diagrams* (LD) : graphical language that represent PLC programs as relay logic diagrams.
- *Functional Block Diagrams* (FBD) : graphical language that represent PLC programs as connection of different function blocks.
- *Instruction List* (IL) : an assembly like language.
- *Structured Text* (ST) : a textual (PASCAL like) programming language.
- *Sequential Function Charts* (SFC): a graphical language for describing top-level control-flow and associated data-flow in the PLC.

The last language differs from the other. It corresponds to a graphical method for structuring programs and allows to describe the system as a parallel state transition diagram. Each state is associated to some actions. An action is described using one of the other 4 PLC programming languages like LD or IL. SFC are well suited for writing concurrent control programs. In this paper we concentrate on the IL and SFC languages.

3 Instruction Lists

Instruction list (IL) is one of the five programming languages defined in the IEC 61131-3 standard. It is an assembly like language widely used for programming PLC systems. Our IL model is a significant subset of the language defined by the IEC 61131-3 standard. This subset covers assignments instructions and boolean and integer operations. It covers also comparison and branching instructions and *on-delay timers*. We choose to consider only booleans and integers as basic data types. In most of PLC systems, reals are available as basic data types, but rarely used. In practice, real number computation costs much time and may be delegated to an external device that can communicate with the PLC. This is motivated by the need to keep the program scan cycle within a relatively small time upper bound. The IL model we present in the following is an extension of the model defined in a previous work [14].

3.1 Syntax

An IL program comprises declarations of variables followed by a list of instructions. An IL program example is the following:

LABEL	OPERATOR	OPERAND
l1:	LD	x
	AND	y
	LD	z
	ORs	
	JMPC	l1

In the first line of the example above, the value of the variable `x` is loaded on a stack. After the execution of the second line, the stack contains the conjunction of `x` and `y`. In the third line, the value of `z` is put on the top of the stack. The

instruction `ORs` of the example above removes the two previous values loaded on the stack and replace them with $(\mathtt{x} \wedge \mathtt{y}) \vee \mathtt{z}$. The branching instruction `JMPC` is executed if the value at the top of the stack is equal to *true*.

An IL instruction starts with an operator that can be followed by one or more operands: variables or constants. In an instruction, the operator can be preceded by a label.
Instructions:

$i ::= \mathtt{LD}\ op \mid \mathtt{LDN}\ op$	load
$\mid \mathtt{ST}\ id \mid \mathtt{STN}\ id \mid \mathtt{SR}\ id \mid \mathtt{RS}\ id$	store, set and reset
$\mid \mathtt{JMP}\ lb \mid \mathtt{JMPC}\ lb \mid \mathtt{JMPC}\ lb$	jump
$\mid \mathtt{AND}\ op \mid \mathtt{OR}\ op \mid \mathtt{XOR}\ op$	boolean operations
$\mid \mathtt{ANDN}\ op \mid \mathtt{ORN}\ op \mid \mathtt{XORN}\ op$	
$\mid \mathtt{ANDs} \mid \mathtt{ORs} \mid \mathtt{XORs}$	
$\mid \mathtt{ADD}\ op \mid \mathtt{MUL}\ op \mid \mathtt{SUB}\ op$	integer operations
$\mid \mathtt{ADDs} \mid \mathtt{MULs} \mid \mathtt{SUBs}$	
$\mid \mathtt{GT}\ op \mid \mathtt{GE}\ op \mid \mathtt{EQ}\ op$	comparison
$\mid \mathtt{GTs} \mid \mathtt{GEs} \mid \mathtt{EQs}$	
$\mid \mathtt{TON}\ id\ ,\ n$	On delay timer
$\mid \mathtt{RET}$	end of program

Operands:

$op ::= id \mid cst$ variable identifier or constant

Constants:

$cst ::= n \in \mathbb{Z} \mid b \in \mathbb{B}$ integer or boolean literal

The data domains of IL constants is the union of integers $\mathbb{Z}$ and booleans $\mathbb{B}$. In practice integers used in PLC are bounded. For simplicity, we restrict ourselves to unbounded integers in the presentation of this work. Adjusting the integer size in Coq – and other higher-order theorem prover based – developments is not a difficult task and has been studied before (e.g., [12]).

We denote the set of IL instructions by *Instr*. For simplicity, we suppose that IL program labels are natural numbers. Since an IL program is a list of instructions, a label indicates the position of the corresponding instruction in the list. For a given program p and an index i, $p(i) \in Instr$ represents the instruction of p at the position i.

3.2 Semantics

We defined a small step operational semantics of IL programs. For the purpose of modeling PLC timers, we suppose having a global discrete time clock and that each program execution cycle has a fixed time duration denoted δ.

Stack: in IL an evaluation *stack* is used for the current result computation. It is also used to store intermediate results that will be pulled back when an instruction like `ADDs` or `ANDs` are executed. A *stack* $V := v_1, \ldots, v_m$ is a finite

$$\text{LD}\ \frac{p(i) = \textbf{LD}\ op}{(s,\sigma,i) \rightarrow (push\ op\ s, \sigma, i+1)} \qquad \frac{p(i) = \textbf{LDN}\ op}{(s,\sigma,i) \rightarrow (push\ \neg op\ s, \sigma, i+1)}\ \text{LDN}$$

$$\text{ST}\ \frac{p(i) = \textbf{ST}\ x \qquad \sigma' = \sigma[x \mapsto top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)} \qquad \frac{p(i) = \textbf{STN}\ x \qquad \sigma' = \sigma[x \mapsto \neg top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)}\ \text{STN}$$

$$\text{SR}\ \frac{p(i) = \textbf{SR}\ x \qquad \sigma' = \sigma[x \mapsto x \vee top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)} \qquad \frac{p(i) = \textbf{RS}\ x \qquad \sigma' = \sigma[x \mapsto x \wedge \neg top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)}\ \text{RS}$$

$$\text{JMPC}_\text{T}\ \frac{p(i) = \textbf{JMPC}\ l \qquad top\ s = \textbf{T}}{(s,\sigma,i) \rightarrow (s,\sigma,l)} \qquad \frac{p(i) = \textbf{JMPC}\ l \qquad top\ s = \textbf{F}}{(s,\sigma,i) \rightarrow (s,\sigma,i+1)}\ \text{JMPC}_\text{F}$$

$$\text{JMPCN}_\text{F}\ \frac{p(i) = \textbf{JMPCN}\ l \qquad top\ s = \textbf{F}}{(s,\sigma,i) \rightarrow (s,\sigma,l)} \qquad \frac{p(i) = \textbf{JMPCN}\ l \qquad top\ s = \textbf{T}}{(s,\sigma,i) \rightarrow (s,\sigma,i+1)}\ \text{JMPCN}_\text{T}$$

$$\text{ANDs}\ \frac{p(i) = \textbf{ANDs} \qquad (t,t') = top2\ s}{(s,\sigma,i) \rightarrow (push\ (t \wedge t')\ (pop2\ s), \sigma, i+1)} \qquad \frac{p(i) = \textbf{AND}\ op \qquad t = top\ s \wedge op}{(s,\sigma,i) \rightarrow (push\ t\ (pop\ s), \sigma, i+1)}\ \text{AND}$$

$$\text{ADDs}\ \frac{p(i) = \textbf{ADDs} \qquad (t,t') = top2\ s}{(s,\sigma,i) \rightarrow (push\ (t + t')\ (pop2\ s), \sigma, i+1)} \qquad \frac{p(i) = \textbf{ADD}\ op \qquad t = top\ s + op}{(s,\sigma,i) \rightarrow (push\ t\ (pop\ s), \sigma, i+1)}\ \text{ADD}$$

$$\text{GTs}\ \frac{p(i) = \textbf{GTs} \qquad (t,t') = top2\ s}{(s,\sigma,i) \rightarrow (push\ (t < t')\ (pop2\ s), \sigma, i+1)} \qquad \frac{p(i) = \textbf{GT}\ op \qquad t = top\ s < op}{(s,\sigma,i) \rightarrow (push\ t\ (pop\ s), \sigma, i+1)}\ \text{GT}$$

$$\text{TON-OFF}\ \frac{p(i) = \textbf{TON}\ Tx, Pt \qquad top\ s = \textbf{F} \qquad \sigma' = \sigma[Tx.Q \mapsto \textbf{F}, Tx.ET \mapsto 0]}{p \vdash (s,\sigma,i) \rightarrow (s,\sigma',i+1)}$$

$$\text{TON-ON}\ \frac{p(i) = \textbf{TON}\ Tx, Pt \qquad top\ s = \textbf{T} \qquad Tx.ET < Pt \qquad \sigma' = \sigma[Tx.Q \mapsto \textbf{F}, Tx.ET \mapsto Tx.ET + \delta]}{p \vdash (s,\sigma,i) \rightarrow (s,\sigma',i+1)}$$

$$\text{TON-END}\ \frac{p(i) = \textbf{TON}\ Tx, Pt \qquad top\ s = \textbf{T} \qquad Tx.ET >= Pt \qquad \sigma' = \sigma[Tx.Q \mapsto \textbf{T}, Tx.ET \mapsto Tx.ET + \delta]}{p \vdash (s,\sigma,i) \rightarrow (s,\sigma',i+1)}$$

Fig. 2. IL operational semantics

sequence of data values. In the following we use the standard stack operations *push* (add an element to the stack), *pop* and *pop2* (remove respectively the top and the two top elements of the stack), *top* and *top2* (return respectively the top and the two top elements of the stack).

States: functions from variable identifiers to data values. They represent the program variable states and are denoted σ of the type $\mathcal{S} = Var \rightarrow D$, where D is the union of the IL variables data domains: $\mathbb{Z} \cup \mathbb{B}$.

Configurations: elements of the set $\mathcal{E} = Stack \times \mathcal{S} \times \mathbb{N}$. A configuration (s, σ, i) corresponds to a stack s, a state σ and a position or program location i.

Transitions: relation on configurations $\subseteq \mathcal{E} \times \mathcal{E}$. Figure 2 gives some relevant inference rules of the IL configurations transitions relation. We denote the relation defined by the inference rules of Figure 2 by $\xrightarrow{x}$ where x is an IL instruction. The IL transition system is defined by an initial configuration $(s_0, \sigma_0, 0)$, where s_0 is the empty stack and σ_0 is the initial state that maps all the integer variables to 0 and boolean variables to $false$.

$$\textsc{Left}\ \frac{i < \mathit{length}\ p \qquad (s,\sigma,i) \xRightarrow{p} (s'',\sigma'',i'') \qquad (s'',\sigma'',i'') \xRightarrow{[p,x]} (s',\sigma',i')}{(s,\sigma,i) \xRightarrow{[p,x]} (s',\sigma',i')}$$

$$\textsc{Right}\ \frac{i = \mathit{length}\ p \qquad (s,\sigma,i) \xrightarrow{x} (s'',\sigma'',i'') \qquad (s'',\sigma'',i'') \xRightarrow{[p,x]} (s',\sigma',i')}{(s,\sigma,i) \xRightarrow{[p,x]} (s',\sigma',i')}$$

$$\textsc{Out}\ \frac{\mathit{length}\ p \leq i}{(s,\sigma,i) \xRightarrow{p} (s,\sigma,i)}$$

Fig. 3. IL natural semantics

The first four transition rules of Figure 2 correspond to the *load* and *store* instructions. In the first case the stack is updated while in the second the variable state is updated. The transitions corresponding to the *set/reset* instructions (rules SR and RS) update the variable state function with the corresponding values for the given operands and the top of the stack. The transition relation for the TON instruction is given by the rules TON-OFF, TON-ON and TON-END of Figure 2. The elapsed time variable ET of the TON timer is incremented by the global constant δ when the timer is activated (the value of top of the stack is *true*). The timer output Q is activated when the elapsed time variable ET is greater or equal to the timer delay parameter PT.

Natural semantics: sometimes in the reasoning about IL programs we need to interpret the execution of the entire program. This can be done using natural semantics or big-step semantics. On top of the small-step semantics presented above, we defined also a big-step semantics of IL programs. The inference rules of this semantics are given in Figure 3. They correspond to the definition of the transitive closure of the small-step semantics relation. As we mentioned before an IL program should terminate during the cycle scan time. This termination property is assured by the rule OUT. A final state is one where the location index is greater than the program instruction list length. By using the rule OUT we can prove that every execution, following the rules of our natural semantics, will reach a final (or stable) configuration.

3.3 Formalization

We formalized the IL semantics defined above in the formal proof system Coq. The Coq system provides a powerful mechanism to define recursive or finite types or sets: *inductive types*. It is especially useful when defining the syntax of a programming language. We define the IL syntax and operational semantics presented above, using the Coq inductive type mechanism. In our formalization, IL instructions are represented by the type `Instr` and an IL program or a list of IL instructions is an object of the type `code := seq Instr`[1].

We also formalized the IL big-step semantics defined in Figure 3 as a Coq inductive relation. The definition is given in the Figure 4. Since it is not always

[1] `seq` is the type of list in SSReflect.

```
Inductive il_exec : code -> ILConf -> ILConf -> Prop :=
| il_exec_consl : forall p x cf cf1 cf2, cf.2 < size p ->
  il_exec p cf cf2 -> il_exec (rcons p x) cf2 cf1 ->
  il_exec (rcons p x) cf cf1
| il_exec_consr : forall p x cf cf1 cf2, cf.2 = size p ->
  il_trans (rcons p x) cf cf2 -> il_exec (rcons p x) cf2 cf1 ->
  il_exec (rcons p x) cf cf1
| il_exec_out : forall p cf, size p <= cf.2 -> il_exec p cf cf.
```

Fig. 4. Coq definition of the IL program execution predicate

possible to know how many transitions are needed to execute an IL program, we define the program execution as a propositional relation rather than a computational function. However it is possible to define a function that returns the configuration corresponding to the result of the execution an IL code after a given number of steps. This function corresponds to the definition given in Figure 5. We also proved that the relational definition and functional one are equivalent. This is given by the lemmas `il_exec_seq_exec` and `il_exec_exec_seq` of Figure 5.

```
Fixpoint il_exec_seq n p cf : ILConf :=
  if n is n'.+1 then il_exec_seq n' p (il_transf p cf) else cf.
Lemma il_exec_seq_exec : forall n p cf cf', cf' = il_exec_seq n p cf ->
  size p <= cf'.2 -> il_exec p cf cf'.
Lemma il_exec_exec_seq : forall p cf cf', il_exec p cf cf' ->
  exists n, il_exec_seq n p cf = cf'.
```

Fig. 5. Coq definition of the IL program execution function and equivalence proofs

4 Sequential Function Charts

The SFC language is a graphical language for modeling PLC. It is part of the IEC 61131–3 standard and frequently used together with IL and other languages of this standard. SFC are used to describe the overall control flow structure of a system. Due to the graphical nature of the language, we have written a tool which generates Coq representations from graphical SFC models.

The parts of the standard describing SFC leave a few semantical aspects open to the implementation of the PLC modeling and code generation tool. In cases where the semantics is not well defined by the standard we have adapted our semantics to be compatible with the EasyLab [1] tool. EasyLab is a tool that allows the graphical modeling of PLC and C code generation. The description given in this work follows the description given in [4].

4.1 Syntax

Syntactically we represent an SFC as a tuple $(S, S_0, T, A, F, V, Val_V)$. It comprises a set of steps S and a set of transitions T between them. A step is a system location which may either be active or inactive in an actual system state, it can be associated with SFC action blocks from a set A. These perform sets of operations and can be regarded as functors that update functions representing memory. Memory is represented by a function from a set of variables V to a set of their possible values Val_V. The mapping of steps to sets of action blocks is done by the function F.

In our SFC framework, action blocks are described using the IL semantics defined in the previous section. We have established functions that allow conversion of SFC states into IL state and vice versa. Thus, the execution of an action block comprises the following steps:

- Conversion of the SFC state into an IL state
- Execution of the IL program associated with the action block using the semantics from Section 3.
- Update of the SFC state by using the final IL state.

A transition is a tuple (S_{in}, g, S_{out}). It features a set of steps that have to be enabled $S_{in} \subseteq S$ in order to take the transition. It features a guard g that has to be evaluated to true for the given system state. The guard g is a function from system memory to a truth value – in Coq we formalize this as a function to the *Prop* datatype. A transition may have multiple successor steps $S_{out} \subseteq S$. The types Val_V that are formalized in our SFC language comprise different integer types and boolean values. The set of SFC steps includes also a set $S_0 \subseteq S$ representing the initially active steps.

Figure 6 shows an example of an SFC structure realizing a loop with a conditional branch. The execution starts with an initialization step *init*. After it has been processed control may pass to either *Step2* or to a step *Return*. Once *Step2* has been processed control is passed to *init* again.

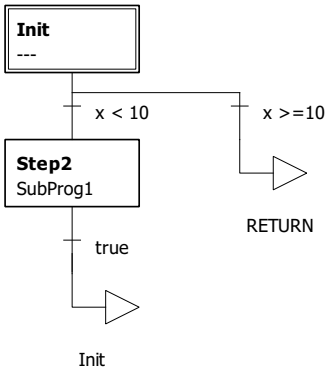


Fig. 6. A loop in the SFC language

Please note that in addition to loops and branches, SFC allows also the definition of parallel processing and synchronization of control. This is due to the multiple successor and predecessor steps in a transition.

The Coq realization of the SFC syntax follows the presented description. For compatibility with the EasyLab tool and to ease generation we distinguish between steps and step identifiers in our Coq files, thereby introducing some level of indirection.

4.2 Semantics

Semantically the execution of an SFC encounters states, which are (m, s, a) tuples. They are characterized by a memory state m, the function from variables to their values, a set of active steps s and a set of active action blocks a that need to be processed.

The semantics is defined by a state transition system which comprises two kinds of rules:

1. A rule for processing an action block from the set of active action blocks a. This corresponds to updating the memory state and removing the processed action block from a.
2. A rule for performing a state transition. The effect on the system state is that some steps are deactivated, others are activated. Each transition needs a guard that can be evaluated to true and a set of active steps. Furthermore, we require that all pending action blocks of a step that is to be deactivated have been executed.

It is custom to specify the timing behavior of a step by time slices: a (maximal) execution time associated with it. In our work, this is realized using special variables that represent time.

5 Tool Support for PLC Verification and Proving Principles

For the generation of graphical SFC representations and reasoning about them we have created a tool (CertPLC, described in a report [3]). It is implemented in Java and uses SFC files built with a graphical PLC configuration environment: EasyLab [1]. The text-based IL code can be imported directly into the generated file. In this section we describe our tool's architecture, usage scenarios and frequently used principles for proving properties.

5.1 The CertPLC Tool

Figure 7 shows the CertPLC ingredients and their interconnections. In an invocation of the tool framework an SFC model is given to a **representation generator** which generates a Coq representation out of it. This is included in one or several files containing the model specific parts of the semantics of the

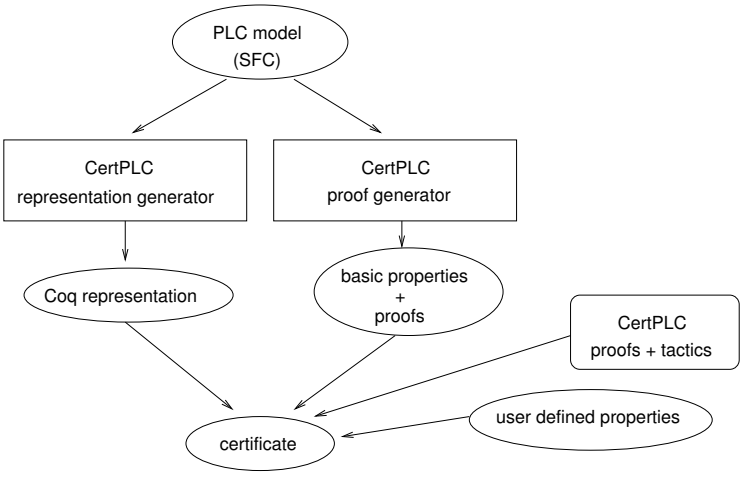


Fig. 7. CertPLC overview

SFC model. The Coq representation is human readable and can be validated against the original graphical SFC specification by experienced users. No representation generation is required for IL, since IL is already a textual format which can be used directly within the Coq proof assistant.

The same SFC model is given to a **proof generator** which generates Coq proof scripts that contain lemmas and their proofs for some basic properties that state important facts needed for machine handling of the proofs of more advanced properties. For example a proof script is generated for a fact that the set of active action blocks in all reachable states of the PLC system does contain only action blocks specified by the syntactic PLC descriptions. The PLC shows only behavior achieved by combining these action blocks.

One goal of CertPLC is the generation of Coq files – a certificate – that certifies a property of a PLC. For this, one needs to formalize the desired property. The property is proved in Coq by using a provided tactic or a hand written proof script. We provide a collection of some **proofs and tactics**. This is a kind of library to be used in our proofs. The Coq system description, used lemmas and their proofs, and the property and its proof form a certificate.

5.2 Proof Structure for Inductive Properties

As stated above, some inductive properties are already generated together with the Coq representation generation. Others can also be proven by using the following scheme: We start with an inductive invariant property I and an SFC description of a PLC SFC. Following the ideas presented in [5] the structure of a proof contained in our certificates is realized by generated proof scripts, generic lemmas and tactics. They establish a proof principle that proves the following goal:

$$\forall\ s\ .\ s \in Reachable_{SFC} \implies I(s)$$

$Reachable_{SFC}$ is the inductively defined set of reachable states, $[\![SFC]\!]$ specifies the state transition relation (cf. Section 5). First we perform an induction using the induction rule of the set of reachable states. This rule is automatically established by Coq when defining inductive sets. After the application the following subgoals are left open:

1. $I(s_0)$ for initial states s_0,
2. $I(s) \wedge (s, s') \in [\![SFC]\!] \implies I(s')$

The first goal can be solved by some relatively simple tactic which just checks that all conditions derived from I are fulfilled in the initial states.

For the second goal the certificate realizes a proof script which – in order to allow efficient certificate checking – performs most importantly the following operations:

- Splitting of conjunctions in invariants into independently verifiable invariants.
- Splitting of disjunctions in invariants into two independently verifiable subgoals.
- Normalizing arithmetic expressions and expressions that make distinctions on active steps in the SFC.
- Exhaustive case distinctions on possible transitions. Each case distinction corresponds to one transition in the control flow graph of the SFC. A typical case can have the following form:

  Precondition on states associated with a case distinction
  Transition condition associated with a case distinction
  Conditions on possible reachable states after one transition
  $\implies$
  Property holds for succeeding states

  The elements in such a goal can feature arithmetic constraints, which can be split into further cases.

Some of the cases that occur can have contradictions in the hypothesis. For example one can imagine an arithmetic constraint for a variable from a precondition of a state contradicting with a condition on a transition. These contradictions result from the fine granularity of our case distinctions. Some effort can be spent to eliminate contradicting cases as soon as possible (cf. [5]) which can speed up the checking process.

6 Case Study

Figure 8 shows an overview of the SFC structure of a PLC program that controls a sorting station on the left side and a picture of the sorting station itself on

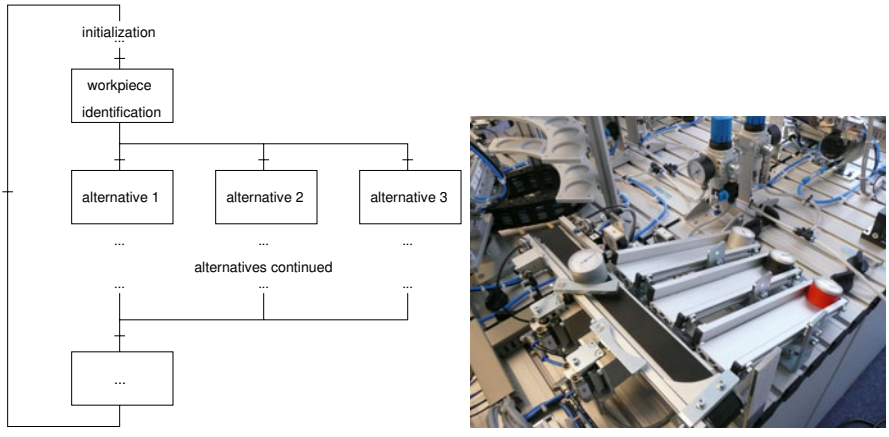


Fig. 8. Sorting machine overview

the right side. Work pieces are transported to two sensors. Based on the values observed by these sensors, a work piece is handled in a different way. The sensor observation is done in the step *workpiece identification*. The handling is done by choosing one of the three alternatives. We have modeled this system in EasyLab and generated the Coq representation of the SFC structure for this case study using CertPLC. We have imported the IL programs describing the actions which are taken at the different SFC steps.

Based on this, we have verified that consistency conditions hold. These comprise:

- The verification of inductive invariant based properties. This is described in Sections 5.1 and 5.2.
- The verification of non-inductive properties. During the conduction of the case study it turned out that non-inductive properties like: Identification of a certain work piece implies treatment in a work piece specific way and this occurs within a fixed amount of execution steps, are also of relevance. Mutual exclusion properties of work piece treatment can be proved by doing these work piece specific proofs for all kinds of possible work pieces, first, and using these results for proving the mutual exclusion property.

Proofs for are done in a modular fashion: we verify the effect of IL parts in the PLC execution and use these proofs to derive facts on the execution of several SFC steps.

Figure 9 shows an example of Coq code + pseudo code to give a look and feel on the nature of our proof goals. Given a concrete workpiece and conditions on a state x which corresponds to a state just before the workpiece identification. A succeeding state x' in the SFC language has to fulfill requirements on variable values m' the set of currently active steps S' and currently active actions A' after a certain execution time. In our example it involves several single SFC

```
conditions on workpiece
->
let '(m,S,A):= x in
  (conditions on m /\ S = SWorkPieceId::nil /\ A = AWorkPieceId::nil
      )
->
...
transition conditions between x x'
...
->
let '(m',S',A'):= x' in
  (some conditions on m' /\ S' = S13::nil /\ A' = nil))
```

Fig. 9. Constraint in Coq

state transitions (state transition rule applications, cf. Section 4) to get from x to x'. This is given in the transition conditions between x and x'. The set of currently active steps in the resulting state x' comprises one step *S13* which corresponds to a step in the first alternative for handling our workpiece.

Evaluation Aspects: Coq representation generation for SFC programs and the import of IL code is feasible for IEC 61131–3 based PLC descriptions that are solely described with these languages. Extending the semantics definition for additional commands which may appear in some PLC descriptions is relatively easy, due to the modularity of our semantics framework.

The inductive proof techniques used in the properties generated by the CertPLC tool and the non-inductive proof techniques used manually in the case study have been successfully applied in previous work which did not deal with PLC (e.g., our own work [5]). Here we have demonstrated their applicability for a realistic PLC. Using our Coq semantics and CertPLC, basic properties of a PLC can be verified by experienced Coq users within several hours. This may result in up to a few hundred lines of proof code for an example as in Figure 9. Common tactic applications are encapsulated into user defined tactics and libraries to further speed this process up, make the scripts smaller, and especially make the approach usable for people who have some knowledge in formal methods but are not Coq experts.

7 Related Work

Formal treatment of PLC and the IEC 61131–3 standard has been discussed by a larger number of authors before. Formalization work on the semantics of the Sequential Function Charts is given in [6,7]. This work was a starting point for our formalization of SFC semantics.

Work on the formal treatment of the FBD language – which is also a part of IEC 61131–3 – can be found in [20,19]. The FBD programs are checked using a model-checking approach.

The approach presented in [16] regards a translation from the IL language to an intermediate representation (SystemC). A SAT instance is generated out of this representation. The correctness of an implementation is guaranteed by equivalence checking with the specification model.

There are plenty of examples of the use of *model checking* for the verification of PLC programs. The paper [2] considers the SFC language. Untimed SFC models are transformed in to the input language of the Cadence SMV tool. Timed SFC models are transformed into timed automata. These can be analyzed by the Uppaal tool. In [13] a semantics of IL is defined using timed automata. The language sub-set contains TON timers but data types are limited to booleans. The formal analysis is performed by the model checker Uppaal.

In [8] an operational semantics of IL is defined. A significant sub-set of IL is supported by this semantics, but it does not include timer instructions. The semantics is encoded in the input language of the model checker Cadence SMV and linear temporal logic (LTL) is used to specify properties of PLC programs.

In contrast to the model checking work, we are using a higher-order theorem prover for our work. In general higher-order theorem provers require a higher level of interaction (we are aiming at overcoming this drawback by generating proof scripts and providing automatic tactics). On the plus side they allow in general richer specifications, abstractions and proofs. In the theorem proving community, there has been some work on the formal analysis of PLC programs. In [17] the theorem prover HOL is used to verify PLC programs written in FBD, SFC and ST languages. In this work, modular verification is used for compositional correctness and safety proofs of programs. For the Coq system, an example of verification of a PLC program with timers is presented in [18]. A quiz machine program is used as an example in this work, but no generic model of PLC programs is formalized. There is also a formalization of a semantics[2] of the LD languages in Coq. This semantics support a sub-set of LD that contains branching instructions. This work is a component of a development environment for PLC.

8 Conclusions and Future Works

Programmable Logic Controller applications can be critical in a safety or economical cost sense. Therefore formal verification of PLC programs does increase the confidence in such applications. In this paper we presented a formal framework for the verification of PLC programs written in the languages IL and SFC. We defined a formal semantics of these two languages in the formal proof system Coq. These semantics are used by the CertPLC tool that automatically generates an SFC formal representation from a graphical representation. Using our

[2] Research report in Korean available at: http://pllab.kut.ac.kr/tr/2009/ldsemantics.pdf

formal semantics, we proved safety properties for a PLC based real industrial demonstrator.

Future Work

The study of other languages from the IEC 61131–3 standard is an interesting subject for future work. Furthermore, we are interested in extending the tool support for verification of properties based on these semantics.

Another perspective of this work is the development of a certified compiler front-end for PLC. This is an ongoing work and we plan to formalize and certify a transformation of PLC programs written in the graphical language LD to IL. This will open the way to the development of a certified compilation chain for PLC. This chain can be build on top of the CompCert C certified compiler [12]. An integration of our formal semantics of PLC and the certified compiler to the EasyLab framework is also an interesting perspective. This can lead to a complete environment for the development of certified PLC programs.

References

1. Barner, S., Geisinger, M., Buckl, C., Knoll, A.: EasyLab: Model-based development of software for mechatronic systems. In: Mechatronic and Embedded Systems and Applications, IEEE/ASME (October 2008)
2. Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., Stursberg, O.: Verification of PLC programs given as sequential function charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 517–540. Springer, Heidelberg (2004)
3. Blech, J.O.: A Tool for the Certification of PLCs based on a Coq Semantics for Sequential Function Charts (2011), http://arxiv.org/abs/1102.3529
4. Blech, J.O., Hattendorf, A., Huang, J.: An Invariant Preserving Transformation for PLC Models. In: IEEE International Workshop on Model-Based Engineering for Real-Time Embedded Systems Design (2011)
5. Blech, J.O., Périn, M.: Generating Invariant-based Certificates for Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS) (accepted)
6. Bornot, S., Huuck, R., Lakhnech, Y., Lukoschus, B.: An Abstract Model for Sequential Function Charts. In: Discrete Event Systems: Analysis and Control, Workshop on Discrete Event Systems (2000)
7. Bornot, S., Huuck, R., Lakhnech, Y., Lukoschus, B.: Verification of Sequential Function Charts using SMV. In: Parallel and Distributed Processing Techniques and Applications (PDPTA 2000). CSREA Press (June 2000)
8. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of PLC programs written in Instruction List. In: IEEE International Conference on Systems, Man, and Cybernetics (2000)
9. The Coq Development Team. The Coq System, http://coq.inria.fr
10. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. INRIA Technical report, http://hal.inria.fr/inria-00258384
11. Huuck, R.: Semantics and Analysis of Instruction List Programs. Electr. Notes Theor. Comput. Sci. (2005)

12. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
13. Mader, A., Wupper, H.: Timed Automaton Models for Simple Programmable Logic Controllers. In: Euromicro Conference on Real-Time Systems (1999)
14. Ould Biha, S.: A formal semantics of PLC programs in Coq. In: 35th IEEE Computer Software and Applications Conference, COMPSAC 2011, Munich (2011)
15. Programmable controllers - Part 3: Programming languages, IEC 61131-3: 1993, International Electrotechnical Commission (1993)
16. Sülflow, A., Drechsler, R.: Verification of plc programs using formal proof techniques. In: Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008), Budapest, pp. 43–50 (2008)
17. Volker, N., Kramer, B.J.: Automated verification of function block-based industrial control systems. Science of Computer Programming 42, 101–113 (2002)
18. Wan, H., Chen, G., Song, X., Gu, M.: Formalization and Verification of PLC Timers in Coq. In: 33rd IEEE Computer Software and Applications Conference, COMPSAC (2009)
19. Yoo, J., Cha, S., Jee, E.: A verification framework for fbd based software in nuclear power plants. In: 15th Asia Pacific Software Engineering Conference (APSEC), Beijing, China, December 3-5 (2008)
20. Yoo, J., Cha, S., Jee, E.: Verification of plc programs written in fbd with vis. Nuclear Engineering and Technology 41(1), 79–90 (2009)

Broadcast Psi-calculi with an Application to Wireless Protocols

Johannes Borgström[1], Shuqin Huang[2], Magnus Johansson[1], Palle Raabjerg[1], Björn Victor[1], Johannes Åman Pohjola[1], and Joachim Parrow[1]

[1] Department of Information Technology, Uppsala University, Sweden
[2] Peking University, China

Abstract. Psi-calculi is a parametric framework for extensions of the pi-calculus, with arbitrary data structures and logical assertions for facts about data. In this paper we add primitives for broadcast communication in order to model wireless protocols. The additions preserve the purity of the psi-calculi semantics, and we formally prove the standard congruence and structural properties of bisimilarity. We demonstrate the expressive power of broadcast psi-calculi by modelling the wireless ad-hoc routing protocol LUNAR and verifying a basic reachability property.

1 Introduction

Psi-calculi is a parametric framework for extensions of the pi-calculus, with arbitrary data structures and logical assertions for facts about data. In psi-calculi (described in Section 2) the purity of the semantics is on par with the original pi-calculus, the generality and expressiveness exceeds many earlier extensions of the pi-calculus, and the meta-theory is proved correct once and for all using the interactive theorem prover Isabelle/Nominal [26].

In order to model wireless communication used in WSN (Wireless Sensor Network) and MANET (Mobile Ad-hoc Network) applications, the concept of broadcast communication is needed, where one transmission can be received by several processes. Broadcast communication cannot be encoded in the pi-calculus [5]; we extend the psi-calculi framework with broadcast primitives (Section 3). The broadcast primitives are added using new operational actions and rules, and new connectivity predicates. We formally prove the congruence properties of bisimilarity and the soundness of structural equivalence laws using the Isabelle/Nominal theorem prover.

The connectivity predicates allow us to model systems with limited reachability, for instance where a transmitter only reaches nodes within a certain range, and systems with changing reachability, for instance due to physical mobility of nodes. In Section 4, we present a technique for treating different generations of connectivity information. Broadcast channels can be globally visible or have limited scope. Scoped channels can be protected from externally imposed connectivity changes, while permitting connectivity changes by processes within the scope of the channel.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 74–89, 2011.

We demonstrate the expressive power of the resulting framework in Section 5, where we provide a model of the LUNAR protocol for routing in ad-hoc wireless networks [24]. The model follows the specification closely, and demonstrates several features of the psi-calculi framework: both unicast and broadcast communication, application-specific data structures and logics, classic unstructured channels as well as pairs corresponding to MAC address and port selector. Our model is significantly more succinct than earlier work [28,27] (ca 30 vs 250 lines). We show an expected basic reachability property of the model: if two network nodes, a sender and a receiver, are both in range of a third node, but not within range of each other, the LUNAR protocol can find a route and transparently handle the delivery of a packet from the sender to the receiver.

We discuss related work on process calculi for wireless broadcast in Section 6, and conclude and present ideas for future work in Section 7.

2 Psi-calculi

This section is a brief recapitulation of psi-calculi; for a more extensive treatment including motivations and examples see [3,4].

We assume a countably infinite set of atomic *names* $\mathcal{N}$ ranged over by $a, b, \ldots, z$. Intuitively, names will represent the symbols that can be scoped, and also represent symbols acting as variables in the sense that they can be subject to substitution. A *nominal set* [18,6] is a set equipped with a formal notion of what it means for a name a to occur in an element X of the set, written $a \in \mathrm{n}(X)$ (often pronounced as "a is in the support of X"). We write $a\#X$, pronounced "a is fresh for X", for $a \notin \mathrm{n}(X)$, and if A is a set of names we write $A\#X$ to mean $\forall a \in A \,.\, a\#X$. In the following $\tilde{a}$ means a finite sequence of names, $a_1, \ldots, a_n$. The empty sequence is written ϵ and the concatenation of $\tilde{a}$ and $\tilde{b}$ is written $\tilde{a}\tilde{b}$. When occurring as an operand of a set operator, $\tilde{a}$ means the corresponding set of names $\{a_1, \ldots, a_n\}$. We also use sequences of other nominal sets in the same way.

A *nominal datatype* is a nominal set together with a set of functions on it. In particular we shall consider substitution functions that substitute elements for names. If X is an element of a datatype, the *substitution* $X[\tilde{a} := \tilde{Y}]$ is an element of the same datatype as X. There is considerable freedom in the choice of functions and substitutions; see [3,4] for details.

A psi-calculus is defined by instantiating three nominal data types and four operators:

Definition 1 (Psi-calculus parameters). *A psi-calculus requires the three (not necessarily disjoint) nominal data types: the (data) terms* $\mathbf{T}$*, ranged over by* M, N*, the conditions* $\mathbf{C}$*, ranged over by* φ*, the assertions* $\mathbf{A}$*, ranged over by* Ψ*, and the four equivariant operators:*

$$\begin{array}{ll} \leftrightarrow : \mathbf{T} \times \mathbf{T} \to \mathbf{C} & \textit{Channel Equivalence} \\ \otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A} & \textit{Composition} \\ \mathbf{1} : \mathbf{A} & \textit{Unit} \\ \vdash \subseteq \mathbf{A} \times \mathbf{C} & \textit{Entailment} \end{array}$$

and substitution functions $[\widetilde{a} := \widetilde{M}]$, substituting terms for names, on each of $\mathbf{T}$, $\mathbf{C}$ and $\mathbf{A}$.

The binary functions above will be written in infix. Thus, if M and N are terms then $M \leftrightarrows N$ is a condition, pronounced "M and N are channel equivalent" and if Ψ and Ψ' are assertions then so is $\Psi \otimes \Psi'$. Also we write $\Psi \vdash \varphi$, "Ψ entails φ", for $(\Psi, \varphi) \in \vdash$.

We say that two assertions are equivalent, written $\Psi \simeq \Psi'$ if they entail the same conditions, i.e. for all φ we have that $\Psi \vdash \varphi \Leftrightarrow \Psi' \vdash \varphi$. We impose certain requisites on the sets and operators. In brief, channel equivalence must be symmetric and transitive, $\otimes$ must be compositional with regard to $\simeq$, and the assertions with $(\otimes, \mathbf{1})$ form an abelian monoid modulo $\simeq$. For details see [3].

A *frame* F can intuitively be thought of as an assertion with local names: it is of the form $(\nu\widetilde{b})\Psi$ where $\widetilde{b}$ is a sequence of names that bind into the assertion Ψ. We use F, G to range over frames. We overload Ψ to also mean the frame $(\nu\epsilon)\Psi$ and $\otimes$ to composition on frames defined by $(\nu\widetilde{b}_1)\Psi_1 \otimes (\nu\widetilde{b}_2)\Psi_2 = (\nu\widetilde{b}_1\widetilde{b}_2)(\Psi_1 \otimes \Psi_2)$ where $\widetilde{b}_1\#\widetilde{b}_2, \Psi_2$ and vice versa. We write $(\nu c)((\nu\widetilde{b})\Psi)$ for $(\nu c\widetilde{b})\Psi$.

Alpha equivalent frames are identified. We define $F \vdash \varphi$ to mean that there exists an alpha variant $(\nu\widetilde{b})\Psi$ of F such that $\widetilde{b}\#\varphi$ and $\Psi \vdash \varphi$. We also define $F \simeq G$ to mean that for all φ it holds that $F \vdash \varphi$ iff $G \vdash \varphi$.

Definition 2 (Psi-calculus agents). *Given valid psi-calculus parameters as in Definition 1, the psi-calculus* agents*, ranged over by $P, Q, \ldots$, are of the following forms.*

$\mathbf{0}$	Nil
$\overline{M}N\,.\,P$	Output
$\underline{M}(\lambda\widetilde{x})N\,.\,P$	Input
$\mathbf{case}\ \varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\mid\Psi\mid\!)$	Assertion

Restriction binds a in P and Input binds $\widetilde{x}$ in both N and P. We identify alpha equivalent agents. An assertion is guarded *if it is a subterm of an Input or Output. An agent is* assertion guarded *if it contains no unguarded assertions. An agent is* well-formed *if in $\underline{M}(\lambda\widetilde{x})N.P$ it holds that $\widetilde{x} \subseteq \mathrm{n}(N)$ is a sequence without duplicates, that in a replication $!P$ the agent P is assertion guarded, and that in* $\mathbf{case}\ \varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$ *the agents P_i are assertion guarded.*

The agent $\mathbf{case}\ \varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$ is sometimes abbreviated as $\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}$, or if $n = 1$ as $\mathbf{if}\ \varphi_1\ \mathbf{then}\ P_1$.

The *frame* $\mathcal{F}(P)$ *of an agent* P is defined inductively as follows:

$$\mathcal{F}(\underline{M}(\lambda\widetilde{x})N\,.\,P) = \mathcal{F}(\overline{M}\,N\,.\,P) = \mathcal{F}(\mathbf{0}) = \mathcal{F}(\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) = \mathcal{F}(!P) = \mathbf{1}$$
$$\mathcal{F}((\!\mid\Psi\mid\!)) = (\nu\epsilon)\Psi$$
$$\mathcal{F}(P \mid Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q)$$
$$\mathcal{F}((\nu b)P) = (\nu b)\mathcal{F}(P)$$

Table 1. Structured operational semantics. Symmetric versions of Com and Par are elided. In the rule Com we assume that $\mathcal{F}(P) = (\nu\widetilde{b}_P)\Psi_P$ and $\mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q$ where $\widetilde{b}_P$ is fresh for all of $\Psi, \widetilde{b}_Q, Q, M$ and P, and that $\widetilde{b}_Q$ is similarly fresh. In the rule Par we assume that $\mathcal{F}(Q) = (\nu\widetilde{b}_Q)\Psi_Q$ where $\widetilde{b}_Q$ is fresh for Ψ, P and α. In Open the expression $\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

$$\textsc{In}\ \frac{\Psi \vdash K \leftrightarrow M}{\Psi \rhd \underline{M}(\lambda\widetilde{y})N \,.\, P \xrightarrow{\underline{K}\,N[\widetilde{y}:=\widetilde{L}]} P[\widetilde{y} := \widetilde{L}]} \qquad \textsc{Out}\ \frac{\Psi \vdash M \leftrightarrow K}{\Psi \rhd \overline{M}\,N \,.\, P \xrightarrow{\overline{K}N} P}$$

$$\textsc{Case}\ \frac{\Psi \rhd P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow{\alpha} P'}$$

$$\textsc{Com}\ \frac{\begin{array}{c}\Psi \otimes \Psi_P \otimes \Psi_Q \vdash M \leftrightarrow K \\ \Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{M}(\nu\tilde{a})N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{K}\,N} Q'\end{array}}{\Psi \rhd P \mid Q \xrightarrow{\tau} (\nu\widetilde{a})(P' \mid Q')}\ \tilde{a}\#Q$$

$$\textsc{Par}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathrm{bn}(\alpha)\#Q \qquad \textsc{Scope}\ \frac{\Psi \rhd P \xrightarrow{\alpha} P'}{\Psi \rhd (\nu b)P \xrightarrow{\alpha} (\nu b)P'}\ b\#\alpha, \Psi$$

$$\textsc{Open}\ \frac{\Psi \rhd P \xrightarrow{\overline{M}(\nu\tilde{a})N} P'}{\Psi \rhd (\nu b)P \xrightarrow{\overline{M}(\nu\tilde{a}\cup\{b\})N} P'}\ \begin{array}{l} b\#\widetilde{a}, \Psi, M \\ b \in \mathrm{n}(N)\end{array} \qquad \textsc{Rep}\ \frac{\Psi \rhd P \mid !P \xrightarrow{\alpha} P'}{\Psi \rhd !P \xrightarrow{\alpha} P'}$$

The *actions* ranged over by α, β are of the following three kinds:

Output $\overline{M}(\nu\tilde{a})N$ where $\alpha \subseteq \mathrm{n}(N)$, Input $\underline{M}\,N$, and Silent τ. Here we refer to M as the *subject* and N as the *object*. We define $\mathrm{bn}(\overline{M}(\nu\tilde{a})N) = \tilde{a}$, and $\mathrm{bn}(\alpha) = \emptyset$ if α is an input or τ. We also define $\mathrm{n}(\tau) = \emptyset$ and $\mathrm{n}(\alpha) = \mathrm{n}(M) \cup \mathrm{n}(N)$ for the input and output actions.

Definition 3 (Transitions). *A* transition *is written $\Psi \rhd P \xrightarrow{\alpha} P'$, meaning that in the environment Ψ the well-formed agent P can do an α to become P'. The transitions are defined inductively in Table 1. We write $P \xrightarrow{\alpha} P'$ without an assertion to mean $\mathbf{1} \rhd P \xrightarrow{\alpha} P'$.*

Agents, frames and transitions are identified by alpha equivalence. In a transition the names in $\mathrm{bn}(\alpha)$ bind into both the action object and the derivative, therefore $\mathrm{bn}(\alpha)$ is in the support of α but not in the support of the transition. This means that the bound names can be chosen fresh, substituting each occurrence in both the object and the derivative.

Definition 4 (Strong bisimulation). *A* strong bisimulation $\mathcal{R}$ *is a ternary relation on assertions and pairs of agents such that $\mathcal{R}(\Psi, P, Q)$ implies all of*

1. *Static equivalence: $\Psi \otimes \mathcal{F}(P) \simeq \Psi \otimes \mathcal{F}(Q)$*
2. *Symmetry: $\mathcal{R}(\Psi, Q, P)$*

3. *Extension of arbitrary assertion:* $\forall \Psi'.\ \mathcal{R}(\Psi \otimes \Psi', P, Q)$
4. *Simulation: for all* α, P' *such that* $\mathrm{bn}(\alpha)\#\Psi, Q$ *there exists a* Q' *such that*

$$\Psi \rhd P \xrightarrow{\alpha} P' \Longrightarrow \Psi \rhd Q \xrightarrow{\alpha} Q' \wedge \mathcal{R}(\Psi, P', Q')$$

We define $P \mathrel{\dot{\sim}_\Psi} Q$ *to mean that there exists a bisimulation* $\mathcal{R}$ *such that* $\mathcal{R}(\Psi, P, Q)$*, and write* $\dot{\sim}$ *for* $\dot{\sim}_{\mathbf{1}}$.

Strong bisimulation is preserved by all operators except input prefix and satisfies the expected algebraic laws such as scope extension, for details see [3,4].

3 Broadcast Semantics

In this section we extend the unicast psi-calculi of the previous section with a broadcast semantics that models wireless (i.e., synchronous and unreliable) broadcast. As an example, assume that the connectivity information Ψ allows receivers M_1 and M_2 to listen to channel K. We would then expect the following transition: $\Psi \rhd \overline{K}\,N.P \mid \underline{M_2}(x).Q \mid \underline{M_3}(y).R \xrightarrow{\overline{K}\,N} P \mid Q[x := N] \mid R[y := N]$.

To allow connectivity to depend on assertions, and to permit broadcast channels to be computed at run-time, we assume a psi-calculus with the following extra predicates:

Definition 5 (Extra predicates for broadcast)

$$\begin{array}{ll} \dot{\prec} : \mathbf{T} \times \mathbf{T} \to \mathbf{C} & \textit{Output Connectivity} \\ \dot{\succ} : \mathbf{T} \times \mathbf{T} \to \mathbf{C} & \textit{Input Connectivity} \end{array}$$

The first predicate, $M \mathrel{\dot{\prec}} K$, is pronounced "$M$ is out-connected to K" and means that an output prefix $\overline{M}\,N$ can result in a broadcast on channel K. The second, $K \mathrel{\dot{\succ}} M$, is pronounced "$M$ is in-connected to K" and means that an input prefix $\underline{M}(\lambda\tilde{x})N$ can receive broadcast messages from channel K. As usual in broadcast calculi, the receivers need to be using the same broadcast channel as the sender in order to receive a message.

As an example, we can model routing table lookup: if tab is a term corresponding to a routing table we can let $\Psi \vdash \mathtt{lookup}(tab, id) \mathrel{\dot{\prec}} ch$ be true if (id, ch) appears in tab. We can also model connectivity: if Ψ contains connectivity information between receivers n and channels ch we may let $\Psi \vdash ch \mathrel{\dot{\succ}} \mathtt{rcv}(n, ch)$ be true if n is connected to ch according to Ψ.

In contrast to unicast connectivity, we do not require broadcast connectedness to be symmetric or transitive, so in particular $M \mathrel{\dot{\prec}} K$ might not be equivalent to $K \mathrel{\dot{\succ}} M$. Instead, for technical reasons related to scope extension, broadcast channels must have no greater support than the input and output prefixes that can make use of them.

Definition 6 (Requirements for broadcast)

1. $\Psi \vdash M \mathrel{\dot{\prec}} K \implies \mathsf{n}(M) \supseteq \mathsf{n}(K)$
2. $\Psi \vdash K \mathrel{\dot{\succ}} M \implies \mathsf{n}(K) \subseteq \mathsf{n}(M)$

Table 2. Operational broadcast semantics. A symmetric version of BrCom is elided. In rules BrCom and BrMerge we assume that $\mathcal{F}(P) = (\nu \widetilde{b}_P)\Psi_P$ and $\mathcal{F}(Q) = (\nu \widetilde{b}_Q)\Psi_Q$ where $\widetilde{b}_P$ is fresh for $P, \widetilde{b}_Q, Q, K$ and Ψ, and that $\widetilde{b}_Q$ is fresh for $Q, \widetilde{b}_P, P, K$ and Ψ.

$$\textsc{BrOut}\ \frac{\Psi \vdash M \dot{\prec} K}{\Psi \rhd \overline{M}\, N \,.\, P \xrightarrow{!\overline{K}\, N} P} \qquad \textsc{BrIn}\ \frac{\Psi \vdash K \dot{\succ} M}{\Psi \rhd \underline{M}(\lambda \widetilde{y}) N \,.\, P \xrightarrow{?\underline{K}\, N[\widetilde{y} := \widetilde{L}]} P[\widetilde{y} := \widetilde{L}]}$$

$$\textsc{BrMerge}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{?\underline{K}\, N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{?\underline{K}\, N} Q'}{\Psi \rhd P \mid Q \xrightarrow{?\underline{K}\, N} P' \mid Q'}$$

$$\textsc{BrCom}\ \frac{\Psi_Q \otimes \Psi \rhd P \xrightarrow{!\overline{K}\, (\nu \widetilde{a}) N} P' \qquad \Psi_P \otimes \Psi \rhd Q \xrightarrow{?\underline{K}\, N} Q'}{\Psi \rhd P \mid Q \xrightarrow{!\overline{K}\, (\nu \widetilde{a}) N} P' \mid Q'}\ \widetilde{a} \# Q$$

$$\textsc{BrClose}\ \frac{\Psi \rhd P \xrightarrow{!\overline{K}\, (\nu \widetilde{a}) N} P'}{\Psi \rhd (\nu b) P \xrightarrow{\tau} (\nu b)(\nu \widetilde{a}) P'}\ \begin{array}{l} b \in \mathsf{n}(K) \\ b \# \Psi \end{array}$$

Definition 7 (Transitions of Broadcast Psi). *To the actions of psi-calculi we add broadcast input, written* $?\underline{K}\, N$ *for a reception of* N *on* K*, and broadcast output, written* $!\overline{K}\, (\nu \widetilde{a}) N$ *for a broadcast of* N *on* K*, with names* $\widetilde{a}$ *fresh in* K*. As before, we omit* $(\nu \widetilde{a})$ *when* $\widetilde{a}$ *is empty, and in examples we omit* N *when it is not relevant. The transitions of well-formed agents are defined inductively in Tables 2 and 1, where we let* α *range over both unicast and broadcast actions.*

The rule BrOut, allows transmission on a broadcast channel K that the subject M of an output prefix is out-connected to. Similarly, the rule BrIn allows input from a broadcast channel K that the subject M of an input prefix is in-connected to. When two parallel processes both receive a broadcast on the same channel, the rule BrMerge combines the two actions. This rule is necessary to ensure the associativity of parallel composition. After a broadcast communication using BrCom, the resulting action is the original transmission. This is different from the unicast Com rule, where a communication yields an internal action τ. Finally, rule BrClose states that a broadcast transmission does not reach beyond its scope. This allows for broadcasting on restricted channels. Dually, the Res rule (of Table 1) ensures that broadcast receivers on restricted channels cannot proceed unless a message is sent. We allow the Open rule to also apply to broadcast output actions, in order to communicate scoped data. The Par rule allows for broadcasts to bypass a process, as in most other broadcast calculi for wireless systems.

We have developed a meta-theory for broadcast psi-calculi. In the following we restrict attention to well-formed agents. The expected compositionality properties of strong bisimilarity hold:

Theorem 8 (Congruence properties of strong bisimulation). *For all Ψ:*

$$\begin{aligned}
P \mathrel{\dot\sim_\Psi} Q &\implies P \mid R \mathrel{\dot\sim_\Psi} Q \mid R \\
P \mathrel{\dot\sim_\Psi} Q &\implies (\nu a)P \mathrel{\dot\sim_\Psi} (\nu a)Q \qquad \textit{if } a\#\Psi \\
P \mathrel{\dot\sim_\Psi} Q &\implies !P \mathrel{\dot\sim_\Psi} !Q \qquad \textit{if } P, Q \textit{ assertion guarded} \\
\forall i. P_i \mathrel{\dot\sim_\Psi} Q_i &\implies \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \mathrel{\dot\sim_\Psi} \mathbf{case}\ \widetilde{\varphi} : \widetilde{Q} \\
P \mathrel{\dot\sim_\Psi} Q &\implies \overline{M}\,N\,.\,P \mathrel{\dot\sim_\Psi} \overline{M}\,N\,.\,Q \\
(\forall \widetilde{L}.\ P[\widetilde{x} := \widetilde{L}] \mathrel{\dot\sim_\Psi} Q[\widetilde{x} := \widetilde{L}]) &\implies \underline{M}(\lambda\widetilde{x})N\,.\,P \mathrel{\dot\sim_\Psi} \underline{M}(\lambda\widetilde{x})N\,.\,Q
\end{aligned}$$

As usual in channel-passing calculi, bisimulation is not a congruence for input prefix. We can characterise strong bisimulation congruence in the usual way.

Definition 9 (Strong Congruence). *$P \sim_\Psi Q$ iff for all sequences σ of substitutions it holds that $P\sigma \mathrel{\dot\sim_\Psi} Q\sigma$. We write $P \sim Q$ for $P \sim_{\mathbf{1}} Q$.*

Theorem 10. *Strong congruence $\sim_\Psi$ is a congruence for all Ψ.*

The standard structural laws hold for strong congruence.

Theorem 11 (Structural equivalence). *Assume that $a\#Q, \widetilde{x}, M, N, \widetilde{\varphi}$. Then*

$$\begin{array}{cc}
\mathbf{case}\ \widetilde{\varphi} : \widetilde{(\nu a)P} \sim (\nu a)\mathbf{case}\ \widetilde{\varphi} : \widetilde{P} & (\nu a)\mathbf{0} \sim \mathbf{0} \\
\underline{M}(\lambda\widetilde{x})N\,.\,(\nu a)P \sim (\nu a)\underline{M}(\lambda\widetilde{x})(N)\,.\,P & Q \mid (\nu a)P \sim (\nu a)(Q \mid P) \\
\overline{M}\,N\,.\,(\nu a)P \sim (\nu a)\overline{M}\,N\,.\,P & (\nu b)(\nu a)P \sim (\nu a)(\nu b)P \\
P \mid (Q \mid R) \sim (P \mid Q) \mid R & !P \sim P \mid !P \\
P \mid Q \sim Q \mid P & P \sim P \mid \mathbf{0}
\end{array}$$

Theorems 8, 10 and 11 give us assurance that any broadcast psi-calculus has a compositional labelled bisimilarity that respects important structural laws. The proofs [21] are formally verified in the interactive theorem prover Isabelle/Nominal. The full formalisation of broadcast psi-calculi amounts to ca 33000 lines of Isabelle code, of which about 21000 lines are re-used from our earlier work [4]. The fact that the BrComm rule defers the closing of the communication to BrClose causes most of the added complications.

4 Modelling Network Topology Changes

When modelling wireless protocols, one important concern is dealing with connectivity changes. We here give a general description of a method of modelling different connectivity configurations using assertions.

The idea is to allow for different generations of assertions by tagging each part of an assertion with a generation number. Only the most recent generation is used; a generation is made obsolete by adding an assertion from a later generation. We here consider broadcast connectivity, but this technique can also be used in other scenarios where there is a need to retract assertions.

In the following we assume a set of broadcast terms $\mathbf{B} \subseteq \mathbf{T}$; we let B, B' range over elements of $\mathbf{B}$. For simplicity, we assume that no rewriting happens

in broadcast output, i.e., that $\dot{\prec}$ is the equality relation of $\mathbf{B}$. Assertions are finite sets of connectivity information, labelled with a generation, with set union as assertion composition $\otimes$ and the empty set as the unit assertion. Formally,

$$\begin{aligned}\mathbf{C} &\triangleq \{\mathsf{currentGeneration}(i) : i \in \mathbb{N}\} \cup \\ &\quad \{K \mathrel{\dot{\succ}} M : K, M \in \mathbf{T}\} \cup \{M \mathrel{\dot{\prec}} K : K, M \in \mathbf{T}\} \\ \mathbf{A} &\triangleq \mathcal{P}_{\text{fin}}(\{\langle i, K \mathrel{\dot{\succ}} M\rangle : i \in \mathbb{N},\ K, M \in \mathbf{T}\} \cup \{\langle i, 0\rangle : i \in \mathbb{N}\})\end{aligned}$$

$\Psi \vdash \mathsf{currentGeneration}(i)$ if $\forall \langle j, *\rangle \in \Psi\ .\ j \leq i$ and $\exists \langle j, *\rangle \in \Psi\ .\ i = j$
where $*$ is $B \mathrel{\dot{\succ}} B'$ or 0
$\Psi \vdash B \mathrel{\dot{\prec}} B'$ if $B = B'$
$\Psi \vdash B \mathrel{\dot{\succ}} B'$ if $\langle i, B \mathrel{\dot{\succ}} B'\rangle \in \Psi$ and $\mathsf{n}(B) \subseteq \mathsf{n}(B')$ and $\Psi \vdash \mathsf{currentGeneration}(i)$

The condition $\mathsf{currentGeneration}(i)$ is used to test if i is the most recent generation. The assertion $\{\langle i, B \mathrel{\dot{\succ}} B'\rangle\}$ states that B' is in-connected to B in generation i if $\mathsf{n}(B) \subseteq \mathsf{n}(B')$, while the assertion $\{\langle i, 0\rangle\}$ states that nothing is connected in generation i.

As an example, we can define a topology controller (assuming a suitable encoding of the τ prefix):

$$T = (\!|\{\langle 1, 0\rangle\}|\!) \mid \tau\,.\,\big((\!|\{\langle 2, K \mathrel{\dot{\succ}} M\rangle, \langle 2, K \mathrel{\dot{\succ}} N\rangle\}|\!) \mid \tau\,.\,((\!|\{\langle 3, K \mathrel{\dot{\succ}} M\rangle\}|\!))\big)$$

In the process $P \mid T$, P can broadcast on K while T manages the topology. Initially $\mathcal{F}(T) = \{\langle 1, 0\rangle\}$ and the broadcast is disconnected; after $T \xrightarrow{\tau} T'$ then $\mathcal{F}(T') = \{\langle 1, 0\rangle, \langle 2, K \mathrel{\dot{\succ}} M\rangle, \langle 2, K \mathrel{\dot{\succ}} N\rangle\}$ and a broadcast on K can be received on both M and N, and after $T' \xrightarrow{\tau} T''$ then a broadcast can be received only on M, since $\mathcal{F}(T'') = \{\langle 1, 0\rangle, \langle 2, K \mathrel{\dot{\succ}} M\rangle, \langle 2, K \mathrel{\dot{\succ}} N\rangle, \langle 3, K \mathrel{\dot{\succ}} M\rangle\}$.

5 The LUNAR Protocol in Psi

In this section we present a model of the LUNAR routing protocol for mobile ad-hoc networks [24,25]. LUNAR is intended for small wireless networks, ca 15 nodes, with a network diameter of 3 hops. It does not handle route reparation, caching etc, and routes must be re-established every few seconds. It is reasonably simple in comparison to many other ad-hoc routing protocols, and allows us to focus on properties such as dynamic connectivity and broadcasting. It has previously been verified in [28,27] using SPIN and UPPAAL; our model is significantly shorter and at an abstraction level closer to the specification.

The LUNAR protocol is at "layer 2.5", between the link and network layers in the Internet protocol stack. Addressing is by pairs of MAC/Ethernet addresses and 64-bit selectors, similarly to the IP address and port number used in UDP/TCP. The selectors are used to find the appropriate packet handler through the FIB (Forwarding Information Base) table.

Below, we define a psi-calculus for modelling the LUNAR protocol. In an effort to keep our model simple we abstract from details such as TTL fields in

messages, optional protocol fields, globally unique host identifiers, etc. We do not deal with time at all.

Channels are of two kinds: broadcast channels are terms node_i with (for simplicity) empty support, whose connectivity is given by the $\dot{\succ}$ and $\dot{\prec}$ predicates as defined in Section 4, and unicast channels which are pairs $\langle sel, mac\rangle$ where *sel* is a selector name and *mac* is a MAC address name. The *mac* part can also be a $\mathsf{RouteOf}(node, ip)$ construction, which looks up the route of an IP address *ip* in the routing table of the node *node*. Special channels $\langle \mathsf{delivered}, \mathsf{node}_i\rangle$ are used to signal delivery of a packet to the IP layer. Assertions record requests originated at the local node using $\mathsf{Redirected}(node, sel)$ and specify found routes using $\mathsf{HaveRoute}(node, destip, hops, sel)$. The conditions contain predicates for testing if a route has been found ($\mathsf{HaveRoute}(node, ip)$), if a selector has been used for a request originating at the local node ($\mathsf{Redirected}(node, sel)$), and to extract the forwarder of a route ($\langle x, \mathsf{RouteOf}(node, ip)\rangle \dot{\leftrightarrow} \langle x, ip\rangle$).

LUNAR protocol messages are of two types. The first is a route request message $\mathsf{RREQ}(selector, targetIP, replyTo)$, where the *selector* identifies the request, *targetIP* is the IP address the route should reach, and *replyTo* is the $\langle sel, mac\rangle$ channel the response should be sent to. The second is a route reply message, $\mathsf{RREP}(hops, fwdptr)$), where *hops* is the number of hops to the destination, and *fwdptr* is a forwarding pointer, i.e. a $\langle sel, mac\rangle$ channel where packets can be sent.

The parameters of the psi-calculus for LUNAR extend the general topology psi-calculus in Section 4 as follows. The sets $\mathbf{T}, \mathbf{C}$ and $\mathbf{A}$ recursively include terms in order to be closed under substitution of terms for names.

$$
\begin{aligned}
\mathbf{T} \triangleq\ & \mathcal{N} \cup \{\mathsf{node}_i : i \in \mathbb{N}\} \cup \{\mathsf{delivered}\} \cup \\
& \{\mathsf{RREQ}(Ser, TargIp, Rep) : Ser,\ TargIp,\ Rep \in \mathbf{T}\} \cup \\
& \{\mathsf{RREP}(i, Fwd) : i,\ Fwd \in \mathbf{T}\} \cup \\
& \{\mathsf{RouteOf}(Node, Ip) : Node,\ Ip \in \mathbf{T}\} \cup \\
& \{\langle Sel, N\rangle : Sel, N \in \mathbf{T}\} \cup \{N + 1 : N \in \mathbf{T}\} \cup \{0\} \\
\mathbf{C} \triangleq\ & \{M = N, \mathsf{HaveRoute}(M, N), \mathsf{Redirected}(M, N) : M, N \in \mathbf{T}\} \\
\mathbf{A} \triangleq\ & \mathcal{P}_{\mathrm{fin}}(\{\mathsf{HaveRoute}(M, N_1, i, N_2) : i,\ M, N_1, N_2 \in \mathbf{T}\} \cup \\
& \qquad \{\mathsf{Redirected}(M, N) : M, N \in \mathbf{T}\})
\end{aligned}
$$

$$
\begin{aligned}
\Psi &\vdash a = a,\ a \in \mathcal{N} \\
\Psi &\vdash \langle a, b\rangle \dot{\leftrightarrow} \langle a, b\rangle,\ a, b \in \mathcal{N} \\
\Psi &\vdash \langle \mathsf{delivered}, \mathsf{node}_i\rangle \dot{\leftrightarrow} \langle \mathsf{delivered}, \mathsf{node}_i\rangle,\ i \in \mathbb{N} \\
\Psi \cup \{\mathsf{HaveRoute}(\mathsf{node}_i, a, j, b)\} &\vdash \langle \mathsf{RouteOf}(\mathsf{node}_i, a), x\rangle \dot{\leftrightarrow} \langle b, x\rangle \\
\Psi \cup \{\mathsf{HaveRoute}(\mathsf{node}_i, a, j, b)\} &\vdash \mathsf{HaveRoute}(\mathsf{node}_i, a) \\
\Psi \cup \{\mathsf{Redirected}(\mathsf{node}_i, s)\} &\vdash \mathsf{Redirected}(\mathsf{node}_i, s) \\
\Psi &\vdash \neg\varphi \text{ if } \neg(\Psi \vdash \varphi)
\end{aligned}
$$

Figures 1-7 describe our psi-calculus model of the LUNAR protocol. We use process identifiers to improve the readability of the model. Process identifiers and recursion can be encoded in a standard fashion using replication, see e.g. [22]. In this section we use process declarations of the form $M(\widetilde{N}) \Leftarrow P$, where M is

a process identifier (and also a term, implicitly included in $\mathbf{T}$), $\widetilde{N}$ a list of terms where occurrences of names are binding, and P is a process s.t. $\mathsf{n}(P) \subseteq \mathsf{n}(\widetilde{N})$. In a process, we write $M(\widetilde{N})$ for invoking a process declaration $M(\widetilde{K}) \Leftarrow P$ such that $\widetilde{N} = \widetilde{K}[\widetilde{x} := \widetilde{L}]$ with $\widetilde{x} = \mathsf{n}(\widetilde{K})$, resulting in the process $P[\widetilde{x} := \widetilde{L}]$. We write **if** φ **then** P **else** Q for **case** $\varphi : P \,[]\, \neg\varphi : Q$, and assume a suitable encoding of the τ prefix.

Our model of the protocol closely follows the informal protocol description in [25, Section 4]. Each figure in our model corresponds quite directly to one or more of part 0-5 of the protocol description. To allocate a selector, we simply bind a name; to associate (or bind) a selector to a packet handler we use a replicated process which receives on the unicast channel described by the pair of the selector and our MAC address (see e.g. the second line of the LunARP process declaration in Figure 1). In the informal protocol description [25], the FIB is "abused" by installing a null packet handler for the selector created when sending a route request. This FIB entry is only used to detect and avoid circular forwarding of route requests. We model this by an explicit assertion Redirected and a matching condition. The routing table is modelled using assertions, to show how these can be used as a global data structure. For simplicity we do not model route timeouts and the deletion of routes, but this could be done using the mechanism in Section 4.

The LUNAR procedure for route discovery starts when a node wants to send a message to a node it does not already have a route to (Figure 7, **else** branch). It then (Figure 1) associates a fresh selector with a response packet handler, and broadcasts a Route Request (RREQ) message to its neighbours. A node which receives a RREQ message (Figure 2) for its own IP address sets up a packet handler to deliver IP packets, and includes the corresponding selector in a response Route Reply (RREP) message to the reply channel found in the RREQ message. If the RREQ message was not for its own IP address, the message is re-broadcast after replacing the reply channel with a freshly allocated reply selector and its own MAC address. When such an intermediary node receives a RREP message (Figure 3), it increments the hop counter and forwards the RREP message to the source of the original RREQ message. When the originator of a RREQ message eventually receives the matching RREP (Figure 4), it installs a route and informs the IP layer about it. The message can then be resent (Figure 7, **then** branch) and delivered (Figure 5) by unicast messages through the chain of intermediary forwarding nodes.

We show the basic correctness of the model by the following theorem, which in essence corresponds to the correct operation of an ad-hoc routing protocol [28, Definition 1]: if there is a path between two nodes, the protocol finds it, and it is possible to send packets along the path to the destination node.

The system to analyse consists of n nodes with their respective broadcast handler; node 0 attempts to transmit a packet to the IP address of node n.

$$\mathsf{Spec}_n(pkt, ip_0, \ldots, ip_n) \Leftarrow (\nu mac_0, \ldots, mac_n) \left(\begin{array}{l} \prod_{0 \le i \le n} \mathsf{BrdHandler}(\mathsf{node}_i, mac_i, ip_i) \\ \mid !\, \overline{\mathsf{IP}}\mathsf{transmit}(\mathsf{node}_0, mac_0, ip_n, pkt) \end{array} \right)$$

$$\mathsf{LunARP}(mynode, mymac, destip) \Leftarrow (\nu rchosen, schosen)\left(\begin{array}{l} !\,\underline{\langle rchosen, mymac\rangle}(x)\,.\,\mathsf{SRrepHandler}(mynode, mymac, destip, x) \\ \mid (\!|\mathsf{Redirected}(mynode, schosen)|\!) \\ \mid \overline{mynode}\langle \mathsf{RREQ}(schosen, destip, \langle rchosen, mymac\rangle)\rangle\,.\,\mathbf{0} \end{array}\right)$$

Fig. 1. Part 0: the initialisation step at the node that wishes to discover a route

$$\begin{array}{l} \mathsf{RreqHandler}(mynode, mymac, myip, \mathsf{RREQ}(schosen, destip, repchn)) \Leftarrow \\ \quad \textbf{if } \mathsf{Redirected}(mynode, schosen) \textbf{ then } \mathbf{0} \\ \quad \textbf{else } \tau\,.\,\Big((\!|\mathsf{Redirected}(mynode, schosen)|\!) \mid \\ \qquad \textbf{if } destip = myip \textbf{ then} \qquad\qquad /^*\ \text{Part 2: Target found}\ ^*/ \\ \qquad\quad (\nu rchosen) \\ \qquad\qquad \left(\begin{array}{l} !\,\underline{\langle rchosen, mymac\rangle}(x)\,.\,\mathsf{IPdeliver}(x, mynode) \\ \mid \overline{repchn}\langle \mathsf{RREP}(0, \langle rchosen, mymac\rangle)\rangle\,.\,\mathbf{0} \end{array}\right) \\ \qquad \textbf{else} \\ \qquad\quad (\nu rchosen) \\ \qquad\qquad \left(\begin{array}{l} !\,\underline{\langle rchosen, mymac\rangle}(x)\,.\,\mathsf{IRrepHandler}(mymac, repchn, x) \\ \mid \overline{mynode}\langle \mathsf{RREQ}(schosen, destip, \langle rchosen, mymac\rangle)\rangle\,.\,\mathbf{0} \end{array}\right)\Big) \end{array}$$

Fig. 2. Part 1: RREQ packet handler, and Part 2: Target found branch

$$\mathsf{IRrepHandler}(mymac, repchn, \mathsf{RREP}(hops, fwdptr)) \Leftarrow (\nu rchosen)\left(\begin{array}{l} !\,\underline{\langle rchosen, mymac\rangle}(x)\,.\,\overline{fwdptr}\,x\,.\,\mathbf{0} \\ \mid \overline{repchn}\langle \mathsf{RREP}(hops + 1, \langle rchosen, mymac\rangle)\rangle\,.\,\mathbf{0} \end{array}\right)$$

Fig. 3. Part 3: Intermediate RREP packet handler

$$\mathsf{SRrepHandler}(mynode, mymac, destip, \mathsf{RREP}(hops, fwdptr)) \Leftarrow (\nu rchosen)\left(\begin{array}{l} !\,\underline{\langle rchosen, mymac\rangle}(x)\,.\,\overline{fwdptr}\,x\,.\,\mathbf{0} \\ \mid (\!|\mathsf{HaveRoute}(mynode, destip, hops, rchosen)|\!) \end{array}\right)$$

Fig. 4. Part 4: Source RREP packet handler

$$\mathsf{IPdeliver}(x, node) \Leftarrow \overline{\langle \mathsf{delivered}, node\rangle}\,x\,.\,\mathbf{0}$$

Fig. 5. Part 5: IP delivery

$$\mathsf{BrdHandler}(mynode, mac, ip) \Leftarrow \underline{mynode}(\lambda s, t, r)\mathsf{RREQ}(s, t, r)\,.\,\left(\begin{array}{l} \mathsf{RreqHandler}(mynode, mac, ip, \mathsf{RREQ}(s, t, r)) \\ \mid \mathsf{BrdHandler}(mynode, mac, ip) \end{array}\right)$$

Fig. 6. Broadcast handler

$$\begin{array}{l} \mathsf{IPtransmit}(mynode, mymac, destip, pkt) \Leftarrow \\ \quad \textbf{if } \mathsf{HaveRoute}(mynode, destip) \textbf{ then } \overline{\langle \mathsf{RouteOf}(mynode, destip), mymac\rangle}\,pkt\,.\,\mathbf{0} \\ \quad \textbf{else } \mathsf{LunARP}(mynode, mymac, destip) \end{array}$$

Fig. 7. IP transmission: if have route, send it to local forwarder, else ask for route

Theorem 12. *If* Ψ *connects* node_0 *and* node_n *via a node* node_i *(i.e.* $\Psi \vdash \mathsf{node}_0 \dot{\succ} \mathsf{node}_i$ *and* $\Psi \vdash \mathsf{node}_i \dot{\succ} \mathsf{node}_n$*), then*

$$\Psi \mid (\nu ip_0, \ldots, ip_n)\mathsf{Spec}_n(pkt, ip_0, \ldots, ip_n) \Longrightarrow \xrightarrow{\overline{\langle \mathsf{delivered}, \mathsf{node}_n \rangle pkt}} \Psi \mid (\nu ip_0, \ldots, ip_n)S$$

and $\mathcal{F}(S) \vdash \mathsf{HaveRoute}(\mathsf{node}_0, ip_n)$*, where* $\Longrightarrow$ *stands for an interleaving of* τ *and broadcast output transitions.*

Proof. By following transitions.

Our analysis is limited to a two-hop configuration due to the labour of manually following transitions in a non-trivial specification. We anticipate this can be automated using a future extension of our symbolic semantics for psi-calculi [10,11].

The definition of BrdHandler illustrates a peculiarity of broadcast semantics: a reader well-versed in pi-calculus specifications with replication and recursion may consider a more concise variant of the definition using replication instead of recursion, e.g.

$$\mathsf{BrdHandler}'(mynode, mac, ip) \Leftarrow \; !\,\underline{mynode}(\lambda s, t, r)\mathsf{RREQ}(s, t, r)\,.\,\mathsf{RreqHandler}(mynode, mac, ip, \mathsf{RREQ}(s, t, r))$$

When the input prefix is over a broadcast channel, as is the case here, the two are not equivalent since a single communication with BrdHandler′ may result in arbitrarily many RreqHandler processes, while BrdHandler only results in one.

6 Related Work

Process calculi with broadcast communication go back to the early 1980's. Milner developed SCCS [16] as a generalisation of CCS [15] to include multiway communication, of which broadcast can be seen as a special case. At the same time Austry and Boudol presented MEIJE [2] as a semantic basis for high-level hardware definition languages.

The first process calculus to seriously consider broadcast with an asynchronous parallel composition was CBS [19,20]. Its development is recorded in a series of papers, examining it from many perspectives. The main focus is on employing broadcast as a high level programming paradigm. CBS was later extended to the pi-calculus in the bπ formalism [5]. Here the broadcast communication channels are names that can be scoped and transmitted between agents. The main point of this work is to establish a separation result in expressiveness: in the pi-calculus, broadcast cannot be uniformly encoded by unicast.

Recent advances in wireless networks have created a renewed interest in the broadcast paradigm. The first process calculus with this in mind was probably CBS$^\sharp$ [17]. This is a development of CBS to include varying interconnection topologies. Input and output is performed on a universal ether and transitions are indexed with topologies which are sets of connectivity graphs; the connectivity graph matters for the input rule (reception is possible from any connected

location). Main applications are on cryptography and routing protocols in mobile ad hoc wireless networks. CBS$^\sharp$ has been followed by several similar calculi. In CWS [14,12] the focus is on modelling low level interference. Communication actions have distinct beginnings and endings, and two actions may interfere if one begins before another has ended. The main result is an operational correspondence between a labelled semantics and a reduction semantics. CMAN [8] is a high level formalism extended with data types, just as the applied pi-calculus extends the original pi-calculus. Data can contain constructors and destructors. There are results on properties of weak bisimulation and an analysis of a cryptographic routing protocol. In the ω-calculus [23] emphasis is on expressing connectivity using sets of group names. An extension also includes separate unicast channels, making this formalism the first to accommodate both multicast and unicast. There are results about strong bisimulation and a verification of a mobile ad hoc network leader election protocol through weak bisimulation. RBPT [7] is similar and uses an alternative technique to represent topology changes, leading to smaller state spaces, and is also different in that it can accommodate an asymmetric neighbour relation (to model the fact that A can send to B but not the other way).

$bA\pi$ [9] is an extension of the applied pi-calculus [1] with broadcast, where connectivity information appears explicitly in the process terms and can change non-deterministically during execution. The claimed result of the paper is proving that a weak labelled bisimulation, for which connectivity is irrelevant, coincides with barbed equivalence. However, for the same reasons as in the applied pi-calculus (cf. [3]), labelled bisimilarity is not compositional in $bA\pi$, so the correspondence does not hold. A suggested fix is to remove unicast channel mobility from the calculus. We would finally mention CMN [13]. The claimed result is to compare two different kinds of semantics for a broadcast operation, but it is in error. The labelled transition semantics contains no rule for merging two inputs as in our BrMerge. As a consequence parallel composition fails to be associative. Consider the situation where P does an output and Q and R both do inputs. A broadcast communication involving all three agents can be derived from $(P|Q) \mid R$ but not from $P \mid (Q|R)$, since in the latter agent the component $Q|R$ cannot make an input involving both Q and R.

It is interesting to compare these formalisms and our broadcast psi from a few important perspectives. Firstly, the broadcast channels are explicitly represented in ω, bπ, CWS and CMN; they are mobile (in the sense that they can be transmitted) only in bπ. In ω, only unicast channels can be communicated. In broadcast psi, channels are represented as arbitrary mobile data terms which may contain any number of names. Secondly, the data transmitted in CMAN and $bA\pi$ is akin to the applied pi-calculus where data are drawn from an inductively defined set and contain names which may be scoped. In ω and bπ data are single names which may be scoped; in the other calculi data cannot contain scoped names. In broadcast psi data are arbitrary terms, drawn from a nominal set, and may include higher order objects as well as bound names. Finally, node mobility is represented explicitly as particular semantic rules in CMAN, CMN, $bA\pi$ and

ω, and implicitly in the requirements of bisimulation in CBS$^\sharp$ and RBPT. In this respect broadcast psi calculi are similar to the latter: connectivity is determined by the assertions in the environment, and in a bisimulation these may change after each transition.

All calculi presented here use a kind of labelled transition semantics (LTS). bπ, $bA\pi$, CBS$^\sharp$, CWS and ω use it in conjunction with a structural congruence (SC), the rest (including broadcast psi) do not use a SC. In our experience SC is efficient in that the definitions become more compact and easy to understand, but introduces severe difficulties in making fully rigorous proofs. $bA\pi$, CWS, CMAN and CMN additionally use a reduction semantics using structural congruence (RS) and prove its agreement with the labelled semantics. Table 3 summarises some of the distinguishing features of calculi for wireless networks.

Table 3. Comparison of some process algebras for wireless broadcast

Calculus	Broadcast Channels	Scoped Data	Mobility	Semantics
$bA\pi$	-	term	in semantics	LTS+SC and RS
CBS$^\sharp$	-	-	in bisimulation	LTS+SC
CWS	constant	-	-	LTS+SC and RS
CMAN	-	term	in semantics	LTS and RS
CMN	name	-	in semantics	LTS and RS
ω	groups	name	in semantics	LTS+SC
RBPT	-	-	in bisimulation	LTS
Broadcast psi	term	term	in bisimulation	LTS

Finally, broadcast psi is different from the other calculi for wireless broadcast in that there is no stratification of the syntax into processes and networks. There is just the one kind of agent, suitable for expressing both processes operating in nodes and behaviours of entire networks. In contrast, the other calculi has one set of constructs to express processes and another to express networks, sometimes leading to duplication of effort (for example, there can be a parallel composition operator both at the process and network level). Our conclusion is that broadcast psi is conceptually simpler and more efficient for rigorous proofs, and yet more expressive.

7 Conclusion

We have extended the psi-calculi framework with broadcast communication, and formally proved using Isabelle/Nominal that the standard congruence and structural properties of bisimilarity hold also after the addition. We have shown how node mobility and network topology changes can be modelled using assertions. Since bisimilarity is closed under all assertions, two bisimilar processes are equivalent in all initial topologies and for all node mobility patterns. We demonstrated

expressive power by modelling the LUNAR protocol for route discovery in wireless ad-hoc networks, and verified a basic correctness property of the protocol.

The model of LUNAR is simplified for clarity and to make manual analysis more manageable. The simplifications are similar to those in the SPIN model by Wibling et al. [28], although we do not model timeouts. Their model [27] is ca 250 lines of SPIN code (excluding comments) while ours is approximately 30 lines. Our model could be improved at the cost of added complexity. For example, allowing broadcast channels to have non-empty support would let us hide broadcast actions, routing tables could be made local by including a scoped name per node, and route deletions could be modelled using generational mechanisms similar to Section 4.

We intend to extend the symbolic semantics for psi-calculi [10,11] with broadcast, and implement the semantics in a tool for automatic verification. We also plan to study weak bisimulation for the broadcast semantics. In order to model more aspects of wireless protocols, we would like to add general resource awareness (e.g. energy or time) to psi-calculi.

References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proceedings of POPL 2001, pp. 104–115. ACM, New York (2001)
2. Austry, D., Boudol, G.: Algèbre de processus et synchronisation. Theor. Comput. Sci. 30, 91–131 (1984)
3. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: Mobile processes, nominal data, and logic. In: Proceedings of LICS 2009, pp. 39–48. IEEE, Los Alamitos (2009)
4. Bengtson, J., Johansson, M., Parrow, J., Victor, B.: Psi-calculi: A framework for mobile processes with nominal data and logic. Logical Methods in Computer Science (2011), Accepted for publication. This is an extended version of [3]
5. Ene, C., Muntean, T.: Expressiveness of point-to-point versus broadcast communications. In: Ciobanu, G., Păun, G. (eds.) FCT 1999. LNCS, vol. 1684, pp. 258–268. Springer, Heidelberg (1999)
6. Gabbay, M., Pitts, A.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing 13, 341–363 (2001)
7. Ghassemi, F., Fokkink, W., Movaghar, A.: Restricted broadcast process theory. In: Cerone, A., Gruner, S. (eds.) SEFM, pp. 345–354. IEEE Computer Society, Los Alamitos (2008)
8. Godskesen, J.C.: A calculus for mobile ad hoc networks. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 132–150. Springer, Heidelberg (2007)
9. Godskesen, J.C.: Observables for mobile and wireless broadcasting systems. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 1–15. Springer, Heidelberg (2010)
10. Johansson, M., Victor, B., Parrow, J.: A fully abstract symbolic semantics for psi-calculi. In: Proceedings of SOS 2009. EPTCS, vol. 18, pp. 17–31 (2010)
11. Johansson, M., Victor, B., Parrow, J.: Computing strong and weak bisimulations for psi-calculi (submitted for publication, 2011)

12. Lanese, I., Sangiorgi, D.: An operational semantics for a calculus for wireless systems. Theor. Comp. Sci. 411(19), 1928–1948 (2010)
13. Merro, M.: An observational theory for mobile ad hoc networks (full version). Inf. Comput. 207(2), 194–208 (2009)
14. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electr. Notes Theor. Comput. Sci. 158, 331–353 (2006)
15. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
16. Milner, R.: Calculi for synchrony and asynchrony. Theor. Comput. Sci. 25, 267–310 (1983)
17. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theor. Comp. Sci. 367(1-2), 203–227 (2006)
18. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation 186, 165–193 (2003)
19. Prasad, K.V.S.: A calculus of broadcasting systems. In: Abramsky, S., Maibaum, T.S.E. (eds.) CAAP 1991 and TAPSOFT 1991. LNCS, vol. 493, pp. 338–358. Springer, Heidelberg (1991)
20. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program. 25(2-3), 285–327 (1995)
21. Raabjerg, P., Åman Pohjola, J.: Broadcast psi-calculus formalisation. Isabelle/HOL-Nominal formalisation of the definitions, theorems and proofs (July 2011), `http://www.it.uu.se/research/group/mobility/theorem/broadcastpsi`
22. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
23. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)
24. Tschudin, C., Gold, R., Rensfelt, O., Wibling, O.: LUNAR: a lightweight underlay network ad-hoc routing protocol and implementation. In: Proc. of NEW2AN 2004, St. Petersburg (February 2004)
25. Tschudin, C.F.: Lightweight underlay network ad hoc routing (LUNAR) protocol. Internet Draft, Mobile Ad Hoc Networking Working Group (March 2004)
26. Urban, C., Tasson, C.: Nominal techniques in isabelle/HOL. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 38–53. Springer, Heidelberg (2005)
27. Wibling, O.: SPIN and UPPAAL ad hoc routing protocol models. Models of LUNAR scenarios used in [28] (2004), `http://www.it.uu.se/research/group/mobility/adhoc/gbt/other_examples`
28. Wibling, O., Parrow, J., Pears, A.: Automatized verification of ad hoc routing protocols. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 343–358. Springer, Heidelberg (2004)

A Formalisation of Java Strings for Program Specification and Verification

Richard Bubel[1], Reiner Hähnle[1], and Ulrich Geilmann[2]

[1] Department of Computer Science and Engineering
Chalmers University of Technology, Sweden
{bubel,reiner}@chalmers.se
[2] Department of Computer Science, KIT Karlsruhe, Germany
ugeilmann@googlemail.com

Abstract. We present a formalisation of Java Strings tailored to specification and verification of programs (using dynamic logic). The formalism allows to specify and verify properties about the content of strings—the most common use-case—in an easy and natural manner. Each instance of type String is related to an abstract data type representing the string content as an immutable sequence of characters. This avoids serious technicalities that would arise if the specification had to resort to Java arrays to represent sequences of characters. We also discuss advanced aspects of Java Strings including string literals and the string pool and support for regular expressions. The approach has been implemented in the KeY verification system. We demonstrate its practical applicability by case studies including the verification of a string sanitization function.

1 Introduction

Most Java programs that deal with user input and output make usage of strings. Formal verification of Java-like languages progressed rapidly regarding the size and complexity of verifiable programs in the past years, yet none of the state-of-art systems [1,2,3,4] offers a comparable degree of support and automation for strings as is on hand for other datatypes of central importance. Typically, strings are given less priority in formalisation by the designers of verification tools, because they deal with aspects of programs that are considered to be computationally trivial. Another reason for little work having been done on strings is that many Java verification systems concentrated on Java Card or other Java dialects that have no string support.

Nevertheless, missing or insufficient support for strings makes it impossible to verify many practically relevant programs or requires to rewrite them before verification. One should also keep in mind that many attacks on software security are based on injection of intentionally malformed input[1] which makes specification and verification of methods with string type much more security-relevant and less trivial than thought at first.

[1] http://www.sans.org/top25-programming-errors/

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 90–105, 2011.

In this paper we present a formalisation of Java strings within the program logic of the KeY verification system [3]. This includes a formal specification of a substantial part of the Java String API (Sect. 3.5) as well as automated reasoning on first-order string functions and predicates.

Strings in Java-like languages are non-trivial to model for several reasons: first of all, strings are (immutable) instances of the String class. This means that strings are objects with all the usual complications such as aliasing. It means also that several dozen API methods can be applied to Strings whose behaviour must be specified. In Java and C# immutability of strings is exploited in the String pool which implements caching of equal string literals.

In order to allow efficient automated reasoning we map Java String objects and methods into a first-order abstract data type for finite character sequences (Sect. 3.1). The mechanism used in KeY for symbolic state updates (described in Sect. 2.2) allows us to do this in a modular fashion, see Sect. 3.2. We also model the caching of string literals in the Java String pool (Sect. 3.3) and support for regular expression on character sequences (Sect. 3.4).

We demonstrate the practicability of our approach with three case studies that exhibit a high degree of automation that can be achieved with our formalisation when verifying Java programs with strings. Much of our work is also applicable to C# which has a very similar string concept than Java.

2 Background

2.1 Java Strings

In contrast to many other programming languages, strings in Java are not identified with an array of characters. Instead, a Java String is an object of class `java.lang.String`. Strings are immutable, i.e., once created their content—the encapsulated character sequence—cannot be changed.

While strings behave like ordinary objects, they are part of the core classes in the Java Language Specification (JLS) [5] and enjoy special support in the form of String literals and the String pool. String literals are Java expressions and consist of a possibly empty sequence of characters enclosed in double quotation marks (§3.10.5 [5]): $\mathit{StringLiteral} := "(\mathit{StringCharacter})^*"$

String literals are intended to behave like literals of primitive types. Consequently, they are immutable and refer always to the same instance of type `String`. Additionally, Java provides the following support for strings:

- The '+'-operator is overloaded to be defined on Strings: if one of its arguments is of type String, the second one is (if necessary) translated to its canonical String representation (§15.18.1 [5]). The result of the application of '+' is a (new) String object representing the concatenation of the argument strings.
- The notion of compile-time constants is extended to cover string literals. A compile-time constant (§15.28 [5]) is a Java expression adhering to a defined syntactical form that can be (and is) evaluated at compile-time to the resulting value.

– It is ensured that two occurrences of String literals of the same name (no matter where they are used) refer always to the same String instance; otherwise one would run into situations where `"abc"=="abc"` evaluates to false.

For the third item it was necessary to introduce the concept of a String pool. The String pool is a cache that maps sequences of characters to corresponding String instances. When are String instances entered into the pool? The answer is twofold:

Primarily, each time when a class is loaded all of its compile-time constants of type String (in particular, all String literals) are resolved and replaced by references to the actual String objects.First it is checked whether an instance of the same content had already been registered. If such an object is found it is used to resolve the reference. Otherwise, a new String instance representing the constant is created, added to the pool and used to resolve the reference.

Additionally, the programmer can add any string to the pool by using the instance method `intern()` of class `java.lang.String`. Invoking `s.intern()` checks first whether an instance equal to `s` had already been registered. If the check is negative, then the String instance referred to by `s` is added to the pool and returned as the result of the method invocation. Otherwise, `s` is not added to the pool and the returned instance is the one already registered.

2.2 Dynamic Logic for Sequential Java

As program logic serves us JavaDL [3] a sorted first-order dynamic logic with updates for sequential Java programs. Actually JavaDL defines a family of logics of which each concrete logic JavaDL(`p`) is associated with a Java program `p`. The program `p` is sometimes referred to as context program. We usually omit the index `p` if no ambiguities arise.

The *signature* $\Sigma_{\mathtt{p}} = ((\mathcal{T}, \sqsubseteq), \mathcal{O}, \Pi, \mathcal{F}, \mathcal{P}, \mathcal{V})$ consists of a set of types $\mathcal{T}$ with type hierarchy $\sqsubseteq$, the usual propositional connectives and first-order quantifiers as well as the two dynamic logic modal operators $\langle\cdot\rangle\cdot$ (*diamond*) and $[\cdot]\cdot$ (*box*). Further, it includes the set of executable program statements (i.e. list of Java statements) Π, and sets of function $\mathcal{F}$, predicate $\mathcal{P}$ and variable $\mathcal{V}$ symbols.

Terms and formulas are inductively defined as usual and we give here only the inductive definitions for the modality operators. Let $\alpha \in \Pi$ be an executable list of Java statements and ϕ a JavaDL formula then

– $\langle\alpha\rangle\phi$ (spoken: *diamond alpha phi*) is a JavaDL formula; its intuitive meaning is that program α terminates *and* ϕ holds its final state.
– $[\alpha]\phi$ (spoken: *box alpha phi*) is a JavaDL formula; its intuitive meaning is that *if* program α terminates *then* ϕ holds its final state.

Note, that the above given intuitive meanings exploit that the sequential fragment of the Java programming language is deterministic. There is a strong relationship between Hoare logics and dynamic logics and the Hoare triple $\{\phi\}\alpha\{\psi\}$ is equivalent to the JavaDL formula $\phi \rightarrow [\alpha]\psi$.

Function (predicate) symbols are partitioned into rigid $\mathcal{F}_r(\mathcal{P}_r)$ and non-rigid $\mathcal{F}_{nr}(\mathcal{P}_{nr})$ symbols, where the interpretation of rigid symbols does not depend

on the current state while non-rigid symbols may change their interpretation depending on the state in which they are evaluated.

Terms and formulas in JavaDL are evaluated relative to a *JavaDL Kripke structure* $\mathcal{K} = (\mathcal{D}, I, \mathcal{S}, \rho)$. The domain $\mathcal{D}$ assigns each type T a set of elements D_T (wrt. the type hierarchy). The interpretation I assigns each rigid function (resp. predicate) symbol f its meaning $I(f)$. On the other hand, states $s \in \mathcal{S}$ are used to assign a meaning to non-rigid symbols. The *state transition relation* $\rho : \Pi \times \mathcal{S} \times \mathcal{S}$ captures the Java semantics. We write $\rho(\alpha)(s_1) = \{s_2\}$, if the program $\alpha \in \Pi$ started in state s_1 terminates reaching the final state s_2. As Java is deterministic, the set of final states is either empty or a singleton set.

A term (or formula) is then evaluated with respect to a JavaDL Kripke structure $\mathcal{K}$, a state $s \in \mathcal{S}$ and a variable assignment $\beta : \mathcal{V} \to \mathcal{D}$ using the valuation function $val_{\mathcal{K},\beta}$. Its definition is standard, thus we give only two cases:

$$
\begin{array}{ll}
val_{\mathcal{K},s,\beta}(a(o_1,\ldots,o_n)) = & \\
\quad s(a)(val_{\mathcal{K},s,\beta}(o_1),\ldots,val_{\mathcal{K},s,\beta}(o_n)) & a \in \mathcal{F}_{nr}, o_1 \ldots o_n \in \mathsf{Trm}_\Sigma \\
val_{\mathcal{K},s,\beta}(\langle\alpha\rangle\phi) = tt \text{ iff.} & \\
\quad \text{exists } s' \in \mathcal{S}.\big(\rho(\alpha)(s) = \{s'\} \wedge val_{\mathcal{K},s',\beta}(\phi)\big) & \alpha \in \Pi_\Sigma, \phi \in \mathsf{For}_\Sigma
\end{array}
$$

The notions *satisfiability*, *model* and *validity* are defined as usual.

Local program variables are modelled as non-rigid constant symbols, while attributes and the array access operator are represented by unary resp. binary non-rigid function symbols. In particular, these symbols form an own class of non-rigid function symbols namely *locations* or *location functions*. The defining element is that their value can be explicitly changed by a program. Instead of $a(o)$ where a is a location function representing an attribute, we write $o.a$.

An important concept in JavaDL is that of *updates*. Updates are a concise mean to express state transitions on the syntactical level and can be seen as some kind of explicit substitutions. An *elementary update* $l := r$ is a pair with l denoting an expression where the top level operator is a location function e.g., a local program variable and r a Java DL term. Its intuitive meaning is that of an assignment where the location denoted by l is assigned the value r.

Updates can be combined to parallel updates $l_1 := r_1 \,||\, \ldots \,||\, l_n := r_n$ or conditional updates $\texttt{if}\ \ \phi;\ l := r$. Parallel updates are evaluated simultaneously, i.e., the assignments within a parallel update do not influence each other. Clashes within a parallel update, i.e., cases where the same location is assigned a value twice, are resolved using a last-one-win semantics and the right-most assignment of the conflicting ones is taken. The conditional update is only "executed" if its condition ϕ holds.

Updates u can be applied to formulas $\{u\}\,\phi$ and terms $\{u\}\,t$ which are JavaDL formulas resp. terms again. The formula/term is then evaluated in the state reached from the current state by applying the assignments of update u.

A complete account to updates and their application is given in [6,3]. A small example: The formula $(i \doteq i_0 \wedge j \doteq j_0) \rightarrow \{i := j \,||\, j := i\}\,(i \doteq j_0 \wedge j \doteq i_0)$ is valid. The parallel update expresses a simultaneous swap of the program variables i and j while the formula behind the update states that the swap has taken place.

To prove the validity of Java DL formulas a Gentzen-style sequent calculus is used and implemented in the KeY-tool [3]. A sequent is denoted by $\Gamma \Longrightarrow \Delta$ and its meaning is equivalent to $\bigwedge_{\varphi\in\Gamma}\varphi \rightarrow \bigvee_{\psi\in\Delta}\psi$ (Γ, Δ are sets of formulas).

The calculus is based on the symbolic execution paradigm and realizes a symbolic interpreter which stepwise reduces complex statements into a sequence of simple statements until they can be moved to an update or translated into a classical first-order formula. For instance, the rule

$$\mathsf{ifSplit}\ \frac{\Gamma, b \doteq \texttt{TRUE} \Longrightarrow \langle\pi\ \{p\}\ \omega\rangle\phi, \Delta \qquad \Gamma, \texttt{b} \doteq \texttt{FALSE} \Longrightarrow \langle\pi\ \{q\}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\pi\ \texttt{if}(b)\{p\}\texttt{else}\{q\}\ \omega\rangle\phi, \Delta}$$

matches on any sequent whose succedent contains a formula with the diamond operator as top level operator and where the first active statement is a conditional statement. The schema variable π matches on the inactive program prefix (opening braces, `try`-s etc.) while ω matches the remaining program. Applying the rule causes the proof to split into two branches one where the `if` statement's condition is assumed to be true and one where it is assumed to be false.

The assignment rule for local variables

$$\mathsf{assignment}\ \frac{\Gamma, b \doteq \texttt{TRUE} \Longrightarrow \{v := se\}\ \langle\pi\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\pi\ v{=}se;\ \omega\rangle\phi, \Delta}$$

is another example of a symbolic execution rule. The schema variable se on the right matches only on side-effect free expressions and therefore the assignment can be directly moved into an update. The program is thus successively reduced to first-order formulas and then treated using classical first-order sequent rules.

The last concept we want to introduce is that of Java-reachable states. Basically, the Kripke structure contains states not reachable by a Java program. For instance, each object has an implicit boolean-typed field `<created>`, which is set to `TRUE` when the object in consideration is created. The Kripke structure contains now states $s \in \mathcal{S}$ where some created objects have a reference to a not yet created one. Obviously, these states can never be reached by a Java program. The non-rigid predicate `iRS` (*in Java-Reachable State*) is defined (and axiomatized) to hold in exactly those states that can be established by a Java program. The formula $\texttt{iRS} \rightarrow \phi$ expresses now that if we are in a state reachable by a Java program then the formula ϕ holds.

3 Specification of Java Strings

3.1 The Abstract Datatype `CharList`

We use the abstract data type (ADT) `CharList` (introduced below) to represent the content of a String object as an immutable sequence of characters. In Sect. 3.2 we will establish a formal connection between character sequences and String objects. Using an abstract data type allows us to specify properties involving (the content of) Strings conveniently. In addition the ADT has a considerably simpler semantics than Java collections or arrays. This helps to avoid

constructors
empty | cons (jchar , CharList)

core auxiliary functions
concat (CharList , CharList): CharList
substring (int , int , CharList): CharList
replace (jchar , jchar , CharList): CharList

query functions
length (CharList): int
charAt (int , CharList): jchar
indexOf(jchar , int , CharList): int

Fig. 1. Excerpt of the `CharList` ADT

cluttering specifications and proofs with unnecessary technicalities. Most function declarations (except some query functions) of the `CharList` ADT are shown in Fig. 1. The usual list constructors `empty` and `cons` are used to create character sequences such as `cons('H',cons('i',empty))`. For sake of readability, we employ the short notation "`Hi`"$_{CL}$. The subscript CL distinguishes elements of `CharList` from Java String literals. The character indices start with 0 and counting is from left-to-right, i.e., `H` has index 0 and `i` has index 1. The auxiliary functions of the datatype are also standard:

- `concat`(s_1,s_2) concatenates two character sequences;
- `substring`(i,j,s) projects onto a subsequence of s starting with (and including) the character at position i and ending at (and excluding) the character at position j;
- `replace`(c_1,c_2,s) evaluates to a character sequence equal to s except that all occurrences of character c_1 have been replaced by the character c_2.

There exist several query functions such as `length`, `charAt`, `indexOf` (in different variants), as well as `contains`, `startsWith`, and `endsWith` that all have their obvious meaning.

`CharList` is an initial ADT and a general induction schema is provided. All auxiliary and query functions are axiomatized in first-order logic. The axioms are given as conditional rewrite rules of the following form:

$$\mathsf{ruleName}\quad \frac{\phi}{t \rightsquigarrow t'}$$

The semantics of such a rule is that t can be rewritten to t' whenever the formula ϕ can be shown to be valid. By way of example, we list the axioms for function `substring` and explain a number of design decisions we made throughout the axiomatization:[2]

$$\mathsf{subBase}\quad \frac{}{\texttt{substring}(0,0,s) \rightsquigarrow \texttt{empty}}$$

[2] The complete axiomatization with detailed explanations is in [7].

subStepOnSnd
$$\frac{j > 0}{\texttt{substring}(0, j, \texttt{cons}(c, s)) \rightsquigarrow \texttt{concat}(\texttt{cons}(c, \texttt{empty}), \texttt{substring}(0, j-1, s))}$$

subStep
$$\frac{i > 0 \wedge i \leq j}{\texttt{substring}(i, j, \texttt{cons}(c, s)) \rightsquigarrow \texttt{substring}(i-1, j-1, s)}$$

We emphasize that the axioms are intentionally incomplete, for instance, the case $\texttt{substring}(-1, 2, \texttt{"Hello"}_{CL})$ is not covered. We use underspecification [8] instead of introducing a separate error term or overspecifying unintended cases. Semantically, this means that an underspecified term has *some* value of type `CharList`, but we do not know which one. Similar cases occur for other functions and predicates such as `charAt` when accessing an out of bounds index.

To reason efficiently about character sequences it is necessary to define a normal form which is obtained by applying the rewrite rules in Fig. 2 exhaustively:

Lemma 1. *Let t denote a term of type `CharList` that (i) consists only of* core auxiliary function *applications (as listed in Fig. 1), free variables, and constant symbols and that is (ii) well-defined, i.e., no subexpression is underspecified. Then the reduction system given in Fig. 2 terminates and establishes a unique normal form t_{NF} with the following properties:*

- *the second argument of a* `concat` *function contains no further* `concat` *function application;*
- *all nested function applications occur exclusively in the order* `substring`*/*`concat`*/*`replace`*;*
- *there is at most one occurrence of a* `substring` *function application.*

Fig. 3 shows some further reduction rules used to eliminate auxiliary functions occurring within a query function.

3.2 Connecting String and `CharList`

In the previous section an abstract representation of Java Strings as sequences of characters has been defined. To enable usage of this abstraction within the program logic for specifying and verifying properties about Java Strings, we need to relate Strings and `CharList` elements. This purpose is achieved by the location function $\texttt{content} : \texttt{java.lang.String} \rightarrow \texttt{CharList}$ that maps each String instance to a sequence of characters. Location functions are functions whose interpretation is state-dependent and whose values can be updated. The domain of the location function is a Java reference type, so we write $s.\texttt{content}$ instead of $\texttt{content}(s)$. As Java Strings are immutable the `content` function is only updated when a new instance is created, for example, when a String concatenation is symbolically executed on the right side of an assignment:

assignAddString
$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}\,\{v := newS\}\quad \{v.\texttt{content} := \texttt{concat}(s1.\texttt{content}, s2.\texttt{content})\}\,\langle \pi\omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ v = s1 + s2;\ \omega\rangle\phi, \Delta}$$

concatLeftAssoc $$\frac{}{\texttt{concat}(s_1, \texttt{concat}(s_2, s_3)) \rightsquigarrow \texttt{concat}(\texttt{concat}(s_1, s_2), s_3)}$$

repSubRed $$\frac{i \geq 0 \wedge j \geq i \wedge j \leq \texttt{length}(s)}{\texttt{replace}(c1, c2, \texttt{substring}(i, j, s)) \rightsquigarrow \texttt{substring}(i, j, \texttt{replace}(c1, c2, s))}$$

repConcatRed

$$\frac{}{\texttt{replace}(c1, c2, \texttt{concat}(s_1, s_2)) \rightsquigarrow \texttt{concat}(\texttt{replace}(c1, c2, s_1), \texttt{replace}(c1, c2, s_2))}$$

subSubRed $$\frac{k \geq 0 \wedge m \geq k \wedge m \leq \texttt{length}(s) \wedge i \geq 0 \wedge j \geq i \wedge j \leq m - k}{\texttt{substring}(i, j, \texttt{substring}(k, m, s)) \rightsquigarrow \texttt{substring}(i + k, k + j, s)}$$

subConcatRed

$$\frac{R \wedge j \leq \texttt{length}(s_1)}{\texttt{substring}(i, j, \texttt{concat}(s_1, s_2)) \rightsquigarrow \texttt{substring}(i, j, s_1)}$$

$$\frac{R \wedge j > \texttt{length}(s_1) \wedge i \geq \texttt{length}(s_1)}{\texttt{substring}(i, j, \texttt{concat}(s_1, s_2)) \rightsquigarrow \texttt{substring}(i - \texttt{length}(s_1), j - \texttt{length}(s_1), s_1)}$$

$$\frac{R \wedge j > \texttt{length}(s_1) \wedge i < \texttt{length}(s_1)}{\begin{array}{l}\texttt{substring}(i, j, \texttt{concat}(s_1, s_2)) \rightsquigarrow \\ \quad \texttt{concat}(\texttt{substring}(i, \texttt{length}(s_1), s_1), \texttt{substring}(i, j - \texttt{length}(s_1), s_2))\end{array}}$$

with $R := i \geq 0 \wedge j \geq i \wedge j \leq \texttt{length}(s_1) + \texttt{length}(s_2)$

Fig. 2. Reduction system for core auxiliary functions of `CharList`

lengthSubRed $$\frac{i \geq 0 \wedge i \leq j \wedge j \leq \texttt{length}(s)}{\texttt{length}(\texttt{substring}(i, j, s)) \rightsquigarrow j - i}$$

charAtSubRed $$\frac{i < k - j \wedge i \geq 0 \wedge j \geq 0 \wedge k \geq j \wedge k \leq \texttt{length}(s)}{\texttt{charAt}(k, \texttt{substring}(i, j, s)) \rightsquigarrow \texttt{charAt}(i + k, s)}$$

Fig. 3. Further reduction rules (excerpt)

In this rule the schema variables v, $s1$, and $s2$ match local program variables of type String. The value update $\mathcal{U}$ and the String-typed term $newS$ cover the technicalities of object creation (this is not relevant for the paper and not further explained—details are in [3]). The only rule where the field `content` is assigned a value are those rules dealing with instance creation of Strings reflecting the immutability of class String.

3.3 String Literals and the String Pool

The Java String pool caches String instances using their content as key. On startup of the virtual machine and after class loading all compile-time constants of type String (in particular all String literals) are resolved to an actual String object as described in Sect. 2.1. New elements can be added to the cache at run-time with method `intern()`, but Java programs cannot remove elements from the cache.

We model the String pool as updatable location function $\texttt{pool} : \texttt{CharList} \to \texttt{java.lang.String}$. The `pool` function is defined by a number of properties. These properties are not valid in all states, but only in those program states actually realizable in Java, thus they are qualified with a predicate `iRS` (for "in Java-reachable state"). Other rules do not need the qualification as they are do not rely on the states and we are free to choose a semanntics for not Java reachable states as long as they are not contradictory.

1. $\texttt{iRS} \to (\texttt{pool}(c) \doteq \texttt{null} \lor \texttt{pool}(c).\texttt{created} \doteq \texttt{TRUE})$ stipulates that each element of the co-domain of is either created or `null`.
2. $(\texttt{iRS} \land \neg(\texttt{pool}(c) \doteq \texttt{null})) \to \texttt{pool}(c).\texttt{content} \doteq c$ ensures that when the String pool maps a character sequence c to an object then the content of that object is equal to c.
3. Any compile-time constant of type String with content $cLit$ is not `null`: $\texttt{iRS} \to \neg(\texttt{pool}(cLit) \doteq \texttt{null})$.

The assignment rule for String literals in the presence of the String pool can now be defined as follows:

assignStringLiteral
$$\frac{\Gamma \Longrightarrow \{v := \texttt{pool}(sLit_{CL}) \,||\, \texttt{pool}(sLit_{CL}).\texttt{content} := sLit_{CL}\}\, \langle \pi\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ v = sLit;\ \omega \rangle \phi, \Delta}$$

Here $sLit$ is a schema variable matching String literals and $sLit_{CL}$ denotes the `CharList` representation of the matched String literal $sLit$. Note that it is unnecessary for completeness to have $\texttt{pool}(sLit_{CL}).\texttt{content} := sLit_{CL}$ as part of the update in the rule premise and that it does not interfere with immutability of Strings, because for Java-reachable states this follows from the axioms of `pool` and we could leave it underspecified for all other states. Adding the update makes, however, first-order reasoning more automatic. In addition, when running the symbolic execution engine on code snippets, it is not necessary to specify that one is in a Java-reachable state.

The rule for concatenation of two string literals is similar (cf. the rule for adding two local variables of type String in Sect. 3.2):

concatenateStringLiteral
$$\frac{\begin{array}{l}\Gamma, c \doteq \texttt{concat}(sLit1_{CL}, sLit2_{CL}) \Longrightarrow \\ \quad \{v := \texttt{pool}(c) \| \texttt{pool}(c).\texttt{content} := c\} \langle \pi\ \omega \rangle \phi, \Delta\end{array}}{\Gamma \Longrightarrow \langle \pi\ v = sLit1 + sLit2;\ \omega \rangle \phi, \Delta}$$

Here c is a new constant of type `CharList`. We can query the pool directly for the resulting concatenation, because for Java-reachable states the third `pool` axiom

guarantees that it is defined. Finally, we give one of the rules for updating the Java String pool with a new element:[3]

$$\textsf{updatePool}\quad \frac{\Gamma, \neg(v \doteq \texttt{null}) \Longrightarrow \quad \{\texttt{if}\,(\texttt{pool}(v.\texttt{content}) \doteq \texttt{null});\ \texttt{pool}(v.\texttt{content}) := v\}\ \langle \pi\ \omega\rangle\phi, \Delta}{\Gamma, \neg(v \doteq \texttt{null}) \Longrightarrow \langle \pi\ v.\texttt{intern}();\ \omega\rangle\phi, \Delta}$$

The conditional update adds v only when the String pool does not already contain an object with (more precisely: mapped to) the same content.

3.4 Regular Expressions for `CharList`

We added also support to match elements of type `CharList` with regular expression or pattern expressions.[4] Pattern expressions (PExp) are represented as terms of type `RegEx`. Table 1 lists the PExp constructors. For instance, the pattern represented by the term $\texttt{repeatStar}(\texttt{regEx}(\texttt{"ab"}_{CL}))$ matches a finite but arbitrarily often repetition of the word "ab". Match expressions are constructed using the predicate `match(RegEx, CharList)`. The predicate `match` takes two arguments: a PExp as first argument and the concrete character sequence to be matched against the pattern as second argument. The match expression is true if and only if the provided pattern matches the complete `CharList`.

Table 1. Pattern expressions (PExp) with cl: `CharList` and $pe, pe1, pe2$: `RegEx`

constructor (of type `RegEx`)		constructor	
`regEx(`cl`)`	matches exactly cl	`repeatStar(`pe`)`	pe^*
`opt(`pe`)`	$pe?$	`repeatPlus(`pe`)`	pe^+
`alt(`$pe1$, $pe2$`)`	$pe1 + pe2$	`repeat(`pe, n`)`	pe^n
`regConcat(`$pe1, pe2$`)`	$pe1 \cdot pe2$		

Our calculus features a complete axiomatisation of the pattern and matching language. Further, there is a number of derived rules to reduce and simplify pattern and match expression terms as far as possible. We give here only a few typical representatives of these axioms and rules:

$$\textsf{altAxiom}\quad \frac{}{\texttt{match}(\texttt{alt}(pe1, pe2), cl) \rightsquigarrow \texttt{match}(pe1, cl) \vee \texttt{match}(pe2, cl)}$$

$$\textsf{regConcatAxiom}\quad \frac{}{\begin{array}{l}\texttt{match}(\texttt{regConcat}(pe1, pe2), cl) \rightsquigarrow \\ \exists\ \texttt{int}\ i; (i \geq 0 \wedge i \leq \texttt{length}(cl)\ \wedge\ \texttt{match}(pe1, \texttt{substring}(0, i, cl)) \wedge \\ \qquad\qquad \texttt{match}(pe2, \texttt{substring}(i, \texttt{length}(cl), cl)))\end{array}}$$

[3] This rule has been simplified with respect to method resolution to make it more readable.

[4] Remark: Our encoding allows to express patterns beyond regular expressions.

The first axiom maps the alternative pattern constructor back to a logical disjunction. The second axiom removes the pattern concatenation by guessing the index where to split the text to be matched into two parts. Each part is then independently matched against the corresponding subpattern.

A typical reduction rule which reduces the pattern expression complexity is rule regConcatConcreteStringLeft:

$$\frac{}{\texttt{match}(\texttt{regConcat}(\texttt{regEx}(s), pe), cl) \rightsquigarrow \texttt{length}(s) \leq \texttt{length}(cl) \wedge \texttt{match}(\texttt{regEx}(s), \texttt{substring}(0, \texttt{length}(s), cl)) \wedge \texttt{match}(pe, \texttt{substring}(\texttt{length}(s), \texttt{length}(cl), cl))}$$

3.5 Specification of the Java String API

To obtain a complete calculus for Java strings, additional rules have to be created which translate an integer or the `null` reference to its String representation. The formalisation of the necessary translate functions is rather tedious, but otherwise straightforward. The technical details are described in [7].

Based on the formalisation described in this section, we specified the majority of the methods declared and implemented in the `java.lang.String` class. The `CharList` ADT functions have been chosen to represent closely the core functionality provided by the String class. The specification of the methods required then merely to consider the border cases of most of the methods. Border cases are typically those cases where the ADT has been left underspecified and that cause an exception in Java.

4 Case Studies

We present three small, but non-trivial case studies to illustrate the practical applicability of the presented approach.

4.1 String Distance Measure

The static method `distance(String,String)` shown in Fig. 4 computes the distance between two strings. The applied distance measure is based on the content of the compared String instances.

For this case-study we are interested in the verification of two properties: (i) the distance measure is commutative and (ii) `distance(String,String)` returns 0 if and only if both strings have the same content, i.e. the `equals`-method of class String returns `true`. The first property can be expressed in dynamic logic as

$$\texttt{iRS} \rightarrow \forall \texttt{java.lang.String}\ s, t; \big(s.\texttt{created} \doteq \texttt{TRUE} \wedge t.\texttt{created} \doteq \texttt{TRUE} \rightarrow \texttt{distance}(s,t) \doteq \texttt{distance}(t,s)\big)$$

To prove the validity of the formula, the two queries on the right side of the implication are first replaced by their definition, i.e., the implementation given

```
public static int distance (String s1, String s2) {
    if (s1 == null || s2 == null) return -1;
    int d = 0; int m = s1.length();
    if (s1.length() > s2.length()) m = s2.length();
    for (int i = 0; i < m; ++i) {
        int f = s1.charAt(i) - s2.charAt(i);
        if (f >= 0) { d += f; } else { d += -f; }
    }
    m = s1.length() - s2.length();
    if (m < 0) { d -= m; } else { d += m; }
    return d;
}
```

Fig. 4. The `distance` method

in Fig. 4. The actual verification work is then to provide a suitable loop invariant for the **`for`** loop. The proof is closed after 11423 nodes on 200 branches and required 21 interactive rule applications.

To verify that the computed distance is 0 if and only if both Strings have the same content, the following dynamic logic formula has to be proven:

$$\texttt{iRS} \rightarrow \forall \texttt{java.lang.String}\ s,t;\ \big((s.\texttt{created} \doteq \texttt{TRUE} \wedge t.\texttt{created} \doteq \texttt{TRUE} \wedge s \neq \texttt{null} \wedge t \neq \texttt{null}) \rightarrow (\texttt{distance}(s,t) \doteq 0 \leftrightarrow s.\texttt{equals}(t) \doteq \texttt{TRUE})\big)$$

The formula can be proven valid and the closed proof consists of 4642 nodes on 102 branches requiring 23 steps of user interaction. The reason for the considerable smaller proof tree compared to the previous one, is that we only need to symbolically execute the `distance()` method once compared to the two times when verifying the commutativity.

4.2 Hash Set

For the second case study we implemented, specified and verified a hashset for Strings called `StringSet` shown in Fig. 5. The hashset allows constant time insertion (`insert(String)`) and lookup (`contains(String)`). The class `StringSet` stores all elements inside the array `elements`. To keep things simple, no collision treatment is performed and the capacity of the set is fixed. Thus insertion of a String may fail, which is indicated by `insert` returning `false`. The array index where a String instance `s` is kept or looked up is its hash value modulo the maximal size of the hashset, namely `s.hashCode() % size`.

The class `StringSet` has been specified completely using the Java Modeling Language (JML) [9]. The specification consists of several contracts for both methods and an instance invariant. The KeY tool provides a JML front-end which allows to translate JML specifications into dynamic logic and then to select from a number of proof-obligations those to be proven. We proved that both methods establish their postconditions and that they respect their assignable

```
public class StringSet {
  private /*@ spec_public nullable @*/ String[] elements;
  private /*@ spec_public @*/ int size;
  /*@ public instance invariant size > 0 && elements != null &&
       elements.length==size && \typeof(elements)==\type(String[]); @*/
  /*@ public normal_behavior
       requires s != null  && elements[(s.hashCode() % size)] == null;
       assignable elements[(s.hashCode() % size)];
       ensures elements[(s.hashCode() % size)] == s && \result == true;
       also ... @*/
  public boolean insert (String /*@ nullable @*/ s) { ... }
  ... // spec. and implementation of contains(String)
}
```

Fig. 5. JML specification of method `insert` of class `StringSet`

clauses. All properties could be proven fully automatically. The smallest proof consisted of 65 nodes on a single brach, while the largest one had 8810 nodes on 99 branches. Nodes and branches together measure the proof complexity. The number of nodes is (basically) equal to the number of applied rules. The proof branches are an additional indication on the number of case distinction considered. Branching is an expensive operation and a high degree usually refers to a complex proof situation.

4.3 String Sanitization

For privacy reasons it is no longer allowed to publish exam results as a list pinned at the blackboard in front of the teachers office. Instead a webform is provided which allows students to query their result by entering their exam number. On the server side the software constructs an SQL query as follows:

```
userEnteredId = textField.getText();
query = "SELECT␣*␣FROM␣examDB␣WHERE␣examNr␣=␣" + userEnteredID;
```

The problem with the above solution is that an interested person can gain access to the complete list of exam results by entering '*' as exam identification number. A more malicious minded person might even enter the SQL command for dropping tables and delete the complete database.

This fictional scenario is typical for injection attacks. A way out is to sanitize (e.g., delete) not allowed characters from user provided input. Applying string sanitization successfully requires to ensure that (i) each untrusted string is sanitized before being processed any further, and (ii), that the sanitization function actually removes all not allowed characters.

The security community focuses mostly on the first issue developing different techniques for taint analyses [10,11]. We were interested in the second issue ensuring that a given sanitization function is implemented correctly.

```
/*@ public normal_behavior
  @ ensures (\forall int i;i>=0 && i<\result.length();
  @  (\exists int j;j>=0 && j<whitelist.length;
  @                      whitelist[j]==\result.charAt(i)));
  @*/
 public String whitelist(String input, char[] whitelist) {
   String stripped = new String();
   int len = input.length();
   /*@ loop_invariant stripped!=null && stripped.length()<=i &&
     @  i>=0 && i<=len && (\forall int k; k>=0 && k<stripped.length();
     @      (\exists int j; j>=0 && j<whitelist.length;
     @                      whitelist[j]==stripped.charAt(k)));
     @ assignable \nothing;
     @ decreasing len - i; @*/
   for (int i = 0; i < len; i++) {
     char c = input.charAt(i);
     if (contains(whitelist, c)) { stripped = stripped + c; }
   }
   return stripped;
 }
```

Fig. 6. Implementation and specification of the `whitelist` sanitization method of class `BaseRuleValidation`. The method has been slightly altered to avoid unboxing of characters and the usage of class `StringBuilder` both are not yet supported by KeY.

The OWASP-ESAPI security framework[5] provides several sanitization methods. We specified and verified the method `whitelist` of class `BasicValidationRule` which is used by the provided sanitization functions. Fig. 6 shows the implementation and JML method specification. The verification was not completely automatic due to the universal quantification used in the loop invariant which our prover could not resolve automatically. We were able to establish the correctness of the sanitization function in 7677 steps with 191 interactive steps.

5 Related Work

In [12] the authors present a solver for string constraints. They represent strings as bitvectors and utilise a bitvector logic to solve the expressed constraints. String constraints are expressed in an input language which allows to specify these constraints in form of a context-free grammar or a regular expression. Their approach is related to our `CharList` theory, but not directly related to Java (or C# strings). It would be interesting to add their solver as additional back-end to KeY to speed-up the verification process; similar to how KeY can already use external SMT provers.

In [13] an Isabelle formalisation of a string library for C0 is presented. The formalisation uses the internal representation of strings in C0 and focuses on

[5] http://code.google.com/p/owasp-esapi-java

the specification and verification of the String library itself. The thesis does not elaborate on the achieved automation. Due to the different type of programming languages the addressed problems are not directly comparable.

Closest related to our approach is the KIV [2] system which uses also a dynamic logic to verify Java programs, but differs fundamentally in its formalisation of the Java programming language. There is some support for String literals. Literals, where used as reference, are replaced by a reference term taking care to reuse the same reference for same literals. Otherwise String literals remain literals specified in terms of an algebraic data type. A"hard coded" output stream is provided and allows to treat example programs like `System.out.println("Hello")`. The intention is to specify and verify e.g. the order of static (class) initialization by injecting `print` statements at various places in the source code. The class `java.lang.String` is not part of the KIV core classes and thus not supported. Adding support would require to add a specification based on an actual implementation of the String class using 16-bit character arrays in contrast to using the string literal datatype. To the best of our understanding the String pool is not (or only limited) modelled explicitly and thus support for the `intern()` method would require additional effort.

The string model of the programming language Spec# [1] is similar to Java. On the logical side, the string formalisation of Spec# specifies no knowledge about the content of string literals and almost no knowledge about the string contents of "regular" string instances. The specification of the `equals` method expresses meta-properties like reflexivity, commutativity and transitivity, but not content equality. But Spec# supports the length property of strings and the verifier is aware of the length of String literals. Special attention was given to an efficient specification of the immutability of Strings [14]. Spec# provides limited support for the String pool to allow treatment of the `switch` construct. The specification of the String pool expresses that the pool is a function on the content of the string and not on the references, but no other properties are specified. The function is also specified as heap independent which is only correct under the assumption that the String pool does not change. The languages Boogie [15], Dafny [16] and Chalice [17] do not feature a string type, but Chalice and Dafny provide support for sequences which are value types.

6 Conclusion and Future Work

We presented a sound and complete content-aware specification of the Java String model including a modifiable String pool. The chosen formalisation allows to express conveniently properties involving the content of strings without the need to resort to a low level encoding using character arrays. We evaluated its applicability on three case-studies one of which a security relevant implementation of a string sanitization function.

As immediate next milestone we intend to specify Java's regular expression package. This will enable us to verify more complex validation rules of the OWASP-ESAPI framework. In this context it is tempting to provide a taint analysis—building on ideas of dynamic taint analyses—is tempting future work.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: an Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg (2005)
3. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
5. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, 3rd edn. The Java Series. Addison-Wesley, Boston (2005)
6. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
7. Geilmann, U.: Formal verification using Java's String class. Studienarbeit, Chalmers University of Technology and Universität Karlsruhe (November 2009), http://www.key-project.org/doc/2009/sta_geilmann.pdf
8. Gries, D., Schneider, F.B.: Avoiding the undefined by underspecification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 366–373. Springer, Heidelberg (1995)
9. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. Draft rev. 1.200 (February 2007)
10. Conti, J.J., Russo, A.: A taint mode for python via a library. In: OWASP AppSec Research 2010 (2010)
11. Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: Proc. of CCS 2008, pp. 39–50. ACM, New York (2008)
12. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for string constraints. In: Proc. of ISSTA 2009. ACM, New York (2009)
13. Starostin, A.: Formal verification of a c-library for strings. Master's thesis, Saarland University (2006)
14. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible immutability with frozen objects. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 192–208. Springer, Heidelberg (2008)
15. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
16. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
17. Leino, K.R., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)

dCTL: A Branching Time Temporal Logic for Fault-Tolerant System Verification

Pablo F. Castro[1,3], Cecilia Kilmurray[1],
Araceli Acosta[2,3], and Nazareno Aguirre[1,3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina
{pcastro,ckilmurray,naguirre}@dc.exa.unrc.edu.ar
[2] Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Córdoba, Argentina
aacosta@famaf.unc.edu.ar
[3] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract. With the increasing demand for highly dependable and constantly available systems, being able to reason about *faults* and their impact on systems is gaining considerable attention. In this paper, we are concerned with the provision of a logic especially tailored for describing fault tolerance properties, and supporting automated verification. This logic, which we refer to as dCTL, employs *temporal deontic operators* in order to distinguish "good" (normal) from "bad" (faulty) behaviors, using deontic permission, prohibition and obligation combined in a novel way with temporal operators. These formulas are interpreted over transition systems, in which normal executions are distinguished from faulty ones. Furthermore, we show that this logic is sufficiently expressive to describe various common properties of interest in fault tolerant systems, and show that it features some desirable characteristics that make it suitable for analysis. Indeed, even though we show that the logic is more expressive than CTL, we prove that it maintains the time complexity of the model checking problem for CTL. The logic, its expressiveness and its use to express properties of fault tolerant systems, are illustrated via some case studies.

Keywords: Formal Methods, Fault Tolerance, Temporal Logic, Model Checking.

1 Introduction

With the increasing demand for highly dependable and constantly available systems, being able to reason about computer systems behavior in order to provide strong guarantees for software correctness, has gained considerable attention, especially for *safety critical systems*. In this context, a problem that deserves attention is that of capturing *faults*, understood as unexpected events that affect a system, as well as expressing and reasoning about the properties of systems in the presence of these faults.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 106–121, 2011.

Various researchers have been concerned with formally expressing fault tolerant behavior, and some formalisms and tools associated with this problem have been proposed [6]. Some recent approaches include the use of model checking for analyzing fault tolerant systems [11], and the employment of synthesis mechanisms for systematically producing controllers that help to achieve fault tolerance [8]. A particular trend in formal methods for fault tolerance, that we take as a starting point in this paper, is based on the observation that normal vs. abnormal behaviors can be treated as behaviors "obeying" and "violating" the rules of correct system conduct, respectively. This leads to a straightforward application of *deontic* operators (operators to express permission, obligation and prohibition) for separating normal from abnormal behaviors, and thus for expressing fault tolerant systems and their properties [7]. This idea has been exploited by various researchers in different ways, e.g., for extending a Hoare logic with the possibility of expressing properties of programs in the presence of exceptions [7], for specifying normal behavior of components in distributed systems [13], for specifying fault tolerant systems and their intended properties [3], and the extension of temporal logics with obligation so that robustness can be expressed [10], amongst others.

The work we present in this paper is related to the above mentioned deontic logic approaches to fault tolerance specification and reasoning. We propose a logic especially tailored for describing fault tolerance properties based on the use of deontic operators, with an emphasis on expressing intended (temporal) properties of fault tolerant systems, rather than (axiomatically) prescribing component/system behavior. We then share the motivation of related works such as [13,3], but components will be described using behavioral models such as transition systems, and the logic will be reserved for expressing properties regarding these systems. We maintain a strong concern on automated verification of these properties. Indeed, this logic, which we refer to as dCTL, is composed of CTL and deontic operators for distinguishing "good" (normal) from "bad" (faulty) behaviors, as other deontic approaches, but the way in which temporal and deontic operators are combined makes the logic suitable for analysis. Our proposed dCTL logic is more expressive than CTL, which as we will argue makes it useful for describing common properties of interest in the context of fault tolerant systems, but it preserves the complexity of the model checking problem for CTL, as we show in this paper. Thus, it constitutes a good candidate for describing temporal properties of fault tolerant systems, when the intention is to use model checking for their analysis. This is so especially compared to related temporal-deontic approaches such as RoCTL* [10,14], for which model checking is currently reduced to CTL* model checking, and thus is significantly less efficient.

We provide a number of case studies which enable us to illustrate the use of the logic, and its expressive power. These case studies, though small, represent simple models of common situations in fault tolerance, and are useful for assessing the expressiveness of the logic. They are presented simply as transition systems in which normal states (those resulting from a normal transition), are distinguished from abnormal ones (those resulting from a fault).

2 Preliminaries

In this section, we reproduce some basic definitions and facts regarding Kripke structures and CTL, which are necessary in the presentation of our logic.

2.1 Kripke Structures

Kripke structures are a standard vehicle for interpreting modal or temporal logic formulas as well as for characterizing the operational behavior of reactive systems [6]. Let AP be a set of atomic propositions. A Kripke structure over AP is a 4-tuple $\langle S, I, R, L\rangle$, where S is a set of elements called *states*, $I \subseteq S$ is a set of *initial states*, $R \subseteq S \times S$ is a *transition* relation between states, and $L : S \to 2^{AP}$ is an interpretation function, which indicates the set of atomic propositions that hold in each state.

Given a Kripke structure $M = \langle S, I, R, L\rangle$, the interpretation of logical connectives and modal operators in a modal logic can typically be defined by resorting to L and the structure of R. For temporal logics, it is usually necessary to employ the notion of *trace* to define the semantics of some operators. A *trace* is simply a maximal sequence of states, adjacent with respect to R. When a trace starts in an initial state, it is called an *execution* of M, with *partial* executions corresponding to non-maximal sequence of adjacent states. Given a trace $\sigma = s_0, s_1, s_2, s_3, \ldots$, the ith state of σ is denoted by $\sigma[i]$, and the final segment of σ starting in position i is denoted by $\sigma[i..]$. Finally, we will denote by $\mathcal{U}_M$ the set of all traces, i.e., maximal sequences of adjacent states, of M.

Without loss of generality, it can be assumed that every state has a successor, as is customary in various temporal logics [2].

Colored Kripke Structures. We define a *colored Kripke structure* as a 5-tuple $\langle S, I, R, L, \mathcal{N}\rangle$, where $\langle S, I, R, L\rangle$ is a Kripke structure and $\mathcal{N} \subseteq S$ is a set of *normal* states. Arcs leading to abnormal states can be thought of as faulty transitions, our representation of *faults* (similar approaches to formally model faults can be found in the literature, e.g., [12]). Then, normal executions will be those transiting only through normal states. The set of normal executions will be denoted by $\mathcal{NT}$. In this paper, we assume that in every colored Kripke structure, and for every normal state, there exists at least one successor state that is also normal, and that at least one initial state is normal. This guarantees that every system has at least one normal execution, i.e., that $\mathcal{NT} \neq \emptyset$.

2.2 Computation Tree Logic

Computation Tree Logic (CTL) is a branching time temporal logic with important applications in model checking [5]. This logic allows for the description of properties over Kripke structures, by complementing propositional connectives with path quantifiers and temporal operators, combined in a certain restricted way. It is a logic of "computation trees" since it allows one to express properties

referring to the tree that is constructed from a Kripke structure, starting from an initial state, and unfolding the structure to form a (typically infinite) tree.

Let us describe the syntax of the logic. Let AP be a set $\{p_0, p_1, \ldots\}$ of atomic propositions; the set Φ of CTL well formed formulas is recursively defined as:

$$\Phi ::= \top \mid p_i \mid \neg\Phi \mid \Phi \rightarrow \Phi \mid \mathsf{EX}\Phi \mid \mathsf{AX}\Phi \mid \mathsf{E}(\Phi\, \mathcal{U}\, \Phi) \mid \mathsf{A}(\Phi\, \mathcal{U}\, \Phi)$$

CTL formulas are interpreted on states over a Kripke structure. Given a Kripke structure $M = \langle S, I, R, L\rangle$, and a state $s \in S$, the semantics of CTL formulas is defined as follows:

- $M, s \models \top$
- $M, s \models p_i \Leftrightarrow p_i \in L(s)$, where $p_i \in AP$.
- $M, s \models \neg\varphi \Leftrightarrow$ not $M, s \models \varphi$.
- $M, s \models \varphi \rightarrow \varphi' \Leftrightarrow (M, s \models \neg\varphi)$ or $(M, s \models \varphi')$.
- $M, s \models \mathsf{EX}\varphi \Leftrightarrow$ for some traces σ such that $\sigma[0] = s$, $M, \sigma[1] \models \varphi$.
- $M, s \models \mathsf{AX}\varphi \Leftrightarrow$ for all traces σ such that $\sigma[0] = s$, $M, \sigma[1] \models \varphi$.
- $M, s \models \mathsf{E}(\varphi\, \mathcal{U}\, \varphi') \Leftrightarrow$ for some traces σ such that $\sigma[0] = s$, there exists a $j \geq 0$ such that $M, \sigma[j] \models \varphi'$, and for every $0 \leq k < j$, $M, \sigma[k] \models \varphi$.
- $M, s \models \mathsf{A}(\varphi\, \mathcal{U}\, \varphi') \Leftrightarrow$ for all traces σ such that $\sigma[0] = s$, there exists a $j \geq 0$ such that $M, \sigma[j] \models \varphi'$, and for every $0 \leq k < j$ satisfies $M, \sigma[k] \models \varphi$.

Some model checkers, particularly SMV, employ CTL as a language for expressing temporal properties of systems. The model checking problem for this logic is known to be linear on the size of the system and the formula being verified, as opposed to the case of CTL*, a more expressive computation tree logic for which the model checking problem is exponential on the size of the verified formula [6].

3 A Deontic Computation Tree Logic: dCTL

In this section we introduce dCTL, the logic that will be employed in order to specify, and later on verify, properties of fault tolerant systems. Formulas in this logic refer to properties of behaviors of colored Kripke structures, as defined in the previous section, in which a distinction between *normal* and *abnormal* states (and therefore also a distinction between normal and abnormal traces) is made. The logic dCTL is defined over CTL, with its novel part being the deontic operators $\mathbf{O}(\psi)$ (obligation), $\mathbf{P}(\psi)$ (permission), and $\mathbf{R}(\psi)$ (repair or recovery), which apply on a certain kind of path formula ψ. The intention of these operators is to capture the corresponding notion of *obligation*, *permission* and *repair* over traces. Intuitively, these operators have the following meaning:

- $\mathbf{O}(\psi)$: property ψ is obliged in every future state reachable via non-faulty transitions.
- $\mathbf{P}(\psi)$: there exists a normal execution, i.e., not involving any faults, starting from the current state and along which ψ holds.
- $\mathbf{R}(\psi)$: property ψ holds in every future faulty state, i.e., resulting from the immediate occurrence of a fault.

Clearly, obligation and permission will enable us to express intended properties which should hold in *all* normal behaviors and *some* normal behaviors, respectively. Repair, on the other hand, will enable us to express properties that should hold when faults occur; they will mainly serve the purpose of imposing restrictions on what should happen when faults occur, so that certain properties can be guaranteed.

These deontic operators have an implicit *temporal* character, since ψ is a path formula. As it will be made clearer later on, these operators, in combination with path formulas of the form $\psi \rightsquigarrow \psi'$ (operator $\rightsquigarrow$ is an *implication* between trace properties), provide some additional expressiveness with respect to CTL, without augmenting the expressiveness of the standard CTL operators A and E. As we will argue in the next section, these operators, used in a combined way, will be useful to state some fault tolerance properties straightforwardly.

Let us present the syntax of our logic. Let AP be a set $\{p_0, p_1, \ldots\}$ of atomic propositions; the sets Φ and Ψ of state and path formulas, respectively, are mutually recursively defined as follows:

$$\Phi ::= \top \mid p_i \mid \neg\Phi \mid \Phi \rightarrow \Phi \mid \mathsf{A}(\Psi \rightsquigarrow \Psi) \mid \mathsf{E}(\Psi \rightsquigarrow \Psi) \mid \mathbf{O}(\Psi \rightsquigarrow \Psi) \mid \mathbf{P}(\Psi \rightsquigarrow \Psi) \mid \mathbf{R}(\Psi \rightsquigarrow \Psi')$$

$$\Psi ::= \mathsf{X}\Phi \mid \Phi\, \mathcal{U}\, \Phi \mid \Phi\, \mathcal{W}\, \Phi$$

Other boolean connectives (here, state operators), such as $\wedge$, $\vee$, etc., can be defined as usual. Also, traditional temporal operators G and F can be expressed, as $\mathsf{G}(\phi) \equiv \phi\, \mathcal{W}\, \bot$, and $\mathsf{F}(\phi) \equiv \top\, \mathcal{U}\, \phi$. The standard boolean operators and the CTL quantifiers A and E have the usual semantics. Notice however that both CTL quantifiers and deontic operators apply to formulas involving the operator $\rightsquigarrow$. This operator relates two path formulas, and it represents a conditional. For instance, $\mathbf{O}(\psi \rightsquigarrow \psi')$ indicates that, for every normal trace σ starting in the current state, if σ satisfies ψ then it also satisfies ψ'. From a more technical perspective, which will be made clearer in later sections, the operator $\rightsquigarrow$ enables us to restrict the way in which path formulas can be combined in the scope of a state operator (a mechanism also exploited in other logics, particularly CTL^2). This will be essential for extending the expressiveness of CTL while retaining its model checking complexity.

Let us formally state the semantics of our logic. We start by defining the relationship $\vDash$, formalizing the satisfaction of dCTL state formulas in colored Kripke structures:

- $M, s \vDash \top$.
- $M, s \vDash p_i \Leftrightarrow p_i \in L(s)$, where $p_i \in AP$.
- $M, s \vDash \neg\varphi \Leftrightarrow$ not $M, s \vDash \varphi$.
- $M, s \vDash \varphi \rightarrow \varphi' \Leftrightarrow (M, s \vDash \neg\varphi)$ or $(M, s \vDash \varphi')$.
- $M, s \vDash \mathsf{A}(\psi \rightsquigarrow \psi') \Leftrightarrow M, \sigma \vDash \psi$ implies $M, \sigma \vDash \psi'$, for all traces σ such that $\sigma[0] = s$.
- $M, s \vDash \mathsf{E}(\psi \rightsquigarrow \psi') \Leftrightarrow M, \sigma \vDash \psi$ implies $M, \sigma \vDash \psi'$, for some traces σ such that $\sigma[0] = s$.

- $M, s \vDash \mathbf{O}(\psi \rightsquigarrow \psi') \Leftrightarrow$ for every $\sigma \in \mathcal{NT}$ such that $\sigma[0] = s$ we have that for every $i \geq 0$, $M, \sigma[i..] \vDash \psi$ implies $M, \sigma[i..] \vDash \psi'$.
- $M, s \vDash \mathbf{P}(\psi \rightsquigarrow \psi') \Leftrightarrow$ for some $\sigma \in \mathcal{NT}$ such that $\sigma[0] = s$ we have that for every $i \geq 0$, $M, \sigma[i..] \vDash \psi$ implies $M, \sigma[i..] \vDash \psi'$.
- $M, s \vDash \mathbf{R}(\psi \rightsquigarrow \psi') \Leftrightarrow$ for every trace σ such that $\sigma[0] = s$ we have that for every $i \geq 0$: if $s[i] \notin \mathcal{N}$, then $M, \sigma[i..] \vDash \psi$ implies $M, \sigma[i..] \vDash \psi'$.

The above satisfaction relation makes use of dCTL satisfaction for path formulas, whose definition is standard:

- $M, \sigma \vDash \mathsf{X}\varphi \Leftrightarrow M, \sigma[1] \vDash \varphi$.
- $M, \sigma \vDash \varphi \,\mathcal{U}\, \varphi' \Leftrightarrow$ there exists $j \geq 0$ such that $M, \sigma[j] \vDash \varphi'$ and for every $0 \leq k < j$, it holds that $M, \sigma[j] \vDash \varphi$.
- $M, \sigma \vDash \varphi \,\mathcal{W}\, \varphi' \Leftrightarrow$ either there exists $j \geq 0$ such that $M, \sigma[j] \vDash \varphi'$ and for every $0 \leq k < j$ it holds that $M, \sigma[j] \vDash \varphi$, or for every $j \geq 0$ we have that $M, \sigma[j] \vDash \varphi$.

As usual, we will denote by $M \vDash \varphi$ the fact that $M, s \vDash \varphi$ holds for every state s of M, and by $\vDash \varphi$ the fact that $M \vDash \varphi$, for every colored Kripke structure M. We will often employ the shorthand $\mathbf{O}(\psi)$, meaning $\mathbf{O}(\top \rightsquigarrow \psi)$ (similarly for other operators and quantifiers). We also apply path operators to state formulas (we just used $\top$ in $\mathbf{O}(\top \rightsquigarrow \psi)$ as a path formula). This can be done thanks to the fact that every state formula φ can be expressed as a path formula, by $\bot \,\mathcal{U}\, \varphi$.

The above introduced deontic operators enjoy some useful properties, some of which we enumerate below. In the following properties, we use φ and φ' for state formulas and ψ for path formulas:

1. $\mathbf{O}(\bot) \equiv \mathbf{O}(\psi) \wedge \mathbf{O}(\neg\psi)$, where $\neg\psi$ denotes the negation of ψ, obtained using the dual temporal operators of ψ and pushing the negation inwards.
2. $\mathbf{O}(\top) \equiv \top$
3. $\mathbf{P}(\bot) \equiv \bot$
4. $\mathbf{R}(\bot) \vdash \mathsf{AG}\varphi \leftrightarrow \mathbf{O}(\varphi)$
5. $\mathbf{R}(\bot) \vdash \mathsf{EG}\varphi \leftrightarrow \mathbf{P}(\varphi)$
6. $\mathbf{R}(\top) \equiv \top$
7. $\mathbf{R}(\bot) \rightarrow \mathbf{P}(\top)$
8. $\mathbf{O}(\varphi) \wedge \mathbf{O}(\varphi') \rightarrow \mathbf{O}(\varphi \wedge \varphi')$
9. $\mathbf{O}(\varphi) \vee \mathbf{O}(\varphi') \rightarrow \mathbf{O}(\varphi \vee \varphi')$
10. $\mathbf{P}(\varphi \wedge \varphi') \rightarrow \mathbf{P}(\varphi) \wedge \mathbf{P}(\varphi')$
11. $\mathbf{P}(\varphi) \vee \mathbf{P}(\varphi') \rightarrow \mathbf{P}(\varphi \vee \varphi')$
12. $\mathbf{R}(\varphi) \wedge \mathbf{R}(\varphi') \rightarrow \mathbf{R}(\varphi \wedge \varphi')$
13. $\mathbf{R}(\varphi) \vee \mathbf{R}(\varphi') \rightarrow \mathbf{R}(\varphi \vee \varphi')$

Let us briefly explain these properties. Property 1 states that expressing that false is obliged (which is equivalent to saying that there will eventually be a fault) is the same as having contradicting obligations. Property 2 expresses that saying that true is obliged is equivalent to true. Similar properties hold for the permission operator. Property 3 indicates that false cannot be allowed. The deduction rules state that, in the absence of faults, the deontic operators can be expressed

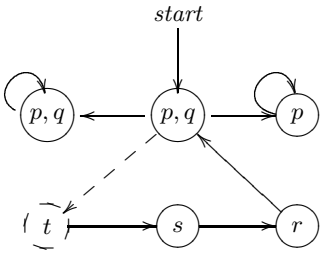


Fig. 1. A simple colored Kripke structure

using standard CTL. The properties of the operator **R** state that true always holds after a fault, while $\mathbf{R}(\bot)$ expresses that there will be no further faults in the future; this last expression implies $\mathbf{P}(\top)$, i.e., that there exist some good executions. Properties 8-13 relate the deontic operators to the standard boolean connectives. Due to space restrictions, we are unable to include the proofs of these properties in this paper; most of them can be proved straightforwardly resorting to the semantics of the involved operators.

In order to illustrate the semantics of the deontic operators, consider the colored Kripke structure in Figure 1, where the set of involved propositional variables is $\{p, q, r, s, t\}$, and each state is labeled by the set of propositional variables that hold in it. Also, the states that are the target of dashed arcs are abnormal states, also dashed, while the remaining ones are normal (i.e., dashed arcs are used for denoting transitions to faulty states, and the only faulty state in this model is the one labeled with t). It is obvious then that in every state of normal paths from the state indicated with *start*, p holds, which in dCTL is expressed as $\mathbf{O}(p)$. Also, there exist normal executions for which $p \wedge q$ always holds, expressed in dCTL as $\mathbf{P}(p \wedge q)$. On the other hand, the repair operator enables us to express properties regarding faulty states, and therefore also faulty executions. For instance, we can express that, immediately after every reachable fault, t holds, and a state in which r holds can be reached. In dCTL, these properties can be expressed as $\mathbf{R}(t)$ and $\mathbf{R}(\mathsf{F}r)$, respectively.

Finally, notice that other deontic operators, especially the *prohibition*, can be expressed using the above introduced ones. Prohibition can be characterized as $\mathbf{F}(\psi) = \neg\mathbf{P}(\psi)$. Intuitively, a (trace) property is forbidden when it cannot be true in a normal behavior. In other words, if such a property is continuously true in a trace, this trace contains some faults.

4 Fault Tolerance Reasoning in dCTL

Now that we have introduced our logic, let us start describing its use for expressing properties of systems in which faults might occur. We will illustrate the use of the logic using a few examples of typical fault tolerance situations.

Fig. 2. A simple model of a memory cell, without faults

4.1 A Memory Cell

Let us consider a system composed of a simple memory cell, which stores a bit of information and supports reading and writing operations. Such a simple system can be characterized as the Kripke structure shown in Figure 2, where each state maintains the current value of the memory cell (m_i, for $i = 0, 1$) and the last write operation that was performed (w_i, for $i = 0, 1$). Obviously, in this system the result of a reading depends on the value stored in the cell. Thus, a property that one might associate with this model, is that the value read from the cell coincides with that of the last writing performed in the system. This property can straightforwardly be expressed using CTL, as follows: $\mathsf{AG}((m_0 \rightarrow w_0) \wedge (m_1 \rightarrow w_1))$. This can be considered part of the *requirements specification* that the implementation described in Fig. 2 is expected to satisfy. Of course, this system expresses ideal behavior and does not take into account faults of any kind, making the use of the deontic operators unnecessary. So let us consider some faults in this scenario. Suppose that, when a bit's value is 1, it can unexpectedly lose its charge and turn into a 0. In this case, the above implementation cannot guarantee the specification is satisfied, since it is obvious that, if after writing 1 in the cell the described fault occurs and a reading is performed, a 0 will be read instead of the last written value 1.

So, the above model must be altered in order to cope with the possibility of the described fault occurring. A typical mechanism for dealing with this situation in fault tolerance is via *redundancy*. For instance, one might decide to implement the same system now using three memory bits instead of one. Writing operations are performed simultaneously in the three bits, whereas reading operations will return the value that is repeated at least twice in the memory bits (known as voting), and write it back in all three of them. The resulting system is depicted in Figure 3. Each state in this model is described by a variable w_i which records the last writing operation performed, together with three bits, described by boolean variables c_0, c_1 and c_2. The occurrence of a fault, which changes a bit with value 1 to hold a 0, is represented by a dashed line. Faulty states (indicated by dashed circles) are those resulting from a fault occurrence. Standard continuous lines denote normal transitions between states, representing reading or writing operations.

Notice that the reading operation is defined in a different way, in the presence of redundancy; the read value is the one that is repeated the most, so reading a 1 can be logically expressed as $r_1 = (c_0 \wedge c_1) \vee (c_0 \wedge c_2) \vee (c_1 \wedge c_2)$. That is, the value read is a 1 if there are at least two "one bits" in the memory cell with redundancy. With r_1 defined, r_0 is defined simply as its negation.

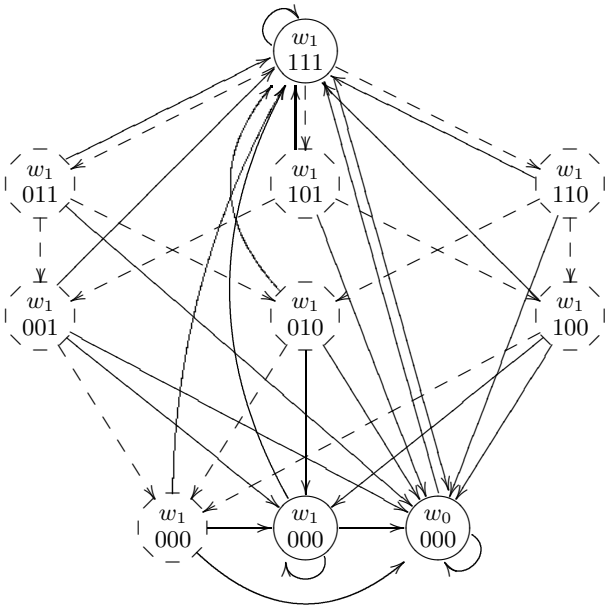


Fig. 3. A model of a memory cell, augmented with redundancy to deal with faults

Now let us discuss the properties one might expect of this augmented memory cell model. The original requirements of our model need to be updated to refer to our new implementation for reading a value (m_i is replaced by the above defined r_i): $\mathsf{AG}((r_0 \rightarrow w_0) \wedge (r_1 \rightarrow w_1))$. Of course, this property does not hold in the model, if faults occur. Still, this property is useful, since its verification, e.g. via model checking, would produce counterexamples that help us understand the situations in which our requirements are violated, in scenarios involving faults.

Besides the previous property, one of the most obvious properties one might be interested in is that, as long as no faults occur, the specification is guaranteed to hold. This can be thought of as a verification that the fault tolerance mechanism incorporated into to original system does not affect the satisfaction of the requirements specification when no faults occur. This first example of a fault tolerance property can be expressed naturally using obligation, in the following way:

$$\mathbf{O}((r_0 \rightarrow w_0) \wedge (r_1 \rightarrow w_1))$$

Let us start expressing properties of faulty scenarios. The motivation for introducing fault tolerance mechanisms is to be able to maintain the system behaving correctly even in the presence of faults. Of course, not every faulty scenario will maintain correct system behavior, so the general invariant property that we originally had becomes a *conditional* invariant, asserting that it will hold as long as fault occurrence is constrained. For example, for our memory cell with redundancy, we could say that the read value will coincide with the last written value

even in the presence of faults, but as long as, whenever a fault occurs, no further faults happen before a read or write operation is performed. In dCTL, this is expressed as follows as follows:

$$\mathbf{R}((\textsf{not-too-broken}\,\mathcal{U}\,\textsf{bits-coincide}) \rightsquigarrow (r_i \rightarrow w_i))$$

where $\textsf{not-too-broken} = \mathbf{P}(\top) \vee r_1$ (at most one fault has occurred since last read/write), and $\textsf{bits-coincide} = (c_0 \leftrightarrow c_1) \wedge (c_0 \leftrightarrow c_2)$ (capturing that the three bits coincide, always a consequence of a read or write). Notice that the subformula to the right of $\rightsquigarrow$ is not restricted to normal behaviors, since it neither uses obligation nor permission operators. But the subformula to the left of $\rightsquigarrow$ restricts what must happen when faults occur, as we wanted: whenever a fault occurs the system must transit nonfaulty transitions, until a read or write operation is performed. This formula is an example of the use of the repair operation. It also employs $\mathbf{P}(\top)$, which expresses that the current state is a normal one (recall the restriction of colored Kripke structures that says that normal states must have at least a normal successor).

The pattern $\mathbf{R}(\psi \rightsquigarrow \phi)$ is a useful one in fault tolerance settings: it expresses that the state property ϕ is guaranteed to hold even in the presence of faults, as long as whenever a fault occurs, the system behaves as ψ indicates. Various interesting engineering questions arise in relation to this pattern (which we do not deal with in this paper); for instance, given a state formula ϕ, one might be interested in synthesizing the weakest formula ψ such that the previous pattern formula holds.

Another interesting property that dCTL enables us to express is that, regardless of how many consecutive faults occur, if the system is in a faulty state, i.e., a fault has just occurred (maybe immediately after another fault), the system always has the chance to behave in a way that the requirements of the system are reestablished. This is naturally expressed using permission, as follows:

$$\textsf{AG}(\neg\mathbf{P}(\top) \wedge \textsf{EX}\mathbf{P}(\top) \rightsquigarrow \textsf{EG}\phi)$$

Since our model does not have unrecoverable faulty states, the formula $\neg\mathbf{P}(\top) \wedge \textsf{EX}\mathbf{P}(\top)$ captures the property of a state being faulty (it is very easy to capture faulty states even in the presence of unrecoverable ones). Notice that the above "error avoidance" property holds in our model, since write operations always take us back to a state in which the requirements of the system are reestablished.

4.2 A Token Ring Protocol

Let us now consider another example. Suppose that we have a simple system composed of three connected nodes, whose activities are regulated via a token ring protocol. In an original system, the three nodes are connected in a ring topology, and a token is passed through by the nodes so that the node that has it in a particular time is the one with access to a particular resource, e.g., permission to send information across the network. It is not difficult to think of a few examples of properties that might be thought of as the requirements

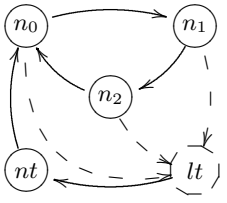


Fig. 4. A model of a token ring of nodes, where tokens can be lost

of the system, such as there is always exactly one node who has the token, and whenever a node has a token, it eventually passes it to the next one in the ring.

A simple fault that can be conceived in this context is one in which, due to the unreliability of the medium, the token might be lost when being transmitted from a node to the next one. If the period that each node has the token is fixed, then a fault detection mechanism can be easily implemented using a timeout (if a node have not seen the token for more than the time limit for each node, times the number of nodes). An abstraction of this situation, including the fault detection and a recovery approach, is depicted in Figure 4. The states n_i correspond to the token being held by node i; when the token is lost, no node has it, and when the detection of the missing token is established, a new token is created, and given to node 0.

The requirements on this system can be straightforwardly specified using CTL, as follows:

$$\mathsf{AG}((n_0 \wedge \neg n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge \neg n_1 \wedge n_2))$$
$$\mathsf{AG}(n_i \rightarrow \mathsf{AX}(n_{i\oplus 1}))$$

where $\oplus$ is addition modulo three. Notice that, for the sake of simplicity, we assume that each node has the token for exactly one instant of time (in the next step, the token has to belong to the next node). If one wants to check that these properties hold when no faults occur in the system, that can be expressed (and later on verified) using dCTL, in a similar way as for our previous example, i.e., by using obligation:

$$\mathbf{O}((n_0 \wedge \neg n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge \neg n_1 \wedge n_2))$$
$$\mathbf{O}(n_i \rightsquigarrow \mathsf{AX}(n_{i\oplus 1}))$$

The second of the above original requirements, though guaranteed when no faults occur, fails in any scenario in which at least a fault occurs. This is a case in which a desirable property needs to be *relaxed*, rather than given up, due to faults: the requirement that the token must be passed to the next node in the *next* instant is transformed into the token being passed at *some future* moment to the next node. That is, the second of our requirements is relaxed into the following: $\mathsf{AG}(n_i \rightarrow \mathsf{AF}(n_{i\oplus 1}))$. This is a progress property that holds for the system, provided it behaves in a strongly fair fashion. Notice that, even though strong fairness in not expressible in CTL, this constraint is typically incorporated

by various model checkers for the verification of liveness properties, and our property is a progress one, a particular case of liveness.

Another interesting fault tolerance property is one that expresses that faults are the only responsible of the token being lost. In other words, if the token is held by a particular node n_i, the token will be passed to the next node, or a fault will occur. This is expressed in dCTL in the following way:

$$\mathsf{AG}(n_i \rightarrow \mathsf{X}(\neg \mathbf{P}(\top) \vee n_{i \oplus 1}))$$

With the two simple case studies presented in this section, we tried to show examples of common properties of interest in the context of fault tolerant systems that can be, in our opinion, naturally expressed in dCTL. Other more general properties of fault tolerant systems, such as the concepts of *closure* and *convergence*, as described in [1], can also be expressed in a direct way. Closure serves the purpose of expressing that, given a state formula ϕ characterizing a required property of a system, φ is (inductively) preserved by the system by non-faulty transitions. In dCTL, this is expressed as $\mathbf{O}(\phi \rightsquigarrow \mathsf{AX}(\varphi))$. Convergence, on the other hand, allows one to express that, from any state satisfying certain property ϕ' (e.g., indicating that the system might be "'mildy" broken), if no further faults occur then the system always comes back to a state satisfying φ (i.e., it eventually recovers from the fault). This can be expressed separated in two parts. First, we express that from any normal state satisfying φ', if we move through nonfaulty transitions, we can eventually reach a state in which φ holds: $\mathbf{O}(\varphi' \rightsquigarrow \mathsf{AF}(\varphi))$. Second, we say that whenever a fault occurs, if we move through states that are normal or satisfy φ', then we can eventually reach a state in which φ holds: $\mathbf{R}(\mathsf{G}(\varphi' \vee \mathbf{P}(\top)) \rightsquigarrow \mathsf{F}\varphi)$. Some of the properties we dealt with in our examples can be thought of as variants of these two concepts.

5 Expressivity and Complexity of dCTL

In this section we show some results regarding the expressiveness and complexity of our logic dCTL. The complexity results enable us to show not only that the above properties of fault tolerant systems can be automatically checked, but also that checking them can be done in polynomial time with respect to the sizes of the model and the verified formula. We start by showing that dCTL formulas can be model checked, by providing a characterization of our logic into the more expressive logic CTL*. This characterization, which is not difficult to devise, introduces a fresh propositional letter n in the encoding, to "mark" normal behaviors. This translation is formalized in the following definition.

Definition 1. *The translation τ from dCTL formulas over an alphabet AP, to CTL* formulas over the alphabet $AP \cup \{n\}$, for some symbol $n \notin AP$, is defined as follows:*

- $\tau(\top) = \top$.
- $\tau(p_i) = p_i$.
- $\tau(\neg\varphi) = \neg\tau(\varphi)$.

- $\tau(\varphi \to \varphi') = \tau(\varphi) \to \tau(\varphi')$.
- $\tau(\mathsf{A}(\psi \rightsquigarrow \psi')) = \mathsf{A}(\tau(\psi) \to \tau(\psi')))$.
- $\tau(\mathsf{E}(\psi \rightsquigarrow \psi')) = \mathsf{E}(\tau(\psi) \to \tau(\psi'))$.
- $\tau(\mathbf{O}(\psi \rightsquigarrow \psi')) = \mathsf{A}(\mathsf{G}n \to \mathsf{G}(\tau(\psi) \to \tau(\psi')))$
- $\tau(\mathbf{P}(\psi \rightsquigarrow \psi')) = \mathsf{E}(\mathsf{G}n \wedge \mathsf{G}(\tau(\psi) \to \tau(\psi')))$
- $\tau(\mathbf{R}(\psi \rightsquigarrow \psi')) = \mathsf{A}(\mathsf{G}(\neg n \to (\tau(\psi) \to \tau(\psi'))))$
- $\tau(\mathsf{X}\varphi) = \mathsf{X}(\tau(\varphi))$.
- $\tau(\varphi \,\mathcal{U}\, \varphi') = \tau(\varphi) \,\mathcal{U}\, \tau(\varphi')$.
- $\tau(\varphi \,\mathcal{W}\, \varphi') = \tau(\varphi) \,\mathcal{W}\, \tau(\varphi')$.

The above translation from dCTL to CTL* is semantics preserving. The following mapping between Kripke structures and colored Kripke structures enables us to argue about the semantics preservation.

Definition 2. *Let $M = \langle S, R, L\rangle$ be a Kripke structure defined over an alphabet $AP \cup \{n\}$. From M, we define the colored Kripke structure $M^* = \langle S, R, L', \mathcal{N}\rangle$ over the alphabet AP, in the following way:*

- *L' is L restricted to AP.*
- *$s \in \mathcal{N} \Leftrightarrow M, s \vDash n$.*

The following theorem shows that our embedding of dCTL in CTL* is semantics preserving.

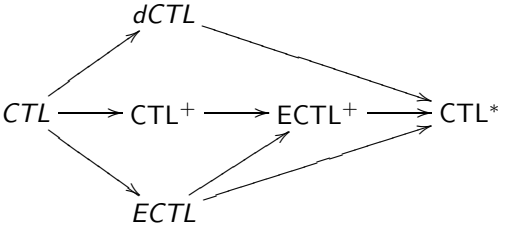
Theorem 1. *For every $M = \langle S, R, L\rangle$ defined over an alphabet $AP \cup \{n\}$, and every dCTL formula φ over AP, the following holds:*

$$M^* \vDash_{dCTL} \varphi \Leftrightarrow M \vDash_{CTL^*} \tau(\varphi).$$

Proof. By induction on the structure of formulas.

It is worth noting that our translation of dCTL deontic operators to CTL* involves some CTL* formulas which are not expressible in CTL. In particular, the formula $\neg\mathbf{P}(p \rightsquigarrow \mathsf{X}p)$, which is translated to $\mathsf{A}(\mathsf{F}\neg n \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$ in CTL*, is not expressible in CTL, nor in none of its extensions CTL$^+$, ECTL and ECTL$^+$. This expressiveness result, which follows from properties given in [9], is summarized in the following theorem.

Theorem 2. *The expressive powers of logics CTL, dCTL, CTL$^+$, ECTL, ECTL$^+$ and CTL*, are related by the following diagram of inclusions:*



Proof. A proof of $\mathsf{AF}(p \wedge \mathsf{X}\neg p)$ not being expressible in ECTL^+ can be found in [9] (cf. Theorem 5 therein). We can use a similar argument to prove that $\mathsf{A}(\mathsf{F}(\neg n) \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$ is not expressible in ECTL^+ either. Consider the sequence of models $N_1, N_2, N_3, \ldots$ and $M_1, M_2, M_3, \ldots$ used in Theorem 5 of [9], and set n to true in every state of these models. Since we have that $M_i, a_i \vDash \mathsf{AF}(p \wedge \mathsf{X}\neg p)$, we also have that $M_i, a_i \vDash \mathsf{A}(\mathsf{F}(\neg n) \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$. Since it is also the case that $N_i, a_i \nvDash \mathsf{AF}(p \wedge \mathsf{X}\neg p)$, for every i, then it must be the case that $N_i, a_i \nvDash \mathsf{A}(\mathsf{F}(\neg n) \vee \mathsf{F}(p \wedge \mathsf{X}\neg p))$. Taking into account that these structures cannot be distinguished by ECTL^+ formulas, it is straightforward the fact that $\neg\mathbf{P}(p \rightarrow \mathsf{X}p)$ is not expressible in ECTL^+. Moreover, this formula is not expressible in any of the logics CTL, ECTL or CTL^+, which are sublogics of ECTL^+.

This same theorem can also be extended to prove that some dCTL formulas are not expressible in other related logics, particularly CTL^2. The argument for the proof is essentially the same used in the above proof.

The model checking problem for CTL, ECTL, CTL^+ and CTL^2 is in P, while for ECTL^+ and CTL^* this problem is PSPACE-complete. Our logic is more expressive than CTL and is able to express formulas not expressible in ECTL^+, so a natural concern is whether the model checking problem for our logic is PSPACE-complete, which would be an unwanted, but reasonable, price paid for its expressiveness. As we show below, the model checking problem for dCTL maintains a polynomial complexity. This, combined with the fact that dCTL is able to express properties not expressible in other known "efficient" (in the sense that their model checking is in P) sublogics of CTL^*, make our logic a novel fragment of CTL^*.

Theorem 3. *The model checking problem for dCTL is in P.*

Proof. The main idea behind the proof is the adaptation of the algorithm described in [2] for CTL model checking, to support also checking deontic formulas. These additional processes can be done in polynomial time, using reachability algorithms.

Temporal models are implemented as graphs, so our set $\mathcal{N}$ of colored Kripke structures can be captured simply by adding a boolean variable n, set to true in exactly those states that belong to $\mathcal{N}$ (recall that, according to our restriction on colored Kripke structures, if n is true in some state s, then n is true in some successor of s). In order to check $M \vDash \varphi$, we start by calculating the sets $Sat(\psi) = \{s \mid M, s \vDash \psi\}$, for every subformula ψ of φ, starting from the subformulas at the bottom in the syntax tree of φ. The main technical difficulty is avoiding the exponential blow up in the translation of formulas of the form $\mathsf{A}(\psi \rightsquigarrow \phi)$ and $\mathsf{E}(\psi \rightsquigarrow \phi)$, etc. This blow up is avoided since these quantifiers always apply to a boolean combination of at most two path formulas. These formulas can then be checked using the process for the equivalent formulas in CTL.

It remains to show how to check formulas of the form $\mathbf{O}(\psi \rightsquigarrow \psi')$ and $\mathbf{P}(\psi \rightsquigarrow \psi')$. For the sake of simplicity, and without loss of generality, we can restrict the analysis to deontic operators applied to a single path formula (it is known

that implications of path formulas in the scope of a path quantifier can be translated to a state formula of a fixed length). Consider the formula $\mathbf{O}(\psi)$, which is equivalent to $\mathsf{A}(\mathsf{G}n \rightarrow \mathsf{G}(\psi))$. In order to build the set $Sat(\mathbf{O}(\psi))$, we can restrict the building process to states where n is true, and calculate that this set of states satisfies $\mathsf{A}(\psi)$. This can simply be checked using the model checking algorithm for CTL. Now consider $\mathbf{P}(\psi)$. This formula is equivalent to the CTL* formula $\mathsf{E}(\mathsf{G}n \wedge \mathsf{G}\psi)$. In order to build the set $Sat(\mathbf{P}(\psi))$, we check that there exists some path of states satisfying n where $\mathsf{G}\psi$ is true; this is done by checking $\mathsf{E}\psi$ for the nodes satisfying n (this can be done in polynomial time, inductively). Then, $s \in Sat(\mathbf{P}(\psi))$ if there is some successor of these states which satisfies both n and $\mathsf{E}\psi$. Finally, checking $\mathbf{R}(\psi)$ demands a similar technique. In summary, these processes can be performed using a depth-first search, and the algorithms for checking CTL formulas. Our extra checking processes are polynomial, therefore the final model checking algorithm is also polynomial with respect to the size of the model and the length of the formula.

6 Conclusions and Future Work

We have proposed a computation tree logic especially tailored for describing temporal properties of fault tolerant systems, and employing temporal deontic operators for this purpose. The deontic operators, which help in making a distinction between normal and abnormal states and behaviors, provide an expressiveness that is sufficiently rich for describing various properties of interest in the context of fault tolerance. We showed that some formulas expressible in our logic cannot be expressed in other known fragments of CTL*, including ECTL$^+$ and its sub-logics. However, and as opposed to the case for ECTL* and CTL*, for which the model checking problem is PSPACE-complete, model checking our logic dCTL is in P.

These results, together with our arguments regarding the usefulness of the logic for fault tolerance system specification, make it an interesting fragment of CTL*. In order to argue about its usefulness, we have developed two small case studies of fault tolerance situations, which despite their simplicity enabled us to illustrate the expressivity of the logic. Expressing temporal properties regarding fault tolerance could alternatively be achieved by a more "low level" approach, e.g., directly referring to faulty states via some atomic state formula capturing exactly such states. We believe that our deontic operators provide an indirect, higher level, way of referring to faults in the expression of fault tolerance properties, capturing some patterns useful in this context. Moreover, properties of deontic operators allow one to reason about formal descriptions at a higher level of abstraction.

We are currently exploring various lines of future work. We are developing more complex examples, and we are experimenting with the use of a μ-calculus model checker, Mucke, used as a target to express dCTL formulas. We are also analyzing alternative deontic operators that would provide an expressive power equivalent to that of our current version of the logic, but featuring a more intuitive reading. Also, we have not been concerned so far about providing an actual

formalism in which the system, the associated faults and the fault tolerance mechanisms are described, in a methodologically sound way. We plan to develop such a setting, incorporating our logic in it.

Acknowledgements. The authors would like to thank Pedro D'Argenio and the anonymous referees for their valuable comments. This work was partially supported by the Argentinian Agency for Scientific and Technological Promotion (ANPCyT), through grant PICT PAE 2007 No. 2772. The fourth author's participation was also supported through ANPCyT grant PICT 2006 No. 2484.

References

1. Arora, A., Gouda, M.: Closure and Convergence: A Foundation of Fault-Tolerant Computing. IEEE Transactions on Software Engineering 19(11) (1999)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
3. Castro, P., Maibaum, T.: Deontic Action Logic, Atomic Boolean Algebras and Fault-Tolerance. Journal of Applied Logic 7(4) (2009)
4. Clarke, E., Draghicescu, I.: Expressibility Results for Linear Time and Branching Time Logic. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 428–437. Springer, Heidelberg (1989)
5. Clarke, E., Emerson, E., Sistla, A.: Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems 8(2) (1986)
6. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
7. Coenen, J.: Specifying Fault Tolerant Programs in Deontic Logic, Computing Science Notes 91/34, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands (1991)
8. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesis of Live Behaviour Models for Fallible Domains. In: Proc. of International Conference on Software Engineering ICSE 2011. IEEE Press, Los Alamitos (2011)
9. Emerson, E., Halpern, J.: "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. J. ACM 33(1) (1986)
10. French, T., McCabe-Dansted, J., Reynolds, M.: A Temporal Logic of Robustness. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 193–205. Springer, Heidelberg (2007)
11. Gnesi, E., Lenzini, G., Martinelli, F.: Logical Specification and Analysis of Fault Tolerant Systems through Partial Model Checking. Electronic Notes on Theoretical Computer Science, vol. 118. Elsevier, Amsterdam (2005)
12. Janowski, T.: On Bisimulation, Fault-Monotonicity and Provable Fault-Tolerance. In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349, pp. 292–306. Springer, Heidelberg (1997)
13. Magee, J., Maibaum, T.: Towards Specification, Modelling and Analysis of Fault Tolerance in Self Managed Systems. In: Proc. of International Workshop on Self-Adaptation and Self-Managing Systems SEAMS 2006. ACM Press, New York (2006)
14. McCabe-Dansted, J., French, T., Reynolds, M., Pinchinat, S.: On the Expressivity of RoCTL*. In: Proc. of the 16th International Symposium on Temporal Representation and Reasoning TIME 2009. IEEE Computer Society, Los Alamitos (2009)

A Machine-Checked Framework for Relational Separation Logic*

Juan Manuel Crespo[1] and César Kunz[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Universidad Politécnica de Madrid

Abstract. Relational methods are gaining growing acceptance for specifying and verifying properties defined in terms of the execution of two programs—notions such as simulation, observational equivalence, non-interference, and continuity can be elegantly cast in this setting. In previous work, we have proposed *program product construction* as a technique to reduce relational verification to standard verification. This method hinges on the ability to interpret relational assertions as traditional predicates, which becomes problematic when considering assertions from relational separation logic. We report in this article an alternative method that overcomes this difficulty, defined as a relational weakest precondition calculus based on separation logic and formalized in the Coq proof assistant. The formalization includes an application to the formal verification of the Schorr-Waite graph marking algorithm. We discuss additional variants of relational separation logic inspired by the standard notions of partial and total correctness, and extensions of the logic to handle non-structurally equivalent programs.

1 Introduction

Separation logic [15, 23, 24] is a formalism devised to verify pointer programs using local reasoning; its extensions and variants have been used successfully in a variety of large scale programs [30] and smaller but challenging examples [17], including lock-free algorithms [13].

Relational reasoning, on the other hand, provides an effective means to understand program behavior: in particular, it allows one to establish that the same program behaves similarly on two different runs, or that two programs execute in a related fashion. Relational judgments are often formalized by quadruples $\{\varphi\}\, c_1 \sim c_2\, \{\psi\}$, denoting that every pair of executions of c_1 and c_2 with initial states related by φ returns with final states related by ψ. Prime examples of relational properties include notions of simulation and observational equivalence, and 2-properties, such as non-interference and continuity.

Syntactic methods [7] have been developed to support relational reasoning. In particular, relational separation logic [29] is a variant of separation logic that supports

* Partially funded by European Projects FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS. César Kunz is funded by a Juan de la Cierva Fellowship, MICINN, Spain.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 122–137, 2011.

reasoning about two pointer programs; it embodies the conventional wisdom that casting program correctness as an equivalence between two programs is often more beneficial than functional verification. More concretely, relational separation logic is intended to prove program correctness by showing the equivalence between the program to be verified and a reference implementation: e.g. Yang [29] provides an elegant proof in relational separation logic that the Schorr-Waite graph marking algorithm is equivalent to depth-first search.

However, these syntactic methods suffer from two important caveats: on the one hand, these logics confine reasoning to structurally equivalent programs with equivalent guards; on the other hand, tool support is negligible, with the exception of recent work by Aleks Nanevski *et al* [22]—which focuses mainly on the specification and proof of a rich set of security policies and its static enforcement. Although the relational postconditions used to describe such policies can be arbitrary relations between pairs of initial, final heaps and results, this tool seems to be specially tailored to reason about two runs of the same program, rather than about two different programs. To some extent it is possible to circumvent such restriction by casting two different programs `P` and `P'` as a single program with a guard deciding which program to execute, i.e. `if x then P else P'`. However, this approach seems a bit awkward and it is not at all clear whether doing this can enable reasoning in terms of relational invariants—which is essential to keep invariants simple.

In recent work [4] we propose a technique—*product program construction*—that reduces relational program reasoning to traditional program reasoning—even for non-structurally equivalent programs. Perhaps more importantly, it enables the use of traditional verification tools, circumventing two of the main issues of techniques supporting relational reasoning. However, this method relies on the ability to interpret relational assertions (predicates on two states) as traditional assertions (predicates on one state), but this is not straightforward when using assertions from relational separation logic. More precisely, an issue arises when trying to interpret the relational assertion

$$R = \begin{pmatrix} p \\ q \end{pmatrix}$$

(two heaps h_1 and h_2 are related by R if p holds in h_1 and q holds in h_2) as a predicate $p \star q$. Note that this interpretation induces a loss of information: predicate R holds for a fixed partition of the heap while the latter holds for any partition of the heap. This loss of information renders our method unsound. Indeed, $\{P \star \mathsf{emp}\}\mathsf{skip};\mathsf{skip}\{\mathsf{emp} \star P\}$ is a valid separation logic judgment for all P, whereas the following relational judgment is not:

$$\begin{pmatrix} P \\ \mathsf{emp} \end{pmatrix} \mathsf{skip} \sim \mathsf{skip} \begin{pmatrix} \mathsf{emp} \\ P \end{pmatrix}$$

We present in this article an alternative approach that overcomes the difficulties of relational verification by product construction, based on a *weakest precondition calculus* for relational separation logic. The calculus is complete and formalized in the Coq proof assistant, and can be regarded as a first step towards providing tool support for relational methods that enables reasoning about heap manipulating programs.

The formalization provides a framework to reason about a small imperative language —using a deep embedding—with heap manipulating instructions very similar to the one

described in Yang's article. We have formalized its semantics and provided a soundness proof of the relational weakest precondition. Local reasoning is supported by proving that the calculus is compatible with the frame rule. Also, we have defined an alternative calculus ensuring total relational correctness relying on *variants* (or ranking functions) defining a well-founded order on states.

The Coq formalization has been used to provide a formal proof of the equivalence of Depth First Search and the Schorr-Waite graph marking algorithm, reproducing the proof of the Schorr-Waite graph marking algorithm performed by Yang. We have extended Yang's proof to the total relational correctness case, hence ensuring that both programs terminate. The formal proof of the Schorr-Waite algorithm required a slightly stronger loop invariant, indicating perhaps a small weakness in the original specification provided in Yang's article.

We also introduce an extension of the relational calculus beyond structurally equivalent program, preserving relational reasoning over loop invariants, and thus retaining the aforementioned advantages. We illustrate the application of the calculus with the validation of a complex program optimization.

Contents. The rest of the paper is structured as follows: Section 2 describes the formalization of the relational weakest precondition calculus, instantiated with a simple programming setting. In this section, we briefly review relational separation logic and present the main properties of the calculus: soundness and framing. Also, we present a variant of the calculus that ensures termination of both programs. Section 3 presents our main case study, the proof of equivalence between the Schorr-Waite graph marking algorithm w.r.t. depth-first search. Section 4 describes an extension to non-structurally equivalent code.

2 Formalization of Relational Separation Logic

We start this section by introducing a simple program setting and then we provide an overview of relational separation logic. Afterwards, we develop our relational calculus based on weakest precondition computation.

The programming language presented in Figure 1 is a mild extension of the typical setting used in standard separation logic [23] to include list expressions. List values are rather uncommon in similar formalizations of imperative languages but are included here to ease the description of the Depth First Search (DFS) algorithm, which uses a stack as auxiliary data structure. In the figure, α stands for a list variable. We let BExp denote the set of boolean expressions and Stmt the set of statements.

State model. We let $\mathcal{S}$ denote the set of states. A state comprises two components: the store and the heap. The store itself comprises two components to accommodate two types of expressions: natural numbers and lists. Each of the store components is modeled the usual way, as a finite mapping from scalar variables in Var_N to natural numbers and as a finite mapping from list variables in Var_L to lists of natural numbers. We assume that the sets of variables Var_N and Var_L are disjoint. We let $\mathsf{upd}(x, n, s)$ stand for the result of updating the variable x to value n in the store s.

$$
\begin{array}{rl}
\textbf{(integer expressions)} & E ::= 0 \mid 1 \mid E + E \mid E \times E \mid E - E \mid \mathsf{hd}(L) \\
\textbf{(boolean expressions)} & B ::= \mathsf{false} \mid B \Rightarrow B \mid E = E \mid E < E \mid L = L \\
\textbf{(list expressions)} & L ::= \alpha \mid \epsilon \mid E{::}L \mid \mathsf{tl}(L) \\
\textbf{(instructions)} & I ::= x := \mathsf{alloc}(E) \mid x := [E] \mid [E] := E \mid \mathsf{free}(E) \mid \\
 & \qquad x := E \mid \alpha := L \mid \mathsf{assert}(B) \\
\textbf{(statements)} & C ::= I \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c \mid \mathsf{skip}
\end{array}
$$

Fig. 1. Syntax of Programs

The heap is modeled as a finite mapping from locations (natural numbers) to values. The special location 0 is denoted null and cannot belong to the domain of a heap. Heaps are equipped with several operations such as look-up, free, fresh, disjoint union and interact in the expected way:

Expression	Meaning
$\mathsf{freshn}(h, n)$	base location for a sequence of n consecutive free cells in h;
$\mathsf{look}(h, n)$	value of the cell n in the heap h;
$\mathsf{mut}(n, m, h)$	result of setting the contents of cell n of heap h to m;
$\mathsf{dealloc}(h, n)$	result of freeing cell n from heap h;

Moreover, we let $\mathsf{dom}(h)$ stand for the set of allocated locations of heap h, and $h_1 \uplus h_2$ denote the disjoint union of heaps h_1 and h_2. In the actual Coq development, failure is captured in an error monad, but for simplicity we omit these details here. Much of the formalization is adapted from Nanevski *et al* [21].

Semantics of basic instructions. The semantics of an instruction i is modeled as a relation $[\![i]\!]$ on states; the rules are given in Fig. 2. The denotation of an instruction is a relation between states. States are noted as tuples (h, s_i, s_l) where h represents the heap and s_i and s_l denote the stores for integer and list variables, respectively. The instruction $x := \mathsf{alloc}(E)$ evaluates the expression E to a natural number n and then allocates n free contiguous heap cells, initializes them with value 0 and sets the value of x to the first allocated cell. The look up instruction $x := [E]$ evaluates expression E to a location n and if it is allocated it updates the value of variable x to the contents of the heap cell n. The mutation instruction $[E_1] := E_2$ evaluates E_1 to a location n and if n is a valid location in the current heap, this is modified so that it maps n to the result of evaluating E_2. A field access $x := E.f$ is used as a syntactic sugar of $x := [E{+}f]$, when the field identifier f represents a known offset. Similarly, we use $E_1.f := E_2$ as a syntax sugar of $[E_1{+}f] := E_2$. The instruction $\mathsf{free}(E)$ releases the heap cell allocated at the location represented by E. The assert instruction has blocking semantics. The remaining assignments for integer and list variables are completely standard.

Semantics of commands. The semantics $[\![c]\!]$ of a command is defined as a relation on states (big step style), using as auxiliary definition the semantics of boolean expressions, modeled as a function from states to booleans. The definitions are standard and omitted. Also, we denote $\langle c, \mu\rangle \rightsquigarrow \langle c', \mu'\rangle$ the small-step command semantics and we use $\rightsquigarrow^\star$ for its reflexive transitive closure. Obviously these two semantic styles are sound and

$$
\begin{array}{ll}
((h,s_i,s_l),(h',s_i',s_l')) \in [\![x := \mathsf{alloc}(E)]\!] & \doteq s_i' = \mathsf{upd}(x,m,s_i) \\
& \quad \wedge\ s_l' = s_l \wedge h' = h \uplus (\biguplus_{i=0}^{n-1} (m+i) \mapsto 0) \\
& \quad \wedge\ m = \mathsf{freshn}(h,n) \wedge n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![x := [E]]\!] & \doteq s_l' = s_l \wedge h' = h \wedge s_i' = \mathsf{upd}(x,\mathsf{look}(h,n),s_i) \\
& \quad \wedge\ n \in \mathsf{dom}(h) \wedge n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![[E_1] := E_2]\!] & \doteq s_i' = s_i \wedge s_l' = s_l \wedge h' = \mathsf{mut}(n,m,h) \\
& \quad \wedge\ n \in \mathsf{dom}(h) \wedge n \in ([\![E_1]\!]\ (h,s_i,s_l)) \\
& \quad \wedge\ m \in ([\![E_2]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![\mathsf{free}(E)]\!] & \doteq s_i' = s_i \wedge s_l' = s_l \wedge h' = \mathsf{dealloc}(h,n) \\
& \quad \wedge\ n \in \mathsf{dom}(h) \wedge n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![x := E]\!] & \doteq h' = h \wedge s_l' = s_l \wedge s_i' = \mathsf{upd}(x,n,s_i) \\
& \quad \wedge\ n \in ([\![E]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![\alpha := L]\!] & \doteq h' = h \wedge s_i' = s_i \wedge s_l' = \mathsf{upd}(\alpha,xs,s_l) \\
& \quad \wedge\ xs \in ([\![L]\!]\ (h,s_i,s_l)) \\
((h,s_i,s_l),(h',s_i',s_l')) \in [\![\mathsf{assert}(B)]\!] & \doteq [\![b]\!](h,s_i,s_l) \wedge h = h' \wedge s_i = s_i' \wedge s_l = s_l'
\end{array}
$$

Fig. 2. Semantics of basic instructions

complete w.r.t. each other, i.e. $[\![c]\!]\ \mu\ \mu'$ if and only if $\langle c,\mu\rangle \rightsquigarrow^\star \langle \mathsf{skip},\mu'\rangle$. Also, we say that a command c is φ-safe if for any μ such that $\varphi\ \mu$ there exists μ' and c' such that $\langle c,\mu\rangle \rightsquigarrow \langle c',\mu'\rangle$, i.e., c is not stuck in φ-states.

2.1 Relational Calculus

We introduce in this section the relational calculus establishing the validity of relational specifications. Relational judgments are formalized as quadruples of the form $\{\varphi\}\ c_1 \sim c_2\ \{\psi\}$, where φ and ψ are relations on states and c_1 and c_2 are programs, establishing a relation over every pair of executions of c_1 and c_2, as formalized in the following definition:

Definition 1 (valid relational judgment). *Two commands c_1 and c_2 satisfy the pre and post-relation φ and ψ, denoted by the judgment $\vDash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$ if for all states μ_1, μ_2 s.t. $[\![\varphi]\!]\ \mu_1\ \mu_2$ one of the following holds:*

- *c_1 diverges with initial state μ_1 iff c_2 diverges with initial state μ_2; or*
- *for all states μ_1' and μ_2' s.t. $[\![c_1]\!]\ \mu_1\ \mu_1'$ and $[\![c_2]\!]\ \mu_2\ \mu_2'$ we have $[\![\psi]\!]\ \mu_1'\ \mu_2'$.*

Assertions. Rather than representing assertions as syntactic objects, we have modeled them as relations between states. All of the assertions presented in Yang's work have a straightforward interpretation as state relations. The definition of some of them is shown in Figure 3. We let P, Q stand for relational assertions and p, q for standard separation logic assertions.

Adopting a shallow embedding of assertions provides extra flexibility by not committing beforehand to a particular logical language, and allows inheriting all the features of Coq's rich higher-order language. This proved to be convenient when defining

$$
\begin{array}{ll}
\mathsf{Same}\ st_1\ st_2 & \doteq st_1.h = st_2.h \\
\mathsf{emp2}\ st_1\ st_2 & \doteq st_1.h = empty \wedge st_2.h = empty \\
(P \star Q)\ st_1\ st_2 & \doteq \exists h_{11}\ h_{12}\ h_{21}\ h_{22}. \\
 & \quad st_1.h = h_{11} \uplus h_{12} \wedge st_2.h = h_{21} \uplus h_{22} \\
 & \wedge\ P(h_{11}, st_1.s_i, st_1.s_l)(h_{21}, st_2.s_i, st_2.s_l) \\
 & \wedge\ Q(h_{12}, st_1.s_i, st_1.s_l)(h_{22}, st_2.s_i, st_2.s_l) \\
\begin{pmatrix} p \\ q \end{pmatrix}\ st_1\ st_2 & \doteq p\ st_1 \wedge q\ st_2
\end{array}
$$

Fig. 3. Definition of some Relational Assertions

$$
\begin{array}{ll}
\mathsf{wp}(x := \mathsf{alloc}(E))\ \varphi\ (h, s_i, s_l) & \doteq \forall n\ m.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \wedge m = \mathsf{freshn}(h, n) \Rightarrow \\
 & \quad \varphi(h \uplus \biguplus_{i=0}^{n-1} (m + i) \mapsto 0, s_i, s_l) \\
\mathsf{wp}(x := [E])\ \varphi\ (h, s_i, s_l) & \doteq \forall n.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \Rightarrow \\
 & \quad \varphi(h, \mathsf{upd}(x, \mathsf{look}(h, n), s_i), s_l) \\
\mathsf{wp}([E_1] := E_2)\ \varphi\ (h, s_i, s_l) & \doteq \forall n\ m.\ n \in (\llbracket E_1 \rrbracket\ (h, s_i, s_l)) \wedge m \in (\llbracket E_2 \rrbracket\ (h, s_i, s_l)) \Rightarrow \\
 & \quad n \in \mathsf{dom}(h) \wedge \varphi(\mathsf{mut}(n, m, h), s_i, s_l) \\
\mathsf{wp}(\mathsf{free}(E))\ \varphi\ (h, s_i, s_l) & \doteq \forall n.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \Rightarrow \\
 & \quad n \in \mathsf{dom}(h) \wedge \varphi(\mathsf{dealloc}(h, n), s_i, s_l) \\
\mathsf{wp}(x := E)\ \varphi\ (h, s_i, s_l) & \doteq \forall n.\ n \in (\llbracket E \rrbracket\ (h, s_i, s_l)) \Rightarrow \varphi(h, \mathsf{upd}(x, n, s_i), s_l) \\
\mathsf{wp}(\alpha := L)\ \varphi\ (h, s_i, s_l) & \doteq \forall xs.\ xs \in (\llbracket L \rrbracket\ (h, s_i, s_l)) \Rightarrow \varphi(h, s_i, \mathsf{upd}(\alpha, xs, s_l)) \\
\mathsf{wp}(\mathsf{assert}(B))\ \varphi\ (h, s_i.s_l)) & \doteq \llbracket B \rrbracket(h, s_i.s_l) \wedge \varphi(h, s_i.s_l)
\end{array}
$$

Fig. 4. Weakest Precondition of basic instructions

a weakest precondition calculus ensuring termination, in which a well-founded relation must be provably decreasing throughout loop iterations—see Subsection 2.2.

Weakest precondition calculus for basic instructions. Most program verification tools rely on weakest precondition calculi rather than program logics: concretely, the prevailing means to verify programs against a pre-condition and a post-condition is to generate a set of proof obligations using a weakest precondition calculus, and finally to discharge the proof obligations using automatic or interactive provers. Our formalization supports a similar methodology for relational judgments, and provides a weakest precondition calculus that computes a set of proof obligations from relational judgments. The weakest precondition of a basic instruction i w.r.t. to a state predicate ϕ is again, a state predicate (a function taking states and returning propositions). Here instead of using λ-abstractions we write the state on the left side as arguments to the wp function. Moreover, by abuse of notation we use pattern matching, i.e. a state is noted as a tuple. The definition of the weakest precondition of the basic instructions is provided in figure 4. Its definition is straightforward and obviously sound w.r.t. the semantics.

Weakest precondition calculus for 2-statements. Our weakest precondition calculus wp_2 operates on 2-statements, which combine two structurally equivalent statements into a single construction. Formally, the set $\mathtt{Stmt}_2$ of 2-statements is defined

$$\mathsf{wp}_2\ \langle i_1, i_2\rangle\ \phi = \mathsf{wp}\ i_1\ (\lambda m_1.\ \mathsf{wp}\ i_2\ (\lambda m_2.\ \phi\, m_1\, m_2))$$

$$\mathsf{wp}_2\ (c_1; c_2)\ \phi = \mathsf{wp}_2\ c_1\ (\mathsf{wp}_2\ c_2\ \phi)$$

$$\mathsf{wp}_2\ (\mathsf{if}\ \langle b, b'\rangle\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2)\ \phi = \Psi_{b,b'} \wedge (b_{\langle 1\rangle} \Rightarrow \mathsf{wp}_2\ c_1\ \phi) \wedge (\neg b_{\langle 1\rangle} \Rightarrow \mathsf{wp}_2\ c_2\ \phi)$$

$$\mathsf{wp}_2\ (\mathsf{while}\ \langle b, b'\rangle\ \mathsf{do}\ c)\ \phi = \exists \varphi.\varphi \wedge \forall m_1, m_2.\ \Psi_{b,b'}\ m_1\ m_2 \wedge \Psi_\varphi\ m_1\ m_2 \wedge \Psi_\phi\ m_1\ m_2$$

where

$$\Psi_{b,b'} \doteq b_{\langle 1\rangle} \Leftrightarrow b'_{\langle 2\rangle} \qquad \textit{guard equivalence}$$
$$\Psi_\varphi \doteq \varphi \wedge b_{\langle 1\rangle} \Rightarrow \mathsf{wp}_2\ c\ \varphi \qquad \textit{invariant preservation}$$
$$\Psi_\phi \doteq \varphi \wedge \neg b_{\langle 1\rangle} \Rightarrow \phi \qquad \textit{valid postcondition}$$

Fig. 5. Relational weakest precondition calculus

inductively by the clauses: i) if i_1 and i_2 are instructions, then $[i_1, i_2]$ is a 2-statement; ii) if c, c_1, c_2 are 2-statements and b, b' are boolean expressions, then $c_1; c_2$, and if $\langle b, b'\rangle$ then c_1 else c_2, and while $\langle b, b'\rangle$ do c are 2-statements. Each 2-statement yields two structurally equivalent statements; we write $c \triangleright (c_1, c_2)$ to denote that c is a 2-statement whose left and right components are the statements c_1 and c_2 respectively. Conversely, any two structurally equivalent statements yield a 2-statement. Intuitively, a 2-statement encodes the simultaneous execution of its components, restricting the calculus to structurally similar programs. In Section 4 we explain how to remove this restriction by the application of a preliminary program transformation.

The weakest precondition calculus wp_2 is defined inductively on the structure of 2-statements; the rules are given in Figure 5, where $b_{\langle 1\rangle}$ and $b_{\langle 2\rangle}$ respectively denote the interpretation of the expression b in the first and second memories, and the extension of connectives to relations is defined in the usual way.

Frame rule. The frame rule lies at the very heart of any separation logic based verification framework, being the cornerstone of so called "local reasoning". In order to support modular verification, we have shown that it holds on the framework presented in this paper. Let P, Q, and R be relational assertions and c a 2-statement. Then, if

1. the proposition $\forall st_1, st_2.\ P\ st_1\ st_2 \Rightarrow \mathsf{wp}_2\ c\ Q\ st_1\ st_2$ holds; and
2. R is independent of the variables modified by c

then the following proposition holds:

$$\forall st_1, st_2.\ (P \star R)\ st_1\ st_2 \Rightarrow \mathsf{wp}_2\ c\ (Q \star R)\ st_1\ st_2$$

Intuitively, the hypothesis 1 and 2 implies that the only part of state that program c is allowed to inspect or operate on is described by P and any other part R will remain unchanged after its execution. This simplifying result has been systematically used to ease the verification of the Schorr-Waite algorithm.

One of the most challenging aspects of characterizing the frame rule in our setting is the fact that there is no syntax for assertions, so the customary side condition on R is formulated semantically by defining the modset of c, i.e. the set of modified variables, and requiring the validity of R to be independent of it.

Soundness. The calculus is sound, i.e. for all statements c_1 and c_2, and 2-statement c s.t. $c \triangleright (c_1, c_2)$, and assertions φ and ψ,

$$[\![\varphi]\!] \subseteq [\![\mathsf{wp}_2\ c\ \psi]\!] \quad \Longrightarrow \quad \vDash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$$

Moreover, the weakest precondition calculus is sound and complete w.r.t. relational separation logic, i.e. for all statements c_1 and c_2, and 2-statement c s.t. $c \triangleright (c_1, c_2)$, and assertions φ and ψ,

$$[\![\varphi]\!] \subseteq [\![\mathsf{wp}_2\ c\ \psi]\!] \quad \Longleftrightarrow \quad \vdash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$$

where $\vdash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$ is used to denote that the judgment is derivable in relational separation logic.

2.2 Total Correctness

One can modify the weakest precondition calculus wp_2 to enforce total correctness. To this end, one must provide for each while statement a variant relation between pairs of initial and final states, and prove that it is a well-founded order (i.e. no infinite descending chains) and that it decreases with each iteration. The clause for the while statement is modified accordingly:

$$\begin{aligned}\mathsf{wp}_2^{\mathsf{tc}}\ (\mathsf{while}\ \langle b, b'\rangle\ \mathsf{do}\ c)\ \phi\ \mu_1\ \mu_2 \doteq\ & \\ & \exists\varphi, \exists\mu, \varphi \wedge \forall m_1, m_2.\ (\Psi_{b,b'}\ m_1\ m_2 \wedge \Psi_\varphi\ m_1\ m_2 \wedge \Psi_\phi\ m_1\ m_2) \wedge \\ & \mathsf{wellfounded}(\mu) \wedge \forall m_1, m_2.\ (\varphi\ m_1\ m_2 \wedge [\![b]\!]\ m_1\ m_2 \Rightarrow \\ & \mathsf{wp}_2\ c\ (\lambda s_1, s_2.\ \mu\ (s_1, s_2)\ (m_1, m_2))\ m_1\ m_2)\end{aligned}$$

where $\Psi_{b,b'}$, Ψ_φ, and Ψ_ϕ are defined as in Figure 5 (replacing wp_2 by $\mathsf{wp}_2^{\mathsf{tc}}$), and μ stands for the variant relation. The predicate $\mathsf{wellfounded}(\mu)$ requires μ to be well-founded to establish the termination of the loop. Notice that we use wp_2 instead $\mathsf{wp}_2^{\mathsf{tc}}$ in the last line of the formulae above, to avoid redundancy on the verification of termination of c, which is already established by Ψ_φ. Then, assuming termination of instructions, we can prove total correctness, i.e. for all statements c_1 and c_2, and 2-statement c s.t. $c \triangleright (c_1, c_2)$, and assertions φ and ψ, and memories μ_1 and μ_2 s.t. $\varphi\ \mu_1\ \mu_2$,

$$[\![\varphi]\!] \subseteq [\![\mathsf{wp}_2^{\mathsf{tc}}\ c\ \psi]\!] \quad \Longrightarrow \quad \exists\mu_1', \mu_2'.\ [\![c_1]\!]\ \mu_1\ \mu_1' \wedge [\![c_2]\!]\ \mu_2\ \mu_2' \wedge \psi\ \mu_1'\ \mu_2'$$

Note that the shallow embedding of assertions plays a crucial role here, a partial application of the variant is used as argument for the wp. This would not be possible if we had established a syntax for the formulae through a deep embedding.

3 Verification of the Schorr-Waite Algorithm

The Schorr-Waite graph marking algorithm is a widely used case study, see Section 5. Yang [29] uses relational separation logic to prove the equivalence between the Schorr-Waite algorithm and depth-first search, and convincingly argues that the proof in relational separation logic is more elegant and more concise than an earlier functional

verification [28] of the SW algorithm in separation logic. In this section we report on a machine-checked proof of the Schorr-Waite algorithm using the weakest precondition calculus described in the previous section. The structure of the proof is similar to Yang's pen-and-paper proof [29]; one difference is that we prove total correctness rather than co-termination.

Algorithm and relational specification. DFS traverses a binary tree marking every node in a depth-first basis. In order to backtrack the tree traversal, it uses a stack as an auxiliary storage to keep track of the parent nodes that need to be revisited. The Schorr-Waite algorithm optimizes the space needed by DFS by removing the stack. The set of nodes to be revisited are encoded as a transformation on the heap structure: pushing a node in the stack is implemented as an inversion of the left edge that is traversed, removing a node from the stack is defined as restoring the original edge. Figure 6 shows a 2-statement merging the Schorr-Waite algorithm (marked with a gray shadow) with DFS.

Verification. We have used the Coq framework to verify the 2-statement in Figure 6 against the specification:

$$\begin{aligned} Pre &\doteq \mathsf{Same} \wedge c{=}c' \wedge \begin{pmatrix} \mathsf{noDang}\, G \wedge c \in G \cup \{\mathsf{nil}\} \\ \mathsf{noDang}\, G \wedge c' \in G \cup \{\mathsf{nil}\} \end{pmatrix} \\ Post &\doteq \mathsf{Same} \end{aligned}$$

where c and c' represent the corresponding tree roots and G denotes the set of tree nodes. The predicate $\mathsf{noDang}\, G$ states that G is a set of non-dangling pointers closed under heap reachability:

$$\mathsf{noDang}\, G \doteq \forall_\star x \in G.\ \exists lr.\ (x \mapsto l, r, -, -) \wedge l \in G \cup \{\mathsf{nil}\} \wedge r \in G \cup \{\mathsf{nil}\}$$

The additional condition $c \in G$ implies that the set of tree nodes reachable from the root c is a subset of G. The specification states that under initial heaps with the same tree structure with root c, SW and DFS terminate with the same final states.

The application of the wp_2 function to the 2-statement and the postcondition above returns a verification condition that contains an existential quantification for the loop invariant. We have used a slightly modified version of the invariant proposed by Yang [29]:

$$\mathsf{Same} \star \mathsf{uniq}\, \alpha \wedge \mathsf{Stack}\, p\, c\, \alpha \wedge p{=}p' \wedge \begin{pmatrix} \mathsf{noDang}\, G \wedge p \in G \wedge c \in G \\ \mathsf{noDang}\, G \wedge p' \in G \wedge \alpha \subseteq G \cup \{\mathsf{nil}\} \end{pmatrix}$$

Basically, the invariant establishes that no dangling pointers can be introduced during the algorithms execution, and provides a relation between the auxiliary stack storage used by DFS and its representation in the Schorr-Waite algorithm. This relation is formalized by the predicate Stack:

$$\begin{aligned} &\mathsf{Stack}\, p\, c\, \epsilon \doteq c{=}\mathsf{nil} \\ &\mathsf{Stack}\, p\, c\, a{::}\alpha \doteq \exists n_0, x.\ \mathsf{Stack}\, c\, n_0\, \alpha \star c{=}a \wedge \\ &\qquad \left[\begin{pmatrix} c \mapsto n_0, x, Marked, Left \\ c \mapsto p, x, Marked, Left \end{pmatrix} \vee \begin{pmatrix} c \mapsto x, n_0, Marked, Right \\ c \mapsto x, p, Marked, Right \end{pmatrix} \right] \end{aligned}$$

$$
\begin{array}{l}
\textsf{if } \langle\ c \neq \textsf{nil}\ , c' \neq \textsf{nil} \rangle \textsf{ then} \\
\quad \left[\begin{array}{ll} p := c.Left; & p' := c'.Left; \\ c.Mark := Marked & c'.Mark := Marked; \\ c.Current := isLeft; & c'.Current := isLeft; \\ c.Left := \textsf{nil} & \alpha := c' :: \epsilon \end{array}\right] \\
\textsf{else} \\
\quad \left[\begin{array}{l} p := \textsf{nil}\ , \\ \end{array}\begin{array}{l} p' := \textsf{nil}; \\ \alpha := \epsilon \end{array}\right] \\
\textsf{fi} \\
\textsf{while } \langle\ c \neq \textsf{nil}\ , \alpha \neq \epsilon \rangle \textsf{ do} \\
\quad \textsf{if } \langle\ p \neq \textsf{nil}\ , p' \neq \textsf{nil} \rangle \textsf{ then} \\
\qquad [\ m := p.Mark\ , m' := p'.Mark] \\
\quad \textsf{else} \\
\qquad [\ m := Marked\ , m' := Marked] \\
\quad \textsf{fi} \\
\quad \textsf{if } \langle\ p \neq \textsf{nil} \wedge m \neq Marked\ , p' \neq \textsf{nil} \wedge m' \neq Marked \rangle \textsf{ then} \\
\qquad \left[\begin{array}{ll} t := p.Left; & \\ p.Left := c; & \alpha := p' :: \alpha; \\ c := p; & p'.Mark := Marked; \\ p := t; & p'.Current := isLeft; \\ c.Mark := Marked; & p' := p'.Left \\ c.Current := isLeft & \end{array}\right] \\
\quad \textsf{else} \\
\qquad [\ d := c.Current\ , d' := (hd\ \alpha).Current] \\
\qquad \textsf{if } \langle\ d = isLeft\ , d' = isLeft \rangle \textsf{ then} \\
\qquad\quad \left[\begin{array}{ll} t := c.Left; & \\ c.Left := p; & \\ p := c.Right; & (hd\ \alpha).Current := isRight; \\ c.Right := t; & p' := (hd\ \alpha).Right \\ c.Current := isRight & \end{array}\right] \\
\qquad \textsf{else} \\
\qquad\quad \left[\begin{array}{ll} t := p; & \\ p := c; & p' := hd\ \alpha; \\ c := p.Right; & \alpha := tl\ \alpha \\ p.Right := t & \end{array}\right] \\
\qquad \textsf{fi} \\
\quad \textsf{fi} \\
\textsf{done}
\end{array}
$$

Fig. 6. Schorr-Waite and DFS 2-statement

In particular, when $c \neq \mathsf{nil}$, c is the top element in the stack α and p its left or right child. The remaining stack elements are related inductively. The difference with respect to Yang's invariant consists on the predicate $\mathsf{uniq}\ \alpha$, that states that the list α does not contain repeated elements. The need for this extra condition became evident when discharging the verification conditions in the Coq proof assistant.

Total correctness. We have also developed a total correctness argument for the Schorr-Waite algorithm using the total correctness version of the weakest precondition calculus presented earlier. Then, we extended the proof with the addition of a variant relation, a lexicographic order similar to the one used by Giorgino et al [12]: let (st_1, st_2) and (st'_1, st'_2) be pairs of states, then $\mathsf{var}(st_1, st_2)(st'_1, st'_2)$ iff one of the following holds:

- the number of unmarked nodes in (st_1, st_2) is smaller than the number of unmarked nodes in (st'_1, st'_2),
- the number of unmarked nodes in (st_1, st_2) and (st'_1, st'_2) is the same but the number of nodes in (st_1, st_2) with $Current$ field set to $isLeft$ is smaller than in (st'_1, st'_2), or
- the number of unmarked nodes and the number of nodes of (st_1, st_2) and (st'_1, st'_2) is the same but the size of the stack α in (st_1, st_2) is smaller than in (st'_1, st'_2).

We showed that this is a well-founded order and proved that it holds for the pre and post states of the loop body using the wp_2 calculus. In particular note that of the three ways to construct the order, the first one corresponds to a push, the second one to a swing and the third one to a pop operation.

4 Beyond Structurally Equivalent Programs

A common caveat of syntactic relational methods is the limited support for non structurally equivalent programs. Although this restriction can be circumvented in the setting of relational separation logic by using the embedding rule, the ability to reason in terms of relational loop invariants is still not supported.

In this section, we present a different strategy that cleanly extends the weakest precondition based calculus presented in Section 2 to cope with non structurally equivalent code. We enhance the previous formalism through a preliminary transformation that is performed on the programs to be verified. This syntactic transformation can yield structurally equivalent programs while retaining some semantic properties that ensure that the relational validity on the transformed programs also holds on the original programs.

Let us first make precise the notion of refinement we adopt. We say that c is a refinement of c', noted $c \succcurlyeq c'$ if the following conditions hold for all μ, μ', μ'' and σ:

- if $[\![c']\!]\ \mu\ \mu'$ then $[\![c]\!]\ \mu\ \mu'$;
- if $[\![c']\!]\ \mu\ \mu'$ and $[\![c]\!]\ \mu\ \mu''$ then $\mu' = \mu''$ and
- if c is σ-safe then c' is σ-safe.

We know, under this rather weak definition of refinement, that in order to establish a relational property on two programs c_1 and c_2, it is sufficient to establish such property for any two programs c'_1 and c'_2 s.t. $c_1 \succcurlyeq c'_1$ and $c_2 \succcurlyeq c'_2$:

$$\text{(RO)}\frac{}{\vdash c; d \succcurlyeq d; c}\ \mathsf{fv}(c) \cap \mathsf{fv}(d) = \emptyset \qquad \text{(SK)}\frac{}{\vdash c; \mathsf{skip} \succcurlyeq c}$$

$$\text{(IF1)}\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{assert}(b); c_1} \qquad \text{(IF2)}\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{assert}(\neg b); c_2}$$

$$\text{(WHU)}\frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(b); c; \mathsf{while}\ b\ \mathsf{do}\ c}$$

$$\text{(WHS)}\frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(b); c; \mathsf{assert}(\neg b)}$$

$$\text{(IFM)}\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c\ ;\mathsf{if}\ b'\ \mathsf{then}\ c'\ \succcurlyeq \mathsf{assert}(b \Leftrightarrow b'); \mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{assert}(b'); c}$$

$$\text{(LRS)}\frac{}{\vdash \mathsf{for}\ i{=}m\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(m \le n' \le n); \mathsf{for}\ i{=}m\ \mathsf{to}\ n'\ \mathsf{by}\ k\ \mathsf{do}\ c; \mathsf{for}\ j{=}i\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c[j/i]}$$

$$\text{(LT)}\frac{}{\vdash \mathsf{for}\ i{=}0\ \mathsf{to}\ n\ \mathsf{by}\ 1\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(n \bmod k = 0); \mathsf{for}\ i{=}0\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ (\mathsf{for}\ j{=}0\ \mathsf{to}\ k\ \mathsf{by}\ 1\ \mathsf{do}\ c[i{+}j/i])}$$

$$\text{(R-RI)}\frac{}{\vdash \mathsf{for}\ i{=}m\ \mathsf{to}\ n\ \mathsf{by}\ k\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(\left\lceil \frac{n-m}{k} \right\rceil = \left\lceil \frac{n'-m'}{k} \right\rceil); \mathsf{for}\ i{=}m'\ \mathsf{to}\ n'\ \mathsf{by}\ k\ \mathsf{do}\ c[(i{-}m'{+}m)/i]}$$

Fig. 7. Syntactic refinement rules (excerpt)

Lemma 1. *For all programs c_1 and c_2, and c_1', c_2' such that $c_1 \succcurlyeq c_1'$ and $c_2 \succcurlyeq c_2'$, if $\vDash \{\varphi\}\ c_1' \sim c_2'\ \{\psi\}$ then $\vDash \{\varphi\}\ c_1 \sim c_2\ \{\psi\}$, provided c_1' and c_2' are φ-safe.*

Figure 7 provides a set of syntactic rules deriving a refinement relation. For clarity, we introduce the statement for $i = m$ to n by k do c as a syntax sugar for statement $i := m$; while $i < n$ do c; $i := i{+}k$. As can be seen in the figure, the rules consist of basic structure transformations. The most complex rules are perhaps (LRS) and (LT), which perform loop range splitting and loop tiling, respectively. The set of refinement rules in Figure 7 is sound, i.e., it induces a refinement relation:

Lemma 2. *For all statements c and c', if $\vdash c \succcurlyeq c'$ then $c \succcurlyeq c'$.*

Example: vectorization of sum. Figure 8 presents a simple algorithm that computes the sum of the values of the node elements in a singly linked list. A program vectorization consists on relying on special purpose *SIMD* (single instruction, multiple data) instructions, taking advantage of the associativity and commutativity of the arithmetic computation performed in a program loop. Intuitively, for this particular example the vectorization consists in grouping the loop iterations in chunks of 4 iterations, and performing 4 addition operations simultaneously with the `_mm_add_epi32` instruction. Figure 9 shows the vectorized algorithm. Let n denote the length of the linked list pointed by `head`. The first loop iterates $n \div 4$ times and computes the summation of the first $4 \times (n \div 4)$ elements of the linked list, storing it in the 128-bits vector `sum`. The second loop computes the summation of the remaining $n \bmod 4$ elements and stores it in variable `rest`. The final value is computed by adding to the variable `rest` the partial results stored in the bit vector `sum`.

$$\frac{sum\,(list*\ head,\ int\ size)}{\begin{array}{l} rest := 0; \\ \mathsf{for}\ i = 0\ \mathsf{to}\ size\ \mathsf{by}\ 1\ \mathsf{do} \\ \quad rest := rest + head.val; \\ \quad head := head.next; \end{array}}$$

Fig. 8. Original version of `sum` algorithm

By applying a sequence of refinement steps over the original program one can obtain a pair of structurally similar programs. Then, providing a relational invariant becomes much simpler than verifying each of the programs functionally. Indeed, assume that the predicate $\mathsf{EqList}(\mathtt{head}, \mathtt{head}', \mathtt{size})$ holds as precondition, with inductive predicate EqList is defined by the following clauses:

$$\begin{array}{ll} \mathsf{EqList}(l_1, l_2, 0) & \doteq l_1 = l_2 = \mathsf{null} \\ \mathsf{EqList}(l_1, l_2, n{+}1) & \doteq \mathsf{EqList}(l_1', l_2', n) \wedge \exists v, l_1', l_2'.\ l_1 \mapsto (v, l_1') \wedge l_2 \mapsto (v, l_2') \end{array}$$

Then, in order to verify that original and vectorized algorithms compute the same value, i.e., that $rest = rest'$ holds as a relational postcondition, it is sufficient to establish the validity of loop invariants of the form:

$$sum[0]{+}sum[1]{+}sum[2]{+}sum[3] = rest$$

and

$$rest'{+}sum[0]{+}sum[1]{+}sum[2]{+}sum[3] = rest$$

Notice that these relational loop invariants are much simpler that those required in a functional verification of the algorithm.

5 Related Work

Relational methods and program verification techniques have been intimately connected since their origins. In particular, methods based on program refinement, program equivalence, and logical relations have been used widely to reason about program correctness. In this respect, it is perhaps surprising that relational program logics have only been introduced recently. Benton [7] develops a relational Hoare logic for a small imperative language and shows how program optimizations can be validated using relational reasoning. Other relational logics include Yang's relational separation logic [29] and Barthe, Grégoire and Zanella's probabilistic relational Hoare logic [5]. More recently, Nanevski, Banerjee and Garg developed a relational separation logic for Hoare type theory [22]. It extends Yang's logic to a richer programming and specification language, and is tailored for reasoning information flow; the logic is formalized in the Coq proof assistant; in contrast to our formalization, it uses a shallow embedding of programs. Independently, Beringer [8] provided a reconstruction of relational separation logic based on a notion of decomposition that allows reducing relational program logics to standard program logics; the soundness of the logic is formalized in the Isabelle

```
ssesum (list* head', int size)
  sum = _mm_set1_epi32(0);
  for i = 0 to size − 3 by 4 do
    curr := _mm_insert_epi32(curr, head'.val, 0);
    head' := head'.next;
    curr := _mm_insert_epi32(curr, head'.val, 1);
    head' := head'.next;
    curr := _mm_insert_epi32(curr, head'.val, 2);
    head' := head'.next;
    curr := _mm_insert_epi32(curr, head'.val, 3);
    head' := head'.next;
    sum := _mm_add_epi32(sum, curr);
  rest' := 0;
  for j = i to size by 1 do
    rest' := rest'+head'.val;
    head' := head'.next;
  rest' := rest'+_mm_extract_epi32(sum, 0)+_mm_extract_epi32(sum, 1)+
    _mm_extract_epi32(sum, 2) + _mm_extract_epi32(sum, 3);
```

Fig. 9. SSE optimized version of `sum` algorithm

proof assistant. In addition to these general-purpose logics, specialized relational logics have been developed for specific properties, and especially information flow [1].

On the formalization side, there have been many machine-checked accounts of separation logic in proof assistants, e.g. [3, 26], including some frameworks designed to support automated reasoning in separation logic [2, 11, 18, 20]. Moreover, the Schorr-Waite algorithm is a classical example in program verification, and has been verified formally using a variety of tools and techniques. Suzuki [25] provides an early machine-supported proof of the Schorr-Waite algorithm using an automated verifier for pointer programs. More recently, Bornat [10] provides a machine-checked proof of the algorithm in the Jape proof assistant. Subsequently, Mehta and Nipkow [19], Hubert and Marché [14], Bubel [6], Jacobs and Piessens [16] formalize the algorithm in Isabelle, Caduceus, KeY and VeriFast respectively. More recently, Giorgino et al [12] prove the correctness and termination of the algorithm in Isabelle, using refinement. All these formalizations use standard program logics.

6 Conclusion

Relational separation logic is a powerful tool devised for reasoning about the relation between heap manipulating programs. To the best of our knowledge, we have formalized in the Coq proof assistant the first certified weakest precondition calculus for relational separation logic. We illustrated its usefulness and scalability by proving a challenging case study: the correctness of the Schorr-Waite graph marking algorithm.

The Coq development has been done using ssreflect library which greatly improves the conciseness of the proofs. For example, the relational weakest precondition,

soundness proofs, the definition and specification and proof of the Schorr-Waite graph marking algorithm and Depth First Search take 1586 lines of specification and 3538 lines of proofs. We believe that the formalization of the verification setting and the formal proof of the algorithms poses no significant overhead over hand-written proofs.

In the future, it would be interesting to formalize the modular proof of the algorithm reported in [9] and to prove the equivalence between different implementations of ADTs; for the latter, we believe that the extensions to non-structurally equivalent code will prove crucial. Another line of work is to extend our formalization to reason about concurrent separation logic [27] and verify the correctness of lock-free algorithms [13].

Acknowledgement. The authors would like to thank Aleksander Nanevski for introducing ssreflect to us and patiently explaining some its main features.

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: Morrisett, G., Peyton Jones, S. (eds.) Principles of Programming Languages, pp. 91–102. ACM, New York (2006)
2. Appel, A.: Tactics for separation logic (January 2006) (unpublished manuscript), http://www.cs.princeton.edu/~appel/papers/septacs.pdf
3. Appel, A.W., Blazy, S.: Separation logic for small-step cminor. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 5–21. Springer, Heidelberg (2007)
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)
5. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Shao, Z., Pierce, B.C. (eds.) Principles of Programming Languages, pp. 90–101. ACM Press, New York (2009)
6. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
7. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Principles of Programming Languages, pp. 14–25. ACM Press, New York (2004)
8. Beringer, L.: Relational program logics in decomposed style (2010) (submitted)
9. Bodík, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: Principles of Programming Languages, pp. 339–352 (2010)
10. Bornat, R.: Proving pointer programs in hoare logic. In: Backhouse, R.C., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
11. Gast, H.: Lightweight separation. In: Mohamed, O., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 199–214. Springer, Heidelberg (2008)
12. Giorgino, M., Strecker, M., Matthes, R., Pantel, M.: Verification of the Schorr-Waite algorithm - From trees to graphs (January 2010)
13. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: Shao, Z., Pierce, B.C. (eds.) Principles of Programming Languages, pp. 16–28. ACM, New York (2009)
14. Hubert, T., Marché, C.: A case study of c source code verification: the schorr-waite algorithm. In: Aichernig, B., Beckert, B. (eds.) Software Engineering and Formal Methods, pp. 190–199. IEEE Computer Society, Los Alamitos (2005)

15. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: Principles of Programming Languages, pp. 14–26 (2001)
16. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Katholieke Universiteit Leuven (2008)
17. Jacobs, B., Smans, J., Piessens, F.: Verifying the composite pattern using separation logic. In: Workshop on Specification and Verification of Component-Based Systems, Challenge Problem Track (November 2008)
18. McCreight, A.: Practical tactics for separation logic. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 343–358. Springer, Heidelberg (2009)
19. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Inf. Comput. 199(1-2), 200–227 (2005)
20. Myreen, M.O.: Separation logic adapted for proofs by rewriting. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 485–489. Springer, Heidelberg (2010)
21. Nanevski, A., Vafeiadis, V., Berdine, J.: Structuring the verification of heap-manipulating programs. In: Hermenegildo, M., Palsberg, J. (eds.) Principles of Programming Languages, pp. 261–274. ACM, New York (2010)
22. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: 2011 IEEE Symposium on Security and Privacy. IEEE Computer Society, Los Alamitos (2011)
23. O'Hearn, P.W., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
25. Suzuki, N.: Automatic Verification of Programs with Complex Data Structures. PhD thesis, Stanford University (1976)
26. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) Principles of Programming Languages, pp. 97–108. ACM, New York (2007)
27. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
28. Yang, H.: Local reasoning for stateful programs. PhD thesis, University of Illinois, Urbana, IL, USA (2001)
29. Yang, H.: Relational separation logic. Theoretical Computer Science 375(1-3), 308–334 (2007)
30. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

A Dataflow Analysis to Improve SAT-Based Bounded Program Verification

Bruno Cuervo Parrino[1], Juan Pablo Galeotti[1,2], Diego Garbervetsky[1,2], and Marcelo F. Frias[2,3]

[1] Departmento de Computación, FCEyN, UBA
[2] CONICET
[3] Department of Software Engineering, ITBA
{bcuervo,jgaleotti,diegog}@dc.uba.ar, mfrias@itba.edu.ar

Abstract. SAT-based bounded verification of programs consists of the translation of the code and its annotations into a propositional formula. The formula is then analyzed for specification violations using a SAT-solver. This technique is capable of proving the absence of errors up to a given scope. SAT is a well-known NP-complete problem, whose complexity depends on the number of propositional variables occurring in the formula. Thus, reducing the number of variables in the logical representation may have a great impact on the overall analysis. We propose a dataflow analysis which infers the set of possible values that can be assigned to each local and instance variable. Unnecessary variables at the SAT level can then be safely removed by relying on the inferred values. We implemented this approach in TACO, a SAT-based verification tool. We present an extensive empirical evaluation and discuss the benefits of the proposed approach.

1 Introduction

Bounded verification [7] is a technique in which all executions of a procedure are exhaustively examined within a finite space given by a bound (a) on the domain sizes and (b) on the number of loop unrollings. The scope of analysis is examined in order to look for an execution trace that violates the provided specification.

Several bounded verifications tools [7, 10, 12, 24] rely on appropriately translating the original piece of software, as well as the specification to be verified, to a propositional formula. The use of a SAT-Solver [3] then allows us to find a valuation for the propositional variables that encodes a failure. Theoretically, SAT-solving time grows exponentially w.r.t. the number of propositional variables. However, modern SAT solvers achieve better results on practical instance problems.

In contrast to other analyses relying on theorem provers such as SMT-solvers [16], SAT-based analyses cannot be used to prove programs are correct. They only guarantee the absence of errors within the given scope. Nevertheless, SAT-based tools are better suited for finding counterexamples. By bounding the scope of analysis, these tools are able to faithfully represent the program behavior without loosing precision (i.e., no false warnings).

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 138–154, 2011.

The size of the propositional formula and its number of variables are highly related to the size and shape of the annotated program, the state representation (for Java: local, global variables and the heap) and the given scope of analysis. Therefore, techniques aiming at reducing any of these factors could possibly have a great impact on the overall verification cost.

Dataflow analysis [15] is a static analysis technique which is widely used for program understanding and optimization. Roughly speaking, it infers facts about the program by collecting the data flowing through its control flow graph (usually an abstraction of the concrete program state). Instances of dataflow analyses are: live variable analysis, available expressions, reachable definitions, constant propagation, etc. These analyses enable compilers to eliminate dead code, reduce unnecessary run-time checks, remove redundant computations, etc.

In this work, we present a novel dataflow analysis for inferring the set of possible values that can be assigned to local and instance variables, at each program point. The obtained information is a safe over-approximation of the actual value each variable may have. We apply this value-propagation analysis in the context of bounded verification where it is possible to use a fine-grained abstraction without compromising termination.

We introduced this analysis in TACO [10], a SAT-based tool specially aimed at verifying JML-annotated [6] Java sequential programs. TACO accurately represents all Java data types (including primitive types such as double and float) and supports nearly all JML syntax. TACO does not report false bugs but is not able to prove the absence of errors above the given scope of analysis. Among its features, TACO introduces a novel technique for removing unnecessary propositional variables at the SAT level. This is accomplished by preprocessing class invariants in order to obtain a good over-approximation of the initial state of the Java memory heap. This set of initial values can be supplied to our dataflow analysis, obtaining a more accurate set of possible values for every program variable. In turn, this information leads to a more aggressive removal of unnecessary propositional variables at the SAT level.

TACO's previous representation was implemented as a simple sequence of `if` statements. This representation introduced at least one join point per loop unrolling which impacted negatively in the precision of the overall dataflow analysis. In this work we introduce an alternative representation for loop unrollings tailored to favor precision of the dataflow analysis.

Our experiments show significant speed-ups in analysis times: about 30 times reduction in average. Surprisingly, the proposed loop encoding has a significant impact in the overall verification time.

Contributions: The technical contributions of this article include:

- A formalization of a dataflow analysis for propagating values through a program control flow graph, including a proof sketch showing that the outcome of the analysis is a sound over-approximation of the program behavior.
- A proof of the fact that the propositional formula obtained by TACO relying in this analysis is equisatisfiable w.r.t. the unoptimized formula.

```
class Node { Node next; }
class List { Node header;
  /*@ invariant (\forall Node n ;\reach(this.header,Node,next).has(n);
    @   !\reach(n.next,Node,next).has(n));
    @*/
}
```

Fig. 1. A singly linked list declaration

- TACO-Flow: an extension of TACO featuring a general dataflow framework including proper generation of control flow graphs, the (bounded) value-propagation analysis and the generation of the optimized SAT-formula.
- An empirical evaluation using benchmarks accepted by the bounded verification community [10] showing an important speed-up in verification time.

Related work: There is plenty of work aiming at improving SAT-based program verification. The most remarkable examples are the approaches implemented in F-Soft [12], Saturn [24], TACO [10] and JForge [7]. Here we will focus only on related work concerning the use of dataflow analysis to alleviate the task of the SAT-solver. For a comprehensive discussion of these tools please refer to [9].

The idea of using dataflow analysis in the context of SAT-based program verification is not new. F-Soft [12] performs a dataflow analysis to compute ranges for values of integer-valued variables and pointers, under the hypothesis that runs have bounded lengths. Saturn [24] compresses formulas using several optimizations (e.g., program slicing) and provides means for specifying analyses aimed at producing method summaries. In [22], the authors proposed an analysis to infer method summaries and enable modular verification. JForge [7] uses a dataflow analysis to find and eliminate logically infeasible branches. In [18] the authors propose a technique based on dataflow analysis (variable-definitions) to split the SAT-problem into several simpler sub-problems.

Outline: §2 introduces the foundations of SAT-based verification and TACO, then it presents the problem we intend to tackle in the current work. §3 presents the value-propagation analysis. §4 shows how this technique is applied in the context of TACO. §5 shows our experimental results and, finally, §6 concludes and discusses future work.

2 Tight Bounds for Improved SAT-Solving

Fig. 1 shows a JML declaration of a singly linked list data structure. It contains a `header` field referring to its first node. Each node links to its next node in the list by the `next` field. The List container is annotated with a JML object invariant which constraints the set of valid linked structures to those who form a finite acyclic sequence of Node elements. The construct `\reach(l,T,f)` denotes the set of objects of type T reachable from a location l using field f.

Method `removeLast` (shown in Fig. 3a) removes the last element of the list (if such an element exists). JML allows one to write a partial specification. In

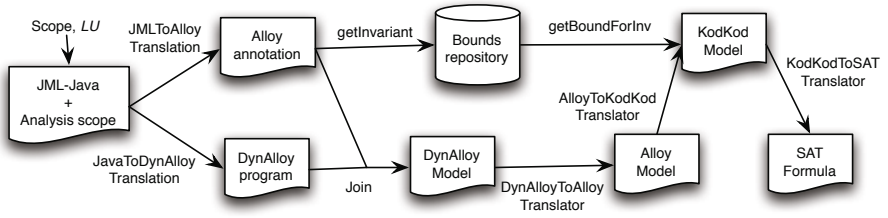


Fig. 2. Translating annotated code to SAT

this example, the `ensures` clause only specifies that the returning Node element should not be reachable from the receiver list.

As shown in Fig. 2, TACO translates the Java code annotated with the JML [6] contract into a DynAlloy specification [8]. DynAlloy is a relational specification language. In other words, every variable in DynAlloy can be seen as a relation of a fixed arity. For the `removeLast` method, the resulting DynAlloy program is shown in Fig. 3b. Signatures *List* and *Node* are introduced to model the Java classes. Also, a singleton signature *null* is defined to model the Java *null* value. **Java variables and fields** are represented using **DynAlloy variables**. Java fields are modelled in DynAlloy as functional binary relations (i.e., `S->one T`), and Java variables are modelled as unary non-empty relations (i.e., `one S`). The following DynAlloy variables are also introduced to model removeLast's Java variables and fields:

```
header: List -> one (Node+null)   prev:   one Node+null
next:   Node -> one (Node+null)   curr:   one Node+null
this:   one List                  return: one Node+null
```

```
/*@ ensures
  @  !\reach(this.header,
  @   Node,next).has(\result);
  @*/
Node removeLast() {
  if (this.header!=null) {
    Node prev = null;
    Node curr = this.header;
    while (curr.next!=null) {
      prev = curr;
      curr = curr.next;
    }
    if (prev==null)
      this.header = null;
    else
      prev.next = null;

    return curr;
  } else
    return null;
}
```

(a) A `removeLast()` method

```
(this.header!=null)?;{
  prev := null;
  curr := this.header;
  {
    (curr.next!=null)?;
    prev = curr;
    curr = curr.next
  }*;
  ((curr.next==null)?;
  (prev==null)?;
    this.header := null
  +
  (prev!=null)?;
    prev.next := null;
  );
  return := curr;
}+
(this.header==null)?;
  return := null
```

(b) The DynAlloy representation

Fig. 3. Java implementation and its DynAlloy representation

If the user wants to perform a bounded verification of `removeLast`'s contract, she must limit the object domains and the number of loop iterations. Let us assume that a scope of at most 5 Node objects, 1 List object and 3 loop unrolls is chosen. The DynAlloy specification is then translated to an Alloy specification as described in [8]. In order to model state change in Alloy, the *DynAlloyToAlloyTranslator* may introduce several **Alloy relations** to represent different values (or *incarnations*) of the same DynAlloy variable in an SSA-like form [5].

In order to translate the Alloy specification into a SAT-problem, the Alloy Analyzer focuses on translating every Alloy relation into a set of **propositional variables**. Each propositional variable is intended to model that a given tuple is contained in the Alloy relation. In the example, as the *Node* domain is restricted to 5 elements, $\{N_1, \ldots N_5\}$ is the set of 5 available Node atoms. This leads to the the following propositional variables modeling the binary relation $next_0$ (which, in turn, models the initial state of the DynAlloy variable `next`):

M_{next_0}	N_1	N_2	N_3	N_4	N_5	`null`
N_1	p_{N_1,N_1}	p_{N_1,N_2}	p_{N_1,N_3}	p_{N_1,N_4}	p_{N_1,N_5}	$p_{N_1,null}$
N_2	p_{N_2,N_1}	p_{N_2,N_2}	p_{N_2,N_3}	p_{N_2,N_4}	p_{N_2,N_5}	$p_{N_2,null}$
N_3	p_{N_3,N_1}	p_{N_3,N_2}	p_{N_3,N_3}	p_{N_3,N_4}	p_{N_3,N_5}	$p_{N_3,null}$
N_4	p_{N_4,N_1}	p_{N_4,N_2}	p_{N_4,N_3}	p_{N_4,N_4}	p_{N_4,N_5}	$p_{N_4,null}$
N_5	p_{N_5,N_1}	p_{N_5,N_2}	p_{N_5,N_3}	p_{N_5,N_4}	p_{N_5,N_5}	$p_{N_5,null}$

Following this representation, propositional variable p_{N_3,N_2} is true if and only if tuple $\langle N_3, N_2 \rangle$ is contained in the Alloy relation $next_0$. Given the selected scope of analysis, if no pre-processing is involved, the resulting SAT-problem will contain 126 propositional variables. Only 36 (28%) variables model the initial Java state, that is the representation of the receiver object instances. The remaining 90 (72%) variables are introduced to represent the intermediate and final stages for computing the `removeLast` method. That is, to model the state evolution during the execution of the method body.

Alloy uses KodKod [21] as an intermediate language, which is then translated to a CNF propositional formula (Fig. 2 sketches the translations involved). KodKod allows the prescription of bounds for Alloy relations. For each relation f, two relational instances L_f (the lower bound) and U_f (the upper bound) are attached. In any Alloy model I, f (the interpretation of relation f in model I), must satisfy $L_f \subseteq \mathsf{f} \subseteq U_f$. Therefore, pairs that are in L_f must necessarily belong to f, and pairs that are not in U_f cannot belong to f. If tuples are removed from an upper bound, the resulting upper bound is said to be *tighter* than before.

Tighter upper bounds contribute by removing propositional variables. Given an Alloy relation f, propositional variables corresponding to tuples that do not belong to U_f can be directly replaced in the translation process with the truth value *false*. This allows us to reduce the number of propositional variables.

TACO preprocesses class invariants and automatically computes a tight upper bound for the initial state of Java class fields. As shown in Fig. 2, bounds are stored in a repository. Since bounds are often reused during the analysis of different methods in a class, the cost of computing the bounds is amortized.

This preprocessing allowed TACO to remove (in the presented example) over 70% of the propositional variables representing the initial state.

2.1 Problem Statement

The technique introduced in [10] limits itself to bound those propositional variables which represent the *initial* state of the program under analysis. One may argue that, as the SAT-solver is not able to recognize the order in which the program control flows, there is no guarantee that the SAT-solving process will avoid partial valuations from intermediate states that could not lead to a valid computation trace.

Dataflow analysis allows us to collect facts about the program behaviour at various points in a program. For instance, we could conclude that (under the scope of analysis previously chosen) for the following statement: `Node curr = this.header;` the set of values that are assignable to `curr` is $\{null, N_1\}$.

In this work we propose a dataflow analysis to over-approximate the set of possible values every Java variable and field may store within the provided scope of analysis. Using this conservative analysis, we can propagate the upper bounds from the initial state to the intermediate states. We believe the resulting tighter upper bounds for the intermediate Alloy relations should contribute by allowing KodKod to remove propositional variables. For instance, for the presented example we are able to remove about 58% of the propositional variables modeling intermediate stages. As we will see in §5, this technique leads to a potential improvement in the performance of the SAT-based verification.

3 Propagating Values in DynAlloy Programs

DynAlloy is based on first-order dynamic logic [11]. The aim of this specification language is to provide a formal characterization of imperative sequential programs. Fig. 4 shows a relevant fragment of DynAlloy's grammar. This fragment corresponds to the DynAlloy programs output by the *JavaToDynAlloy* translator in TACO. As shown in Fig. 3b, typical structured programming constructs can be described using these basic logical constructs. Given a DynAlloy program, our dataflow analysis computes an over approximation of all the possible variable assignments for every program location. We chose DynAlloy as a platform for the dataflow analysis because: 1) it is closer to the SAT-problem than the Java representation, and 2) it is the last intermediate representation in the TACO pipeline where a notion of control flow and state change still remains. In other words, DynAlloy contains the last imperative representation of the code under analysis.

Concrete Semantics: We begin by defining a concrete semantics for the execution of this DynAlloy fragment which mimics the execution of Java programs. As the DynAlloy relational semantics is interpreted in terms of atoms, *Atom* represents the set of all atoms in this interpretation. We denote by $JVar \uplus JField$ the set of DynAlloy variables. A DynAlloy variable belonging to $JVar$ corresponds

$$
\begin{array}{rll}
program ::= & v := expr & \text{“copy”} \\
| & v.f := expr & \text{“store”} \\
| & skip & \text{“skip action”} \\
| & formula? & \text{“test”} \\
| & program + program & \text{“non-deterministic choice”} \\
| & program; program & \text{“sequential composition”} \\
| & program^* & \text{“iteration”} \\
| & \langle program \rangle(\overline{x}) & \text{“invoke program”}
\end{array}
$$

$$expr ::= \textbf{null} \mid v \mid v.f$$

Fig. 4. Relevant DynAlloy fragment

to the representation of a Java variable and its concrete value is a single atom. Similarly, a variable belonging to $JField$ models a Java field whose concrete value is a mapping (functional relation) from atoms to atoms. A *concrete* state $c \in E$ maps each DynAlloy variable to a concrete value.

$$E = JVar \uplus JField \rightarrow Atom \cup \mathcal{P}(Atom \times Atom)$$

We denote by $M[\phi]_c$ the truth value for formula ϕ at the concrete state c. Similarly, we denote by $X[expr]_c$ the value of expression $expr$ in the concrete state c. The value of $X[expr]_c$ for the DynAlloy expressions that we will consider could be defined as follows:

$$
\begin{array}{ll}
X[null]_c & = \{\langle null \rangle\} \\
X[v]_c & = \{c(v)\} \\
X[v.f]_c & = c(v); c(f)
\end{array}
$$

where the composition of relations R (arity i) and S (arity j) is defined as follows:

$$R; S = \{\langle a_1, \ldots, a_{i-1}, b_2, \ldots, b_j \rangle : \exists b(\langle a_1, \ldots, a_{i-1}, b \rangle \in R \wedge \langle b, b_2, \ldots, b_j \rangle \in S)$$

$R \rightarrow S$ denotes the Cartesian product between relations R and S. $R + +S$ denotes the relational overriding defined as follows[1]:

$$R + +S = \{\langle a_1, \ldots, a_n \rangle : \langle a_1, \ldots, a_n \rangle \in R \wedge a_1 \notin dom(S)\} \cup S$$

DynAlloy's relational semantics is given in [8]. Here we present an alternative definition based on the collecting semantics [17] which is useful for proving the correctness of our proposed dataflow technique. A collecting semantics defines how information flows through a program control flow graph (CFG). Given a DynAlloy program P it is possible to obtain its CFG. The CFG describes the structure of the program. In the collecting semantics, every time a new value traverses a node it is recorded. Therefore, each node keeps track of all values that passed through it.

[1] Given a n-ary relation R, $dom(R)$ denotes the set $\{a_1 : \exists a_2, \ldots, a_n$ such that $\langle a_1, a_2, \ldots a_n \rangle \in R\}$.

Definition 1. *Given a DynAlloy program P and a formula ϕ representing input states, the collecting semantics of P starting with state ϕ is the least fix point (LFP) of the following equations:*

For each node n in the CFG of P:

$$in(n) = \begin{cases} \{c_0 | M[\phi]_{c_0}\} & n \text{ is the entry of } CFG(P) \\ \bigcup_{p \in pred(n)} out(p) & otherwise \end{cases}$$
$$out(n) = \mathcal{F}_c(n, in(n))$$

where $in(n)$, $out(n)$ denote the input and output values of node n and $pred(n)$ the set of predecessors of n.

The transfer function $\mathcal{F}_c : DynAlloyProgram \times C \to C$ with $C \in \mathcal{P}(E)$ models how the concrete state changes as a DynAlloy statement is executed. The definition is the following:

$$\begin{aligned} \mathcal{F}_c(skip, cs) &= cs \\ \mathcal{F}_c(\phi?, cs) &= \{c | c \in cs \wedge M[\phi]_c\} \\ \mathcal{F}_c(v := expr, cs) &= \{c[v \mapsto X[expr]_c] | c \in cs\} \\ \mathcal{F}_c(v.f := expr, cs) &= \{c[f \mapsto c(f) + +(c(v) \to X[expr]_c)] | c \in cs\} \end{aligned}$$

Abstract Semantics: We represent an *abstract* state by mapping each DynAlloy variable to its corresponding abstract value. The abstract value of a DynAlloy variable modelling a Java field is a (probably non-functional) binary relation from atoms to atoms. A DynAlloy variable representing a Java variable maps to a set of atoms.

$$A = JVar \uplus JField \to \mathcal{P}(Atom) \cup \mathcal{P}(Atom \times Atom)$$

An abstract value represents the set of all concrete values a DynAlloy variable *may* have (i.e., an over-approximation) in a given program location. In order to operate with this abstraction we need A to be a lattice. Let $\langle A, \sqsubseteq \rangle$ such that for all $a, a' \in A$:

- $a \sqsubseteq a'$ iff $\forall x \in JVar \uplus JField(a(x) \subseteq a'(x))$,
- $a \sqcup a' = a''$ such that $\forall x \in JVar \uplus JField(a''(x) = a(x) \cup a'(x))$

The abstraction function $\alpha : E \to A$ formalizes the notion of approximation of a concrete state by an abstract state. Given a concrete state $c \in E$:

$$\alpha(c) = a \text{ s.t. } \forall v \in JVar(a(v) = \{c(v)\}) \wedge \forall f \in JField(a(f) = c(f))$$

Notice that the chosen abstract domain is indeed very similar to the concrete domain. The main difference is that an abstract value can represent several concrete values (i.e., the powerset of $Atom$). Several concrete values are merged into a single abstract value after a join point in the CFG.

The concretization function $\gamma : A \to C$ is defined as: $\gamma(a) = \{c \mid \alpha(c) \sqsubseteq a\}$. Let X_α be the abstract evaluation function which is identical to X except that $a(v)$ directly returns a set. The abstract transfer function $\mathcal{F} : DynAlloyProgram \times A \to A$ is defined by the following set of rules:

- $\mathcal{F}(skip, a) = a$
- $\mathcal{F}(\phi?, a) = a$
- $\mathcal{F}(v := expr, a) = a[v \mapsto X_\alpha[expr]_a]$
- $\mathcal{F}(v.f := expr, a) =$ **let** $from = a(v)$, $to = X_\alpha[expr]_a$ **in**
  if $|from| = 1$
  then $a[f \mapsto a(f) + +(from \rightarrow to)]$ (strong update)
  else $a[f \mapsto a(f) \cup (from \rightarrow to)]$ (weak update)

Notice that the semantics of the store operation distinguishes two cases: 1) the abstraction is precise enough to perform an update of a unique source, 2) an over-approximated step must be taken. Due to space limitations we do not include the dataflow equations for the analysis. It is essentially equal to the collecting semantics but using the $\sqcup$ operator to merge states.

Correctness: Here we show that the abstraction is a sound over-approximation of the collecting semantics.

Theorem 1. *Let* $cs \in C$, $a \in A$, $n \in CFG(P)$,

$$\alpha(cs) \sqsubseteq a \Rightarrow \alpha(\mathcal{F}_c(n, cs)) \sqsubseteq \mathcal{F}(n, a)$$

Proof sketch: It can be proved for each statement separately. The proof for skip and test actions is trivial. The only case that requires some care is the store operation. It follows directly from the definitions of α, $\mathcal{F}_c(n, cs)$, and $\mathcal{F}(n, \alpha)$ (see a complete proof in the companion report [4]).

Corollary 1. *For each node* $n \in CFG(P)$*, the LFP of the dataflow analysis equations is an over approximation of the LFP of the corresponding equations in the collecting semantics.*

Termination: It follows trivially from the fact that a finite *Atom* set leads to a finite lattice (both the concrete and abstract domains are finite).

4 Effective Removal of Variables Using Dataflow Analysis

We now present the mechanism to effectively remove propositional variables in the SAT-formula. As previously mentioned, TACO removes propositional variables by introducing *tighter* upper bounds for those Alloy relations representing the initial Java memory heap. KodKod allows one to prescribe bounds for Alloy relations of any arity (unary relations included). The *DynAlloyToAlloy* translator introduces several versions of the same DynAlloy variable in order to model state change in Alloy. Our goal is to compute a tighter upper bound for each Alloy relation modelling different versions of the same DynAlloy variable.

The execution of the *DynAlloyToAlloy* translator is separated into several phases. Each phase performs a semantic preserving transformation of the DynAlloy specification. The following phases are executed in an orderly fashion:

1. **Unroll:** Removes loops by unrolling them up to the provided limit.
2. **Inline:** Replaces program invocation with the corresponding method bodies.
3. **SSA:** Applies an SSA-like transformation of the DynAlloy program.
4. **NoLocals:** Promotes local variables to program parameters.

The resulting DynAlloy specification is then translated into an Alloy representation following the rules presented in [8]. Once the single static assignment (SSA) transformation is applied, DynAlloy variables and Alloy relations match. Therefore, if we perform our value-propagation analysis on this final DynAlloy representation, we will obtain an over approximation of all possible values an Alloy relation may have.

By default, Alloy associates a conservative upper bound for each relation which was not explicitly bounded. If an upper bound is found in the repository, TACO instruments the Alloy representation by including the stored upper bound, refining the initial state. It is worth mentioning that, as the upper bounds stored in the repository can be seen as an over approximation of the values of the initial Java memory heap, we use them as a refined entry abstraction for our dataflow analysis.

The refined entry abstraction is then completed for the remaining variables depending on the intended meaning of the DynAlloy variable. For those variables modelling Java parameters, all tuples that satisfy the type definition within the scope of analysis are associated. For any other DynAlloy variable x, no tuple is associated (i.e., $a(x) = \emptyset$).

In order to properly introduce tighter bounds for all Alloy relations, we inspect the abstract value of each DynAlloy variable at the *exit* location. Given a DynAlloy variable $x \in JVar \uplus JField$, the abstract value for x at this location could be written as an upper bound of (the Alloy relation) x and fed to the KodKod input, leading to the removal of unnecessary propositional variables. For cases where x's abstract value maps to an empty set, a special measure has to be taken in order to enforce Alloy's relational constraints.

Definition 2. *Let a_{exit} be the computed abstract value for the exit of the CFG.*

$$U_x^{DF} = \begin{cases} a_{exit}(x) & \text{if } a_{exit}(x) \neq \emptyset \\ defVal(x) & \text{otherwise} \end{cases}$$

where $defVal(x)$ returns the default Java values (e.g, 0 for `Integer`*, false for* `Bool`*, etc) in case x models a Java variable, and a total function whose range contains default values in case x represents a Java field.*

We call U_x^{TACO} to the upper bound fed by TACO to KodKod when no dataflow analysis is performed. The following theorem ensures that the technique is safe (i.e., it does not miss faults).

Theorem 2. *Let θ be the Alloy formula output by the* DynAlloyToAlloy *translator. Given an Alloy model I such that $M[\theta \wedge \bigwedge_x (x \subseteq U_x^{TACO})]_I =$ **true***

*Then, there is an Alloy model I' such that $M[\theta \wedge \bigwedge_x (x \subseteq U_x^{DF})]_{I'} =$ **true***

Proof. Let I be an Alloy model satisfying the hypothesis. We know by definition that $defVal(x) \subseteq U_x^{DF}$. Let x be an Alloy relation such that $I(x) \subseteq U_x^{TACO} \setminus U_x^{DF}$. Due to Corollary 1, x must match a DynAlloy variable which simultaneously satisfies that: 1) x represents a local variable, and 2) no assignment nor access to x occurs within the set of traces codified by I. Therefore, x's value has no effect on the set of traces codified by I. This means that this set remains unchanged if we replace x's value with any other value (e.g. $defVal(x)$). Due to the fact that θ encodes a partial correctness assertion [8], the satisfiability of θ does not depend on x's value. Thus, $M[\theta]_I = M[\theta]_{I[x \mapsto defVal(x)]}$ by substitution. Therefore, we can define I' as:

$$I'(x) = \begin{cases} I(x) & \text{if } I(x) \subseteq U_x^{DF} \\ defVal(x) & \text{otherwise} \end{cases}$$

where $M[\theta]_I = M[\theta]_{I'}$ and $I'(x) \subseteq U_x^{DF}$ □.

4.1 Loop Optimization

The Java while construct `while` B `do` P `od` can be expressed in DynAlloy as $(B?; P)^*; (\neg B)?$. Given a loop limit of k, the **Unroll** phase transforms the loop into:

$$\underbrace{((B?; P) + skip); \ldots; ((B?; P) + skip)}_{k-times}; (\neg B)?$$

Although semantically correct, this representation of the while construct permits several permutations. For instance: the program trace $B?; P; skip$ is equivalent to $skip; B?; P$. This apparently harmless symmetry has a tremendous impact since our dataflow analysis is branch insensitive. Due to this, the computed over approximation becomes too coarse.

This observation led us to modify the **Unroll** phase. The new nested unrolling encodes `while` B `do` P `od` into $T_k(B, P); (\neg B)?$, where T is recursively defined as:

$$T_0(B, P) = skip$$
$$T_n(B, P) = (((B?; P); T_{n-1}(B, P)) + skip)$$

5 Empirical Evaluation

In this section we present the experimental evaluation we performed in order to validate our approach. We aim at answering the following two research questions:

RQ1: Is our approach capable of outperforming the current SAT-based analyses?
RQ2: Where do the performance gains come from?

In order to answer these questions we implemented TACO-Flow. TACO-Flow[2] is an extension of TACO which implements the approach described in §4. That is, a new encoding of loop unrolls, a generic dataflow framework for DynAlloy programs, our value-propagation analysis as an instance of this framework, and finally, its application as a means to remove propositional variables in the Alloy intermediate representation.

[2] TACO-Flow and the benchmarks are available at `http://www.dc.uba.ar/tacoflow`

Table 1. Analysis times (in seconds) for TACO and TACO-Flow, speed-up and variables in the obtained propositional formula

Method	Analysis Times (secs.)				Variables	
	TACO	T-Flow	Dataflow	Speed-up	# TACO	Reduction
SList.contains.s20	8.25	5.34	0.13	1.54	1743	2.70%
SList.insert.s20	9.91	11.30	0.41	0.88	3892	11%
SList.remove.s20	18.11	8.20	0.12	2.21	3749	15.92%
AList.contains.s20	29.20	8.43	0.19	3.46	3573	34.73
AList.insert.s20	10.66	9.04	0.14	1.18	4732	0.97%
AList.remove.s20	144.35	11.94	0.17	12.09	4580	11.55%
CList.contains.s20	109.7	47.53	0.16	2.31	2530	28.74%
CList.insert.s20	59.9	45.82	0.18	1.31	4512	30.78%
CList.remove.s20	1649.47	353.66	0.33	4.66	5365	11.39%
AvlTree.find s20	25.82	25.28	0.14	1.02	1412	5.95%
AvlTree.findMax.s20	1439.32	119.03	1.96	12.09	2505	2.95%
AvlTree.insert.s17	32018.14	20744.99	98.78	1.54	204799	30.33%
BinHeap.findMin.s20	109.86	10.45	0.85	10.51	2224	9.80%
BinHeap.decK.s18	18216.79	335.42	2.63	54.31	9469	1.16%
BinHeap.insert.s17	25254.66	122.22	8.52	206.63	33477	28.89%
BinHeap.extMin.s20	1149.71	451.19	21.71	2.55	55188	27.55%
TreeSet.find.s20	14475.49	769.64	1.74	18.81	3032	2.51%
TreeSet.insert.s13	26447.76	719.78	35.12	36.74	38822	1.26%
BSTree.contains.s13	28602.33	9190.95	0.19	3.11	648	0.15%
BSTree.insert.s12	7251.39	14322.00	0.44	0.51	2396	13.28%
BSTree.remove.s09	18344.38	1214.49	1.79	15.10	13369	5.42%

We considered the benchmarks presented in [10] and compare the analysis times of TACO and TACO-Flow. We analized the following case studies: **LList**: An implementation of sequences based on singly linked lists; **AList**: The implementation `AbstractLinkedList` of interface `List` from the Apache package commons.collections, based on circular doubly-linked lists; **CList**: The implementation `NodeCachingLinkedList` of interface `List` from the Apache package commons.collections; **BSTree:** A binary search tree implementation from Visser et al. [23]; **TreeSet**: The implementation of class `TreeSet` from package `java.util`, based on red-black trees; **AVLTree**: An implementation of AVL trees obtained from the case study used in [2]; **BHeap**: An implementation of binomial heaps used as part of a benchmark in [23]; For each class we consider the most representative set of methods featuring insertion, deletion, and look-up. All methods are correct with respect to their contracts except for BHeap.extMin which contains an actual fault discovered in [10].

We analyzed mainly correct implementations since we are interested in measuring the worst case scenario for bounded-verification (i.e., search space exhaustion). The case of BHeap.extMin is included to show the analysis does not miss bugs that are catchable in the given scope.

Both TACO and TACO-Flow were fed with an initial set of upper-bounds. These upper-bounds were discovered using a cluster of computers as reported in our previous work [10]. TACO-Flows used them to produce an entry abstraction for the value-propagation analysis.

We were interested in assessing the impact of the techniques in terms of analysis time and in seeing if the overhead introduced by the dataflow analysis can be compensated by the obtained performance gains.

Hardware and Software platform: All experiments were run on an Intel Core i5-570 processor running at 2.67GHz and 8GB DDR3 total main memory, on a Debian's GNU/Linux v6 operating system.

For every case study we checked that their class invariants are preserved and their method contracts are satisfied. For each method we selected the greatest scope that TACO could verify within a given time threshold (10 hours). The maximum scope is restricted to at most 20 node elements for each experiment. This is due to the fact that this is the greatest scope used for evaluating TACO in our most recent work. If loops are found, they were unrolled up to 10 times. Table 1 shows the end-to-end analysis times using both TACO and TACO-Flow, the cost of the dataflow analysis in TACO-Flow and its speed-up (the ratio TACO/TACO-Flow). The last two columns show the number of propositional variables of the SAT-formula produced by TACO and the percentage of reduction introduced by TACO-Flow.

Notice that the overall speed-up was very significant in almost all cases. More specifically, it was approximately 20 times faster in average. Only two methods exhibited a loss in performance. The required time to compute the dataflow analysis was, in general, negligible and compensated by the speed-up obtained in the end-to-end execution.

We strongly believe that the exhibited speed-up could also lead to an increase in scalability. For instance, the maximum scope that TACO successfully analyzed within the time threshold for method insert in class BinHeap was 17 node elements. In contrast, TACO-Flow was able to analyze the same scope with a speed-up of 8.52x and it easily analyzed the same method up to the maximum scope for which we had initial upper-bounds. More experimental validation is required in order to justify this claim.

Our research was driven by the hypothesis that verification times were sensitive to a decrease in the number of propositional variables. To validate that hypothesis, we collected the number of propositional variables generated by both TACO and TACO-Flow. TACO-Flow indeed reduced the number of propositional variables and the obtained performance gains appeared to confirm the hypothesis. As the reduction percentage did not seem to be directly related to the gain proportion, a further investigation of this matter is necessary.

TACO-Flow differs from TACO in the introduction of a new encoding for loop unrollings (hereinafter denoted as TACO$^+$) and the removal of propositional variables based on the dataflow analysis output. We decided to measure each contribution separately (Tables 2 and 3).

Surprisingly, TACO$^+$ showed an impressive improvement in the analysis time. We conjecture this is because the new encoding avoids a significant number of paths in the CFG leading to isomorphic valuations in the SAT-formula. For instance, for a CFG of only one loop, the application of n loop unrollings in TACO leads to 2^n paths whereas the same application in TACO$^+$ leads to $2(n-1)$ potential paths (see Fig. 5). Even tough this result was not initially expected, it is actually a consequence of the introduction of the dataflow analysis in TACO-Flow which needs a better encoding of loop unrolls to mitigate precision loss.

Table 2. TACO$^+$ speed-up

Method	TACO vs TACO$^+$
SList.contains	1.64
SList.insert	0.64
SList.remove	1.80
AList.contains	2.86
AList.insert	1.15
AList.remove	14.10
CList.contains	1.47
CList.insert	0.69
CList.remove	3.89
AvlTree.find	0.97
AvlTree.findMax	12.72
AvlTree.insert	1.09
BinHeap.findMin	13.99
BinHeap.decK	76.94
BinHeap.insert	178.49
BinHeap.extMin	2.35
TreeSet.find	13.05
TreeSet.insert	31.24
BSTree.contains	4.12
BSTree.insert	0.60
BSTree.remove	18.26

Table 3. TACO-Flow speed-up

Method	TACO$^+$ vs TACO-Flow
SList.contains	0.94
SList.insert	1.38
SList.remove	1.23
AList.contains	1.21
AList.insert	1.03
AList.remove	0.86
CList.contains	1.57
CList.insert	1.89
CList.remove	1.20
AvlTree.find	1.06
AvlTree.findMax	0.95
AvlTree.insert	1.42
BinHeap.findMin	0.75
BinHeap.decK	1.09
BinHeap.insert	1.15
BinHeap.extMin	1.08
TreeSet.find	1.44
TreeSet.insert	0.90
BSTree.contains	0.76
BSTree.insert	0.84
BSTree.remove	1.33

Now, we focus on Table 3. For every method we took the maximum scope that TACO$^+$ could analyze and ran TACO-Flow on the same setting. It is worth noticing that, even when the improvements are less impressive than those shown in Table 2, this rather simple dataflow analysis is able to obtain significant gains. For instance, in some cases it is about 90% with approximately 16% on average.

Unfortunately, there are a few cases where some performance loss is reported. At first glance, these cases contradict our initial hypothesis accounting that a reduction in the number of propositional variables leads to performance improvement. Tighter upper-bounds are not only used by KodKod to remove propositional variables. KodKod computes a symmetry breaking predicate (SBP) based on the provided lower and upper-bounds. This SBP reduces many (but often not all) isomorphic valuations (i.e., symmetries). Our guess is that the introduction of our tighter upper bounds for the intermediate states is degrading the SBP that KodKod produces (this reasoning may also apply to the two benchmarks with no speed-ups in Table 1). This effect could be avoided by injecting the tighter upper-bounds directly at the SAT level without affecting KodKod. Although this is a clear research direction to follow, this goes beyond the proposed scope of this article.

Threats to validity: A first concern is related with the fact that we are empirically comparing the proposed approach only against our own previous work. Even though this concern is valid, we would like to point out that TACO was recently compared against several state-of-the-art SAT-based, model checkers and SMT-based verification tools [10].

A second concern is about how representative the benchmarks are. In this regard, the benchmarks we have chosen appear recurrently in case studies used by the bounded verification community [1, 7, 14, 19, 23]. In addition, the algorithms

(a) TACO's loop unroll encoding **(b)** New encoding of loop unrollings

Fig. 5. Loop unroll encodings in TACO and TACO-Flow

found in the case studies are commonplace. They recurrently appear in many applications [20] ranging from container classes to XML parsers. Therefore, even if it is not possible to perform general claims about all applications, they can be used as a relative measure of how well the proposed approach performs compared with other tools aiming at verifying heap manipulating algorithms.

Another threat to validity is the length of these benchmarks. They target code manipulating rather complex data structures, working at the intraprocedural level. In the presence of contracts for methods, modular SAT-based analysis could be applied by replacing method calls by their corresponding contracts and then analyzing the resulting code. This approach is followed for instance in [7].

Finally, TACO-Flow relies on having a pre-computed set of initial upper-bounds. The distributed computation cost of this artifact is significant with respect to the sequential analysis time. Nevertheless, as already mentioned, this computational cost can be amortized along time.

6 Conclusions and Further Work

In this article we presented a value-propagation analysis aiming at reducing SAT-solving verification costs. Applying this technique required the implementation of a dataflow framework in TACO. As a means to mitigate precision loss we introduced a new encoding for loops. This had an unexpected positive impact in the overall performance.

In summary, the whole approach led to an important increase of performance in the whole verification process. More experimentation is required to assess with confidence whereas the approach is capable of increasing the scope of analysis beyond the current state-of-the-art.

We strongly believe there is still room for reducing verification cost by relying on dataflow analyses. For instance, an alias analysis can be used to rule-out infeasible valuations. We are currently implementing this analysis using our framework. Initial results seems to be promising. We also want to check our conjecture about KodKod's symmetry breaking predicate. The current TACO prototype removes variables at the Alloy level. We plan to develop a new prototype that could remove them at the SAT-level. We believe that this will mitigate the observed performance loss.

References

1. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: ISSTA 2002, pp. 123–133 (2002)
2. Belt J., Robby, Deng X.: Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses. In: FSE 2009, pp. 355–364 (2009)
3. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185 (2009)
4. Cuervo Parrino, B., Galeotti, J.P., Garbervetsky, D., Frias, M.: A dataflow analysis to improve SAT-based program verification, Technical Report (May 2011), http://www.dc.uba.ar/tacoflow/techrep.pdf
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM TOPLAS 13(4), 451–490
6. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
7. Dennis, G., Yessenov, K., Jackson, D.: Bounded Verification of Voting Software. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 130–145. Springer, Heidelberg (2008)
8. Frias, M.F., Galeotti, J.P., Lopez Pombo, C.G., Aguirre, N.: DynAlloy: Upgrading Alloy with Actions. In: ICSE 2005, pp. 442–450 (2005)
9. Galeotti, J.: Software Verification Using Alloy. PhD. Thesis, UBA (2011)
10. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Proceedings of ISSTA 2010, pp. 25–36 (2010)
11. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. Foundations of Computing. MIT Press, Cambridge (2000)
12. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F-Soft: Software Verification Platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
13. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2006)
14. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA 2000, pp. 14–25 (2000)
15. Kindall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206 (1973)
16. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. Nielson, F.: A denotational framework for data flow analysis. Acta Inf. 18, 265–287 (1982)
18. Shao, D., Gopinath, D., Khurshid, S., Perry, D.E.: Optimizing Incremental Scope-Bounded Checking with Data-Flow Analysis. In: ISSRE 2010, pp. 408–417 (2010)
19. Sharma, R., Gligoric, M., Arcuri, A., Fraser, G., Marinov, D.: Testing Container Classes: Random or Systematic? In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 262–277. Springer, Heidelberg (2011)
20. Siddiqui, J.H., Khurshid, S.: An Empirical Study of Structural Constraint Solving Techniques. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 88–106. Springer, Heidelberg (2009)

21. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
22. Taghdiri, M., Seater, R., Jackson, D.: Lightweight extraction of Syntactic Specifications. In: FSE 2006, pp. 276–286 (2006)
23. Visser, W., Păsăreanu, C.S., Pelánek, R.: Test Input Generation for Java Containers using State Matching. In: ISSTA 2006, pp. 37–48 (2006)
24. Xie, Y., Aiken, A.: Saturn: A scalable framework for error detection using Boolean satisfiability. ACM TOPLAS 29(3) (2007)

Reverse Hoare Logic*

Edsko de Vries and Vasileios Koutavas

Trinity College Dublin, Ireland
{Edsko.de.Vries,Vasileios.Koutavas}@cs.tcd.ie

Abstract. We present a novel Hoare-style logic, called Reverse Hoare Logic, which can be used to reason about state reachability of imperative programs. This enables us to give natural specifications to randomized (deterministic or nondeterministic) algorithms. We give a proof system for the logic and use this to give simple formal proofs for a number of illustrative examples. We define a weakest postcondition calculus and use this to show that the proof system is sound and complete.

1 Introduction

Hoare Logic [12] is a popular method for proving properties of imperative programs. The following specification for sort is a classical example:

$$\{\top\}\ \mathsf{sort}(\mathtt{a})\ \{\mathrm{sorted}(\mathtt{a}) \wedge \mathtt{a} \in \Pi(\mathtt{old}(\mathtt{a}))\} \tag{1}$$

The empty *precondition* of this *Hoare triple* says that sort makes no assumptions about its input; the *postcondition* says that array a will be sorted and a permutation ($\Pi(\mathtt{old}(\mathtt{a}))$) of the original ("old") value of a after sort terminates.

For other algorithms, especially randomized ones, it is not so clear what the right specification is. For instance, consider an algorithm to shuffle the elements of an array. Certainly, shuffle should generate a permutation of the array, but

$$\{\top\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} \in \Pi(\mathtt{old}(a))\} \tag{2}$$

is an incomplete specification of shuffle at best. In fact, clearly sort satisfies specification (2) too, but for most purposes sort would be a badly behaved implementation of shuffle! A better specification would require that shuffle can generate *all* permutations. If we allow so-called *logic variables* in the triples we might give the Hoare triple (schema)

$$\{\alpha \in \Pi(\mathtt{a})\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} = \alpha\} \tag{3}$$

Unfortunately, although an abstract implementation of shuffle based on nondeterministic choice ($\sqcup$) such as

$$\bigsqcup_{\alpha \in \Pi(\mathtt{a})} \mathtt{a} := \alpha \tag{4}$$

satisfies specification (3), real implementations of shuffle that rely on a pseudo-random number generator do not: the permutation they generate will be dictated by the state of the random number generator.

* This research was supported by SFI project SFI 06 IN.1 1898.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 155–171, 2011.

We can model a random number generator as a stream of random numbers available through some global variable $\mathtt{r}$. This suggests a Hoare triple of the form

$$\{\alpha \in \Pi(\mathtt{a}), \mathtt{r} = ?\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} = \alpha\} \tag{5}$$

but it is unclear what we should put at the location of (?). We would like to state that every permutation can be generated by *some* (unspecified) choice of random number stream, but unfortunately we are forced to be more precise than that. Choose some enumeration of permutations, and let $\Pi_\iota(\alpha)$ be the ιth permutation of α. Then we could give the following specification:

$$\{\alpha = \Pi_\iota(\mathtt{a}), \mathtt{r} = \rho_\iota\}\ \mathsf{shuffle}(\mathtt{a})\ \{\mathtt{a} = \alpha\} \tag{6}$$

Specification (6) says that if shuffle is executed in an initial state with random number stream ρ_ι, then it will generate the ιth permutation. This is however rather unsatisfactory. We cannot define ρ without detailed knowledge of the algorithm; an alternative shuffling program would require a different mapping ρ' and would not satisfy the above specification. In fact, we would prefer not to have to mention $\mathtt{r}$ in the specification at all so that we do not have to adjust the specification when we refine an abstract, non-deterministic, implementation of shuffle to a real one that relies on some global state.

The main problem is that Hoare Logic uses a universal quantification over *initial* states ("for all initial states satisfying the precondition..."), while shuffle is more naturally specified using a universal quantification over *final* states ("for all permutations..."). This is precisely the purpose of Reverse Hoare Logic, in which we can give the following specification for shuffle:

$$\langle \mathtt{a} = \alpha \rangle\ \mathsf{shuffle}(\mathtt{a})\ \langle \mathtt{a} \in \Pi(\alpha) \rangle \tag{7}$$

or, using an operator "$\mathtt{new}$" dual to "$\mathtt{old}$", without logic variables:

$$\langle \mathtt{new}(\mathtt{a}) \in \Pi(\mathtt{a}) \rangle\ \mathsf{shuffle}(\mathtt{a})\ \langle \top \rangle \tag{8}$$

The *reverse triple* (7) is satisfied when all final states in which $\mathtt{a}$ is a permutation of α are reachable by executing shuffle in some initial state in which $\mathtt{a}$ has value α. It states precisely that shuffle can generate all permutations, exposes none of the implementation so that we can give this specification without reference to the algorithm, and can be used for many algorithms, regardless of whether they use random number streams or non-determinism.

Reverse triples are statements about the reachability of all "good" final states but say nothing about "bad" states. They are essentially dual to Hoare triples, which are statements about "bad" states not being reachable but do not guarantee that "good" states are reachable. This duality between the logics can also be observed in certain proof rules, such as the rule for consequence, where implications are reversed, and in the proof of completeness, which requires the definition of a *weakest postcondition*, rather than a weakest precondition, calculus. We believe that the two logics complement each other and their combination can express a complete specification for shuffle.

Since we are interested in reachability, we are interested in the *existence* of paths: a triple $\langle P \rangle\ c\ \langle Q \rangle$ is satisfied when for each final state σ' satisfying Q there is a state σ satisfying P such program c, when started in state σ, *can* terminate in state σ'. In this sense, Reverse Hoare Logic is a total logic, although the above triple does not require that c never diverges.

We make the following contributions in this paper:

1. We define Reverse Hoare Logic, a logic in which we can naturally express reachability specifications for imperative programs. Equivalent specifications in Hoare Logic would be less abstract and more difficult to define.
2. We give a proof system (Sect. 3) which can be used to prove the validity of reverse triples, without appealing to the underlying model of the logic. Some of the rules are familiar from Hoare Logic but others are different in subtle and sometimes surprising ways; for instance, the rule for loops requires the loop variant to *increase* rather than decrease.
3. We show the usefulness of the proof system by using it to derive admissible rules for complex commands, and use it to give an elegant proof for the specification of shuffle (Sect. 4).
4. We show that the proof system is sound and complete (Sect. 6). The completeness proof is based on a *weakest postcondition* calculus, which is also useful in showing the invalidity of reverse triples (Sect. 5).

2 Definitions

We use a standard imperative language with local definitions and choice ($\sqcup$); its big-step semantics is given in Fig. 1 as a mapping from states to states. Throughout this paper we will use $(\sigma, x \mapsto n)$ to denote the *extension* of σ with variable x (i.e., $x \notin \operatorname{dom} \sigma$) and $(\sigma \dagger x \mapsto n)$ to denote the extension *or* update of σ. We assume the Barendregt convention: all bound variables are assumed to be different to all other variables, and we identify programs up to alpha-renaming.

We let e range over arithmetic expressions, n over natural numbers, and let b range over boolean expressions. We use $[\![e]\!]_\sigma$ to denote the evaluation of expression e in state σ; we leave the exact definition of the syntax for arithmetic expressions and their evaluation relation open.

We use a standard first-order infinitary assertion language based on $L_{\omega_1,\omega}$ with the satisfaction relation shown in Fig. 1. To make the technical development smoother we will allow for logic variables, ranged over by ι, in expressions in the assertion language. We use e to range over these "extended" expressions too, as the intended meaning will be clear from the context. The value of these variables is given by an *interpretation* I, a mapping from logic variables to values. We can define existentials and universals in the assertion language as syntactic sugar:

$$\forall \ell \in L \cdot P \stackrel{\text{def}}{=} \bigwedge_{\ell \in L} P \qquad \exists \ell \in L \cdot P \stackrel{\text{def}}{=} \bigvee_{\ell \in L} P \tag{9}$$

Likewise, we will use $P \wedge Q$ and $P \vee Q$ to denote binary conjunctions and disjunctions, respectively. A formula P is *valid* iff $\sigma \vDash^I P$ for all states σ and interpretations I.

Operational Semantics

$$\frac{}{\sigma \xrightarrow{\texttt{skip}} \sigma} \qquad \frac{}{\sigma \xrightarrow{x:=e} \sigma \dagger x \mapsto [\![e]\!]_\sigma}$$

$$\frac{(\sigma, x \mapsto [\![e]\!]_\sigma) \xrightarrow{c} (\sigma', x \mapsto n')}{\sigma \xrightarrow{\texttt{local } x=e \texttt{ in } c} \sigma'} \qquad \frac{\sigma \xrightarrow{c_0} \sigma' \qquad \sigma' \xrightarrow{c_1} \sigma''}{\sigma \xrightarrow{c_0;c_1} \sigma''}$$

$$\frac{\sigma \vDash b \qquad \sigma \xrightarrow{c_0} \sigma'}{\sigma \xrightarrow{\texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1} \sigma'} \qquad \frac{\sigma \vDash \neg b \qquad \sigma \xrightarrow{c_1} \sigma'}{\sigma \xrightarrow{\texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1} \sigma'}$$

$$\frac{\sigma \vDash \neg b}{\sigma \xrightarrow{\texttt{while } b \texttt{ do } c} \sigma} \qquad \frac{\sigma \vDash b \qquad \sigma \xrightarrow{c} \sigma' \qquad \sigma' \xrightarrow{\texttt{while } b \texttt{ do } c} \sigma''}{\sigma \xrightarrow{\texttt{while } b \texttt{ do } c} \sigma''}$$

$$\frac{\sigma \xrightarrow{c_0} \sigma'}{\sigma \xrightarrow{c_0 \sqcup c_1} \sigma'} \qquad \frac{\sigma \xrightarrow{c_1} \sigma'}{\sigma \xrightarrow{c_0 \sqcup c_1} \sigma'}$$

Satisfaction Relation

$$\begin{array}{ll}
\sigma \vDash^I \top & \\
\sigma \vDash^I (e_0 = e_1) & \text{if } [\![e_0]\!]_{\sigma,I} = [\![e_1]\!]_{\sigma,I} \\
\sigma \vDash^I (e_0 \leq e_1) & \text{if } [\![e_0]\!]_{\sigma,I} \leq [\![e_1]\!]_{\sigma,I} \\
\sigma \vDash^I \neg P & \text{if not } \sigma \vDash^I P \\
\sigma \vDash^I P \Rightarrow Q & \text{if } \sigma \vDash^I P \text{ implies } \sigma \vDash^I Q
\end{array}$$

$$\sigma \vDash^I \bigwedge_{\ell \in L} P_\ell \quad \text{if } \sigma \vDash^I P_\ell \text{ for all } \ell \in L$$

$$\sigma \vDash^I \bigvee_{\ell \in L} P_\ell \quad \text{if } \sigma \vDash^I P_\ell \text{ for some } \ell \in L$$

Fig. 1. Operational Semantics and Satisfaction of Assertions

Definition 1 (Reverse Hoare validity). *We write* $\vDash^I \langle P \rangle\ c\ \langle Q \rangle$ *iff*

$$\forall \sigma' \vDash^I Q \cdot \exists \sigma \vDash^I P \cdot \sigma \xrightarrow{c} \sigma'$$

We write $\vDash \langle P \rangle\ c\ \langle Q \rangle$ *iff* $\vDash^I \langle P \rangle\ c\ \langle Q \rangle$*, for all* I*.*

3 Program Logic

In order to abstract away from states and interpretations, we introduce a program logic, shown in Fig. 2, which gives proof rules for each of the constructs in the language. We will show in Sect. 6 that these rules are sound and complete; in this section, we explain the rules and give examples.

Rules Skip and Assn are familiar from Hoare Logic. We can think of the existential ι in the postcondition as the old value of x. Here is a simple example:

$$\dfrac{\dfrac{}{\langle \mathtt{x} = \iota' \rangle\ \mathtt{x} := \mathtt{x} + 1\ \langle \exists \iota \in \mathbb{Z} \cdot \iota = \iota' \wedge \mathtt{x} = \iota + 1 \rangle}\ \textsc{Assn}}{\langle \mathtt{x} = \iota' \rangle\ \mathtt{x} := \mathtt{x} + 1\ \langle \mathtt{x} = \iota' + 1 \rangle}\ \textsc{Con} \tag{10}$$

We use rule Con to bring the postcondition into the right form; note the direction of the implications! This is a consequence of the underlying semantics of reverse Hoare triples.

$$\frac{}{\langle P\rangle\ x := e\ \langle \exists \iota \in L \cdot P(\iota/x) \wedge x = e(\iota/x)\rangle}\ \textsc{Assn}\ \text{(where } L \text{ is the type of } x\text{)}$$

$$\frac{\langle P \wedge x = e\rangle\ c\ \langle Q\rangle \qquad x \notin \operatorname{fn} P}{\langle P\rangle\ \texttt{local}\ x = e\ \texttt{in}\ c\ \langle \exists x \in L \cdot Q\rangle}\ \textsc{Local}\ \text{(where } L \text{ is the type of } x\text{)}$$

$$\frac{\langle P_\iota \wedge b\rangle\ c\ \langle P_{\iota+1}\rangle}{\langle P_0\rangle\ \texttt{while}\ b\ \texttt{do}\ c\ \langle \neg b \wedge \exists \iota \in \mathbb{N} \cdot P_\iota\rangle}\ \textsc{While}\ (\iota \text{ fresh})$$

$$\frac{}{\langle P\rangle\ \texttt{skip}\ \langle P\rangle}\ \textsc{Skip} \qquad \frac{\langle P\rangle\ c_0\ \langle Q\rangle \qquad \langle Q\rangle\ c_1\ \langle R\rangle}{\langle P\rangle\ c_0; c_1\ \langle R\rangle}\ \textsc{Seq}$$

$$\frac{\langle P \wedge b\rangle\ c_0\ \langle Q\rangle}{\langle P\rangle\ \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \langle Q\rangle}\ \textsc{Then} \qquad \frac{\langle P \wedge \neg b\rangle\ c_1\ \langle Q\rangle}{\langle P\rangle\ \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \langle Q\rangle}\ \textsc{Else}$$

$$\frac{\langle P\rangle\ c_0\ \langle Q\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle Q\rangle}\ \textsc{Left} \qquad \frac{\langle P\rangle\ c_1\ \langle Q\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle Q\rangle}\ \textsc{Right}$$

$$\frac{P' \Rightarrow P \quad \langle P'\rangle\ c\ \langle Q'\rangle \quad Q \Rightarrow Q'}{\langle P\rangle\ c\ \langle Q\rangle}\ \textsc{Con} \qquad \frac{\left(\forall \ell \in L \cdot \langle P\rangle\ c\ \langle Q_\ell\rangle\right)}{\langle P\rangle\ c\ \left\langle \bigvee_{\ell \in L} Q_\ell \right\rangle}\ \textsc{Split}$$

$$\frac{\langle P\rangle\ c\ \langle Q\rangle}{\langle P \wedge R\rangle\ c\ \langle Q \wedge R\rangle}\ \textsc{Frame}\ \text{(no variable occurring free in } R \text{ is modified by } c\text{)}$$

Fig. 2. Reverse Hoare Rules

In Hoare Logic, $\{P(e/x)\}\ x := e\ \{P\}$ is a popular alternative rule for assignment. The naive translation of this rule to $\langle P(e/x)\rangle\ x := e\ \langle P\rangle$ is neither sound nor complete; for instance, it allows us to derive $\langle \top\rangle\ \mathtt{x} := 2\ \langle \top\rangle$, which is invalid: it says that any state at all (including one where $\mathtt{x} \neq 2$) is reachable by executing $\mathtt{x} := 2$, which is clearly not true. On the other hand, $\langle \top\rangle\ \mathtt{x} := \mathtt{y}\ \langle \mathtt{x} = 2\rangle$ cannot be derived using this rule, while this is a valid reverse Hoare triple.

The rule for the introduction of a local variable x hides x from the postcondition and requires that x must have the specified initial value in the initial state. Here is an example:

$$\dfrac{\dfrac{\dfrac{}{\langle \mathtt{x} = \mathtt{y}\rangle\ \mathtt{z} := \mathtt{x}\ \langle \exists \iota \in \mathbb{Z} \cdot \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{x}\rangle}\ \textsc{Assn}}{\langle \mathtt{x} = \mathtt{y}\rangle\ \mathtt{z} := \mathtt{x}\ \langle \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}\rangle}\ \textsc{Con}}{\langle \top\rangle\ \texttt{local}\ \mathtt{x} = \mathtt{y}\ \texttt{in}\ \mathtt{z} := \mathtt{x}\ \langle \exists \mathtt{x} \in \mathbb{Z} \cdot \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}\rangle}\ \textsc{Local}}{\langle \top\rangle\ \texttt{local}\ \mathtt{x} = \mathtt{y}\ \texttt{in}\ \mathtt{z} := \mathtt{x}\ \langle \mathtt{z} = \mathtt{y}\rangle}\ \textsc{Con} \tag{11}$$

Rule Local is particularly useful in our setting because all global variables assigned to by the program need to be mentioned in the postcondition. For example, the triple

$$\langle \top \rangle \ \mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x} \ \langle \mathtt{z} = \mathtt{y} \rangle \tag{12}$$

is *not* valid, since final states in which $\mathtt{x} \neq \mathtt{y}$ are not reachable by executing $\mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}$.

The rule for sequential composition is as expected; the rule for conditions is more interesting. Rule Then can be used if all states that satisfy the post-condition can be reached by executing the true-branch of the conditional; rule Else is the analogous rule for the false-branch of the conditional. Typically, rule Split will first be used to partition the set of final states into those that can be reached by the true-branch and those that can be reached by the false-branch. For example:

$$\dfrac{\dfrac{\dfrac{\vdots}{\langle b \rangle \ \mathtt{x} := 1 \ \langle \mathtt{x} = 1 \rangle}}{\langle \top \rangle \ \ldots \ \langle \mathtt{x} = 1 \rangle}\ \textsc{Then} \qquad \dfrac{\dfrac{\vdots}{\langle \neg b \rangle \ \mathtt{x} := 2 \ \langle \mathtt{x} = 2 \rangle}}{\langle \top \rangle \ \ldots \ \langle \mathtt{x} = 2 \rangle}\ \textsc{Else}}{\langle \top \rangle \ \mathtt{if}\ b\ \mathtt{then}\ \mathtt{x} := 1\ \mathtt{else}\ \mathtt{x} := 2 \ \langle \mathtt{x} = 1 \vee \mathtt{x} = 2 \rangle}\ \textsc{Split} \tag{13}$$

The rules for non-deterministic choice are similar to the rules for conditionals. It remains to explain the rule for the loop construct. Like in standard Hoare logics for total correctness [1] we have to provide a *loop variant* P, a predicate over natural numbers. Unlike in standard Hoare logics, however, we have to show that P holds for a smaller number *before* the loop body. Consider the following example:

$$\langle \mathtt{x} = 0 \rangle \ \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \mathtt{x} := \mathtt{x} + 1 \sqcup \mathtt{b} := \bot \ \langle \neg \mathtt{b} \rangle \tag{14}$$

Intuitively, specification (14) is valid: any state which satisfies $\neg\mathtt{b}$, in particular, any state where $\mathtt{x}$ has some value n, can be reached from a state in which $\mathtt{x} = 0$ (pick a state where $\mathtt{b} = \top$) by executing $\mathtt{x} := \mathtt{x} + 1$ the first n iterations through the loop, followed by one iteration executing $\mathtt{b} := \bot$. We can prove the validity of this program by picking the loop variant[1]

$$\phi_n \stackrel{\text{def}}{=} (\mathtt{x} = n \wedge \mathtt{b}) \vee (\mathtt{x} = n - 1 \wedge \neg \mathtt{b}) \tag{15}$$

The proof is given in Fig. 3.

4 Case Studies

In this section we prove admissible proof rules for complex commands using the program logic and use them when we formally prove that shuffle can generate any permutation of the input array. The proof rules are summarized in Fig. 4.

[1] To simplify the example we assume $0 - 1 = 0$ for natural numbers.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle\ \mathtt{x} := \mathtt{x}+1\ \langle \mathtt{x} = \iota + 1 \wedge \mathtt{b}\rangle}\textsc{Left} \quad \dfrac{\vdots}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle\ \mathtt{b} := \bot\ \langle \mathtt{x} = \iota \wedge \neg\mathtt{b}\rangle}\textsc{Right}}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota + 1 \wedge \mathtt{b}\rangle}\textsc{Con} \quad \dfrac{}{\langle \mathtt{x} = \iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota \wedge \neg\mathtt{b}\rangle}\textsc{Con}}{\langle \phi_\iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota + 1 \wedge \mathtt{b}\rangle}\textsc{Con} \quad \dfrac{}{\langle \phi_\iota \wedge \mathtt{b}\rangle \ldots \langle \mathtt{x} = \iota \wedge \neg\mathtt{b}\rangle}\textsc{Con}}{\langle \phi_\iota \wedge \mathtt{b}\rangle\ \mathtt{x} := \mathtt{x}+1 \sqcup \mathtt{b} := \bot\ \langle \phi_{\iota+1}\rangle}\textsc{Split}}{\langle \phi_0\rangle \ldots \langle \neg\mathtt{b} \wedge \exists \iota \in \mathbb{N} \cdot \phi_\iota\rangle}\textsc{While}}{\langle \mathtt{x} = 0\rangle\ \mathtt{while\ b\ do\ x} := \mathtt{x}+1 \sqcup \mathtt{b} := \bot\ \langle \neg\mathtt{b}\rangle}\textsc{Con}$$

Fig. 3. Loop Example (ϕ_n defined in equation (15))

4.1 Picking Random Numbers

Assuming the availability of a stream of random numbers $n_1 : n_2 : \ldots$ through global variable $\mathtt{r}$, picking a random number boils down to

$$\Big(x := \mathtt{rnd}[0, e]\Big) \stackrel{\text{def}}{=} \Big(x := \mathrm{head}(\mathtt{r}); \mathtt{r} := \mathrm{tail}(\mathtt{r})\Big) \tag{16}$$

The proof that this satisfies the expected specification is an easy exercise. Let Q be the post-condition $\exists \iota \cdot P(\iota/x) \wedge e \leq x \leq e'$. Observe that

$$\dfrac{\dfrac{}{\left\langle \begin{array}{l} P \\ \wedge\ \exists \rho, \iota \cdot e \leq \iota \leq e' \\ \wedge\ \mathtt{r} = \iota : \rho \end{array} \right\rangle \ldots \left\langle \begin{array}{l} \exists \iota' \cdot P(\iota'/x) \\ \wedge\ \exists \rho, \iota \cdot e \leq \iota \leq e' \\ \wedge\ \mathtt{r} = \iota : \rho \\ \wedge\ x = \mathrm{head}(\mathtt{r}) \end{array} \right\rangle}\textsc{Assn}}{\langle P \wedge \exists \rho, \iota \cdot e \leq \iota \leq e' \wedge \mathtt{r} = \iota : \rho\rangle \ldots \langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle}\textsc{Con}}{\langle P\rangle\ x := \mathrm{head}(\mathtt{r})\ \langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle}\textsc{Con} \tag{17}$$

Moreover,

$$\dfrac{\dfrac{\dfrac{}{\langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle\ \mathtt{r} := \mathrm{tail}(\mathtt{r})\ \langle \exists \rho' \cdot Q \wedge \exists \rho \cdot \rho' = x : \rho \wedge \mathtt{r} = \mathrm{tail}(\rho')\rangle}\textsc{Assn}}{\langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle\ \mathtt{r} := \mathrm{tail}(\mathtt{r})\ \langle Q \wedge \exists \rho \cdot \mathtt{r} = \rho\rangle}\textsc{Con}}{\langle Q \wedge \exists \rho \cdot \mathtt{r} = x : \rho\rangle\ \mathtt{r} := \mathrm{tail}(\mathtt{r})\ \langle Q\rangle}\textsc{Con} \tag{18}$$

The required specification follows by combining (17) and (18) using Seq. We leave the generalization to $x := \mathtt{rnd}[e, e']$ as an (easy) exercise for the reader.

4.2 Arrays

We deal with arrays by adopting the approach taken in Hoare Logic [13]: we consider an expression language that contains expressions $a[e]$ and $a \dagger e \mapsto e'$ for array indexing and override, respectively.

We define two derived commands to update the value of an array at a particular index and two swap to elements:

$$\Big(a[e] := e'\Big) \stackrel{\text{def}}{=} \Big(a := a \dagger e \mapsto e'\Big)$$
$$\Big(\texttt{swap}\ a[e, e']\Big) \stackrel{\text{def}}{=} \Big(\texttt{local}\ x = a[e]\ \texttt{in}\ a[e] := a[e']; a[e'] := x\Big)$$

The associated proof rules Upd and Swap are shown in Fig. 4. As syntactic conventions, we use a to range over array valued expressions, $\texttt{a}$ for array valued program variables, and α for array valued logic variables.

4.3 Iteration

We introduce a `for` loop as syntactic sugar; for simplicity, we fix the lower bound:

$$\Big(\texttt{for}\ x\ \texttt{in}\ [0, e)\ \texttt{do}\ c\Big) \stackrel{\text{def}}{=} \Big(\texttt{local}\ x = 0\ \texttt{in}\ \texttt{while}\ x < e\ \texttt{do}\ c; x := x + 1\Big) \quad (19)$$

Rule For is simpler than While, because termination is guaranteed; hence, we only need to provide a loop *invariant* P, rather than a loop variant.[2]

Proving For is a good exercise. Let $\phi_n \stackrel{\text{def}}{=} P \wedge x = n \leq e$. We derive the rule as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\langle P\rangle\ c\ \langle P({}^{x+1}/_{x})\rangle}{\langle P \wedge x = i < e\rangle\ c\ \langle P({}^{x+1}/_{x}) \wedge x = i < e\rangle}\ \textsc{Frame} \quad \nabla}{\langle P \wedge x = i < e\rangle\ c; x := x + 1\ \langle P \wedge x = i + 1 \leq e\rangle}\ \textsc{Seq}}{\langle P \wedge x = i \leq e\rangle\ c; x := x + 1\ \langle P \wedge x = i + 1 \leq e\rangle}\ \textsc{Con}}{\langle \phi_0\rangle\ \texttt{while}\ x < e\ \texttt{do}\ \ldots\ \langle \neg(x < e) \wedge \exists i \cdot \phi_i\rangle}\ \textsc{While}}{\langle P \wedge x = 0\rangle\ \texttt{while}\ x < e\ \texttt{do}\ \ldots\ \langle \neg(x < e) \wedge x \leq e \wedge P\rangle}\ \textsc{Con}}{\langle P \wedge x = 0\rangle\ \texttt{while}\ x < e\ \texttt{do}\ \ldots\ \langle x = e \wedge P\rangle}\ \textsc{Con}}{\langle P({}^{0}/_{x})\rangle\ \texttt{local}\ x = 0\ \texttt{in}\ \ldots\ \langle \exists x \cdot x = e \wedge P\rangle}\ \textsc{Local}}{\langle P({}^{0}/_{x})\rangle\ \texttt{for}\ x\ \texttt{in}\ [0, e)\ \texttt{do}\ c\ \langle P({}^{e}/_{x})\rangle}\ \textsc{Con} \quad (20)$$

where ∇ is a derivation with conclusion

$$\big\langle P({}^{x+1}/_{x}) \wedge x = i < e\big\rangle\ x := x + 1\ \langle P \wedge x = i + 1 \leq e\rangle \quad (21)$$

4.4 Shuffle

We are now in a position to prove that shuffle can generate any permutation. We give the Fisher-Yates implementation of shuffle as follows:

$$\texttt{for x in}\ [0, |\texttt{a}|)\ \texttt{do local y} = 0\ \texttt{in y} := \texttt{rnd}[\texttt{x}, |\texttt{a}| - 1]; \texttt{swap a}[\texttt{x}, \texttt{y}] \quad (22)$$

[2] P truly is a loop invariant; although we require $P({}^{x+1}/_{x})$ after c, this means that P will be true again after x is incremented.

$$\frac{\langle P\rangle\ c\ \langle P(^{x+1}/_x)\rangle}{\langle P(^{0}/_x)\rangle\ \texttt{for}\ x\ \texttt{in}\ [0,e)\ \texttt{do}\ c\ \langle P(^{e}/_x)\rangle}\ \textsc{For}\quad \begin{pmatrix}c \text{ cannot change } x \text{ or any variable}\\ \text{appearing in } e\end{pmatrix}$$

$$\frac{x \notin e, e'}{\langle P\rangle\ x := \texttt{rnd}[e,e']\ \langle \exists \iota \cdot P(^{\iota}/_x) \wedge e \le x \le e'\rangle}\ \textsc{Rnd}$$

$$\frac{a \notin e}{\langle P\rangle\ a[e] := e'\ \langle \exists \alpha \in [\mathbb{Z}] \cdot P(^{\alpha}/_a) \wedge a = \alpha \dagger e \mapsto e'(^{\alpha}/_a)\rangle}\ \textsc{Upd}$$

$$\frac{a \notin e, e'}{\langle P\rangle\ \texttt{swap}\ a[e,e']\ \langle \exists \alpha \in [\mathbb{Z}] \cdot P(^{\alpha}/_a) \wedge a = \alpha \dagger e \leftrightarrow e'\rangle}\ \textsc{Swap}$$

Fig. 4. Derived Proof Rules

In order to prove shuffle correct, we need a lemma that says that any permutation of an array α of size $|\alpha|$ can be generated by first swapping the first element with an element $0 \le m < |\alpha|$, then the second element with an element $1 \le m' < |\alpha|$, etc. Formally:

Lemma 1 (Permutations). *Define an indexed predicate π_n as follows.*

$$\begin{array}{lll} a'\ \pi_0\ a & \textit{iff} & a' = a \\ a'\ \pi_{n+1}\ a & \textit{iff} & \exists m \cdot a'\ \pi^m_{n+1}\ a \end{array}$$

where $a'\ \pi^m_{n+1}\ a = \exists a'' \cdot n \le m < |a| \wedge \left(a' = a'' \dagger n \leftrightarrow m\right) \wedge a''\ \pi_n\ a$.
Then $a' \in \Pi(a)$ *iff* $a'\ \pi_{|a|}\ a$.

We can use $\mathtt{a}\ \pi_{\mathtt{x}}\ \alpha$ as our loop invariant[3] and give the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\nabla_1 \qquad \nabla_2}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{y} = 0\rangle \ \ldots\ \langle \mathtt{a}\ \pi^{\mathtt{y}}_{\mathtt{x}+1}\ \alpha\rangle}\ \textsc{Seq}}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha\rangle\ \mathtt{local\ y} = 0\ \mathtt{in}\ \ldots\ \langle \exists \mathtt{y} \cdot \mathtt{a}\ \pi^{\mathtt{y}}_{\mathtt{x}+1}\ \alpha\rangle}\ \textsc{Local}}{\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha\rangle\ \mathtt{local\ y} = 0\ \mathtt{in}\ \ldots\ \langle \mathtt{a}\ \pi_{\mathtt{x}+1}\ \alpha\rangle}\ \textsc{Con}}{\left\langle (\mathtt{a}\ \pi_{\mathtt{x}}\ \alpha)(^{0}/_{\mathtt{x}})\right\rangle\ \mathtt{for\ x\ in}\ [0,|\mathtt{a}|)\ \mathtt{do}\ \ldots\ \left\langle (\mathtt{a}\ \pi_{\mathtt{x}}\ \alpha)(^{|\mathtt{a}|}/_{\mathtt{x}})\right\rangle}\ \textsc{For}}{\langle \mathtt{a} = \alpha\rangle\ \mathsf{shuffle}(\mathtt{a})\ \langle \mathtt{a} \in \Pi(\alpha)\rangle}\ \textsc{Con}\ (\text{Lem. 1}) \tag{23}$$

where ∇_1 is a derivation with conclusion

$$\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{y} = 0\rangle\ \mathtt{y} := \mathtt{rnd}[\mathtt{x}, |\mathtt{a}| - 1]\ \langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{x} \le \mathtt{y} < |\mathtt{a}|\rangle \tag{24}$$

[3] Technically the subscript to π must be a natural number; we can regard this loop invariant as syntactic sugar for $\bigvee_{n \in \mathbb{N}} \mathtt{a}\ \pi_n\ \alpha \wedge x = n$.

which can be proved using Rnd and Con. Derivation ∇_2 is given by

$$\dfrac{\dfrac{}{\left\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{x} \leq \mathtt{y} < |\mathtt{a}| \right\rangle\ \mathtt{swap\ a[x,y]}\ \left\langle \begin{array}{l} \exists \alpha' \cdot\ \alpha'\ \pi_{\mathtt{x}}\ \alpha \\ \quad \wedge\ (\mathtt{a} = \alpha' \dagger \mathtt{x} \leftrightarrow \mathtt{y}) \\ \quad \wedge\ \mathtt{x} \leq \mathtt{y} < |\mathtt{a}| \end{array} \right\rangle}\ \textsc{Swap}}{\left\langle \mathtt{a}\ \pi_{\mathtt{x}}\ \alpha \wedge \mathtt{x} \leq \mathtt{y} < |\mathtt{a}| \right\rangle\ \mathtt{swap\ a[x,y]}\ \left\langle \mathtt{a}\ \pi^{\mathtt{y}}_{\mathtt{x}+1}\ \alpha \right\rangle}\ \textsc{Con} \tag{25}$$

The difficulty of this proof is on par with the difficulty of proofs of comparable properties in Hoare Logic, for instance proving that sort is correct. As in Hoare Logic proofs, the key step in the proof is identifying the loop invariant.

5 Weakest Postcondition Calculus

In Sect. 3 we presented a program logic for deriving reverse Hoare triples. In this section we present a weakest-postcondition calculus, shown in Fig. 5, which computes $\mathrm{wpo}(P, c)$, the *weakest* postcondition given a precondition P and program c. We then have that $\langle P \rangle\ c\ \langle Q \rangle$ is a valid triple if and only if $Q \Rightarrow \mathrm{wpo}(P, c)$. As we shall see in Sect. 6, the calculus is also an essential ingredient in proving completeness of the program logic.

Moreover, a weakest postcondition calculus gives us a tool for proving that triples are *not* valid. For instance, in Sect. 3 we claimed that triple (12)

$$\langle \top \rangle\ \mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}\ \langle \mathtt{z} = \mathtt{y} \rangle$$

was invalid. Using our calculus we can prove this formally. We compute

$$\begin{aligned} &\mathrm{wpo}(\top, \mathtt{x} := \mathtt{y}; \mathtt{z} := \mathtt{x}) = \mathrm{wpo}(\mathrm{wpo}(\top, \mathtt{x} := \mathtt{y}), \mathtt{z} := \mathtt{x}) \\ &= \mathrm{wpo}(\mathtt{x} = \mathtt{y}, \mathtt{z} := \mathtt{x}) = (\mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{x}) = (\mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}) \end{aligned}$$

Since $\mathtt{z} = \mathtt{y} \not\Rightarrow \mathtt{x} = \mathtt{y} \wedge \mathtt{z} = \mathtt{y}$ (it is easy to find a counterexample) we can conclude that this triple is indeed invalid, and the weakest postcondition tells us precisely what is missing: the condition on $\mathtt{x}$.

6 Soundness and Completeness

We first show that any reverse Hoare triple derivable by the proof system is valid. We rely on two auxiliary lemmas, which we state without proof. The first is a standard substitution lemma:

Lemma 2. *$\sigma \vDash^I P(^n/_x)$ iff $(\sigma \dagger x \mapsto n) \vDash^I P$.*

The second states that predicates are preserved by programs that do not modify any of the variables mentioned in the predicate:

Lemma 3. *If $\sigma' \vDash^I P$, $\sigma \xrightarrow{c} \sigma'$ and no variable occurring free in P is modified by c, then $\sigma \vDash^I P$.*

$$
\begin{aligned}
\mathrm{wpo}(P, \texttt{skip}) &= P \\
\mathrm{wpo}(P, x := e) &= \exists \ell \in L \cdot P(\ell/x) \wedge x = e(\ell/x) \\
\mathrm{wpo}(P, c_0; c_1) &= \mathrm{wpo}(\mathrm{wpo}(P, c_0), c_1) \\
\mathrm{wpo}(P, \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1) &= \mathrm{wpo}(P \wedge b, c_0) \vee \mathrm{wpo}(P \wedge \neg b, c_1) \\
\mathrm{wpo}(P, c_0 \sqcup c_1) &= \mathrm{wpo}(P, c_0) \vee \mathrm{wpo}(P, c_1) \\
\mathrm{wpo}(P, \texttt{local } x = e \texttt{ in } c) &= \exists x \in L \cdot \mathrm{wpo}(P \wedge x = e, c) \\
\mathrm{wpo}(P, \texttt{while } b \texttt{ do } c) &= \neg b \wedge \bigvee_{n \in \mathbb{N}} \Upsilon_{b,c,P}(n)
\end{aligned}
$$

where

$$
\begin{aligned}
\Upsilon_{b,c,P}(0) &= P \\
\Upsilon_{b,c,P}(n+1) &= \mathrm{wpo}(\Upsilon_{b,c,P}(n) \wedge b, c)
\end{aligned}
$$

Fig. 5. Weakest Postcondition Calculus

We can now state and prove soundness.

Theorem 1 (Soundness). *If* $\vdash \langle P \rangle\ c\ \langle Q \rangle$ *then* $\vDash \langle P \rangle\ c\ \langle Q \rangle$.

Proof. By induction on the derivation of $\langle P \rangle\ c\ \langle Q \rangle$. The cases for Skip, Seq, Then, Else, Left, Right, Con and Split are straightforward. We give details for the other cases.

1. Case $\dfrac{}{\langle P \rangle\ x := e\ \langle \exists i \cdot P(i/x) \wedge x = e(i/x) \rangle}$ Assn.
   Pick an I, σ' such that $\sigma' \vDash^I \exists i \cdot P(i/x) \wedge x = e(i/x)$ i.e. $\exists n \cdot \sigma' \vDash^I P(n/x) \wedge x = e(n/x)$. Choose $\sigma = (\sigma' \dagger x \mapsto n)$. By Lem. 2 we have $\sigma' \vDash P(n/x)$ iff $\sigma', x \mapsto n \vDash P$. Finally, since $\sigma \dagger x \mapsto [\![e]\!]_\sigma = (\sigma' \dagger x \mapsto n) \dagger x \mapsto [\![e]\!]_{\sigma' \dagger x \mapsto n} = \sigma' \dagger x \mapsto [\![e]\!]_{\sigma' \dagger x \mapsto n} = \sigma' \dagger x \mapsto [\![e(n/x)]\!]_{\sigma'} = \sigma'$, we have $\sigma \xrightarrow{x:=e} \sigma'$.
2. Case $\dfrac{\langle P \wedge x = e \rangle\ c\ \langle Q \rangle \qquad x \notin \mathrm{fn}\, P}{\langle P \rangle\ \texttt{local } x = e \texttt{ in } c\ \langle \exists x \cdot Q \rangle}$ Local.
   Pick an I, σ' such that $\sigma' \vDash^I \exists x \cdot Q$. We have $\sigma' \vDash^I \exists x \cdot Q$ i.e. $\exists n' \cdot \sigma' \vDash^I Q(n'/x)$ i.e. $\exists n' \cdot (\sigma', x \mapsto n') \vDash^I Q$. By the induction hypothesis there exist a σ_0 such that $\sigma_0 \xrightarrow{c} (\sigma', x \mapsto n')$ and $\sigma_0 \vDash^I P \wedge x = e$. Hence, there must be some $\sigma \vDash^I P$ such that $\sigma_0 = \sigma, x \mapsto [\![e]\!]_{\sigma_0} = \sigma, x \mapsto [\![e]\!]_\sigma$ (since $x \notin e, P$). The proof is completed by $\dfrac{(\sigma, x \mapsto [\![e]\!]_\sigma) \xrightarrow{c} (\sigma', x \mapsto n')}{\sigma \xrightarrow{\texttt{local } x=e \texttt{ in } c} \sigma'}$.
3. Case $\dfrac{\langle P_i \wedge b \rangle\ c\ \langle P_{i+1} \rangle}{\langle P_0 \rangle\ \texttt{while } b \texttt{ do } c\ \langle \neg b \wedge \exists i \cdot P_i \rangle}$ While.
   Pick an interpretation I and a state σ' such that $\sigma' \vDash^I \neg b \wedge \exists i \cdot P_i$ i.e. $\exists n \cdot \sigma' \vDash^I \neg b \wedge P_n$. If $n = 0$ we take $\sigma = \sigma'$ and we're done. Otherwise we use the premise to obtain a series of states which satisfy $P(n') \wedge b$ for a decreasing n' until we reach a state that satisfies $P(0)$.
4. Case $\dfrac{\langle P \rangle\ c\ \langle Q \rangle}{\langle P \wedge R \rangle\ c\ \langle Q \wedge R \rangle}$ Frame.

Pick an interpretation I and a state $\sigma' \vDash^I Q \wedge R$. Clearly $\sigma' \vDash^I Q$ so that by the premise there exist an $\sigma \vDash^I P$ where $\sigma \xrightarrow{c} \sigma'$; $\sigma \vDash^I R$ follows from Lem. 3.

We prove relative completeness[4] in two steps. We establish that our weakest-postcondition calculus calculates the largest set of states that can be reached by a precondition and a program, and then use this result to prove completeness.

Definition 2 (Weakest postcondition). *The weakest postcondition of a precondition P and program c, with respect to an interpretation I, is defined by*

$$wpo^I[\![P, c]\!] \stackrel{def}{=} \{\sigma' \mid \exists \sigma \cdot \sigma \vDash^I P \wedge \sigma \xrightarrow{c} \sigma'\}$$

The calculus presented in Sect. 5 characterizes exactly this set. This property is sometimes referred to as the *expressivity* of the assertion language. To prove it, we first give a characterization of Υ:

Lemma 4 (Υ). *Given a program c such that*

$$\forall \sigma', P, I \cdot \sigma' \in wpo^I[\![P, c]\!] \text{ iff } \sigma' \vDash^I wpo(P, c)$$

and given boolean predicate b and precondition P we have $\forall n, \sigma' \cdot$

$$\sigma' \vDash^I \Upsilon_{b,c,P}(n) \text{ iff } \exists \sigma_{0 \le i \le n} \cdot \sigma_n = \sigma' \wedge \sigma_{0 \le i < n} \vDash b \wedge \sigma_0 \vDash^I P \wedge \sigma_{0 \le i < n} \xrightarrow{c} \sigma_{i+1}$$

Proof. By induction on n.

Proposition 1. $\forall \sigma' \cdot \sigma' \in wpo^I[\![P, c]\!]$ *iff* $\sigma' \vDash^I wpo(P, c)$.

Proof. By induction on c, similar to the proof of Theorem 1. The case for loops relies on Lem. 4.

It remains to show that the weakest postcondition is always derivable in the program logic; completeness of the logic then follows.

Proposition 2. $\forall c, P$ *we have* $\vdash \langle P \rangle\ c\ \langle wpo(P, c) \rangle$.

Proof. By induction on c. The cases for Skip and Assn are immediate. The other cases are detailed below.

1. Case $c = \texttt{local}\ x = e\ \texttt{in}\ c_0$.

$$\frac{\langle P \wedge x = e \rangle\ c_0\ \langle \mathrm{wpo}(P \wedge x = e, c) \rangle}{\langle P \rangle\ \texttt{local}\ x = e\ \texttt{in}\ c_0\ \langle \exists x \cdot \mathrm{wpo}(P \wedge x = e, c) \rangle}\ \textsc{Local}$$

2. Case $c = c_0; c_1$.

$$\frac{\langle P \rangle\ c_0\ \langle \mathrm{wpo}(P, c_0) \rangle \quad \langle \mathrm{wpo}(P, c_0) \rangle\ c_1\ \langle \mathrm{wpo}(\mathrm{wpo}(P, c_0), c_1) \rangle}{\langle P \rangle\ c_0; c_1\ \langle \mathrm{wpo}(\mathrm{wpo}(P, c_0), c_1) \rangle}\ \textsc{Seq}$$

[4] Also known as completeness in the sense of Cook [1, Sect. 2.8].

3. Case $c = \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1$.

$$\dfrac{\dfrac{\langle P \wedge b\rangle\ c_0\ \langle \mathrm{wpo}(P \wedge b, c_0)\rangle}{\langle P\rangle\ \ldots\ \langle \mathrm{wpo}(P \wedge b, c_0)\rangle}\ \textsc{Then} \quad \dfrac{\langle P \wedge \neg b\rangle\ c_1\ \langle \mathrm{wpo}(P \wedge \neg b, c_1)\rangle}{\langle P\rangle\ \ldots\ \langle \mathrm{wpo}(P \wedge \neg b, c_1)\rangle}\ \textsc{Else}}{\langle P\rangle\ \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \langle \mathrm{wpo}(P \wedge b, c_0) \vee \mathrm{wpo}(P \wedge \neg b, c_1)\rangle}\ \textsc{Split}$$

4. Case $c = \texttt{while}\ b\ \texttt{do}\ c_0$.

$$\dfrac{\dfrac{\dfrac{\langle \Upsilon_{b,c_0,P}(n) \wedge b\rangle\ c_0\ \langle \mathrm{wpo}(\Upsilon_{b,c_0,P}(n) \wedge b, c_0)\rangle}{\langle \Upsilon_{b,c_0,P}(n) \wedge b\rangle\ c_0\ \langle \Upsilon_{b,c_0,P}(n+1)\rangle}\ \textsc{Con}}{\langle \Upsilon_{b,c_0,P}(0)\rangle\ \texttt{while}\ b\ \texttt{do}\ c_0\ \left\langle \neg b \wedge \bigvee_n \Upsilon_{b,c_0,P}(n) \right\rangle}\ \textsc{While}}{\langle P\rangle\ \texttt{while}\ b\ \texttt{do}\ c_0\ \langle \neg b \wedge \exists i \cdot \Upsilon_{b,c_0,P}(i)\rangle}\ \textsc{Con}$$

5. Case $c = c_0 \sqcup c_1$.

$$\dfrac{\dfrac{\langle P\rangle\ c_0\ \langle \mathrm{wpo}(P, c_0)\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle \mathrm{wpo}(P, c_0)\rangle}\ \textsc{Left} \quad \dfrac{\langle P\rangle\ c_1\ \langle \mathrm{wpo}(P, c_0)\rangle}{\langle P\rangle\ c_0 \sqcup c_1\ \langle \mathrm{wpo}(P, c_0)\rangle}\ \textsc{Right}}{\langle P\rangle\ c_0 \sqcup c_1\ \langle \mathrm{wpo}(P, c_0) \vee \mathrm{wpo}(P, c_1)\rangle}\ \textsc{Split}$$

Lemma 5. *If* $\vDash \langle P\rangle\ c\ \langle Q\rangle$ *then* $Q \Rightarrow wpo(P, c)$.

Proof.

$$\begin{array}{lll} \vDash \langle P\rangle\ c\ \langle Q\rangle & \text{iff } \forall I, \sigma' \vDash^I Q \cdot \exists \sigma \vDash^I P \wedge \sigma \xrightarrow{c} \sigma' & \text{iff } \forall I, \sigma' \vDash^I Q \cdot \sigma' \in \mathrm{wpo}^I[\![P, c]\!] \\ & \text{iff } \forall I, \sigma' \vDash^I Q \cdot \sigma' \vDash^I \mathrm{wpo}(P, c) & \text{iff } Q \Rightarrow \mathrm{wpo}(P, c) \end{array}$$

Theorem 2 (Completeness). *If* $\vDash \langle P\rangle\ c\ \langle Q\rangle$ *then* $\vdash \langle P\rangle\ c\ \langle Q\rangle$.

Proof. Follows from rule Con and Prop. 2.

7 Related Work

Hoare Logic was introduced by Hoare in 1969 [12], and has since grown into a very active research field. Ten years after its invention a survey already spanned two papers [1,2]. Recent work has introduced the concept of *separation* into the logic to deal with aliasing [25], extended it to functional languages [15,16] and embedded Hoare logic in type theory [22].

As already discussed, however, Hoare logics are not suitable for state reachability specifications, with the exception of probabilistic Hoare logics [23,18,10,5,21]. A probabilistic specification for shuffle might say that the probability of the generation of any permutation is greater than zero; indeed, it might say that every permutation is equally likely. Although such exact guarantees cannot be proven using Reverse Hoare Logic, proofs in Reverse Hoare Logic are simpler than proofs

in probabilistic Hoare logics. The assertion language in Reverse Hoare Logic is a familiar first order logic, and reasoning does not involve manipulating probabilities or state distributions.

The notions of "weakest postcondition" and "strongest precondition" appear only occasionally in the literature, mostly in the context of incomplete knowledge. For instance, when using dynamic binding in object oriented programming, the caller can only assume the weakest postcondition, i.e. the postcondition of the method in the top of the inheritance hierarchy [11]. Similar situations arise with web services [19] and contracts [24], and in artificial intelligence when omitting information in an attempt to simplify a search domain [3,26]. In UTP these notions are used to define recursion in the theory of "designs" [27].

Dynamic logic [8,9] is a multi-modal logic; for any program c the formula $\langle c \rangle P$ is satisfied if c can terminate in a state satisfying P, and the formula $[c]P$ is satisfied if all states that c terminates in satisfy P (there might not be any). The Hoare triple $\{\alpha \in \Pi(a)\}\ \mathsf{shuffle}\ \{a = \alpha\}$ can be expressed as

$$\forall \alpha \cdot \alpha \in \Pi(a) \rightarrow \langle \mathsf{shuffle}(\mathtt{a}) \rangle (a = \alpha) \tag{26}$$

in Dynamic Logic. Like in Hoare Logic, however, an implementation of shuffle which relies on a global random number stream would not satisfy specification (26). The more general treatment of qualifiers in Dynamic Logic means that in Dynamic Logic we do not have to be quite as precise as in Hoare Logic:

$$\forall \alpha, \iota \cdot \exists \rho \cdot \alpha = \pi_i(a), \mathtt{r} = \rho \rightarrow \langle \mathsf{shuffle}(\mathtt{a}) \rangle (a = \alpha) \tag{27}$$

Specification (27) is better than the Hoare triple (6) as we do not have to specify the precise relation between permutations and random number streams, so that this specification is satisfied by more implementations of shuffle. Nevertheless, this specification still exposes an implementation detail (the reliance on r).

One might envision extending Dynamic Logic with a reverse modality to obtain a "Reverse Dynamic Logic". This would certainly be of interest. Finally, we remark that a lot of the literature on Dynamic Logic is concerned with the treatment of "totality" in the presence of non-determinism; this is less relevant in our setting, since we are interested in reachability.

If we have an inverse operation c^{-1} on programs such that $\sigma \xrightarrow{c^{-1}} \sigma'$ exactly when $\sigma' \xrightarrow{c} \sigma$, it is immediate from the definitions of validity of total Hoare triples (with "for all initial... exists final..." semantics) that $\langle P \rangle\ c\ \langle Q \rangle$ exactly when $\{Q\}\ c^{-1}\ \{P\}$. Reverse Hoare Logic is thus related to the old idea of program inversion [6,28], but we avoid the need for computing program inverses. From another perspective, Reverse Hoare Logic provides a way to prove Hoare specifications of inverse programs without having to compute the inverse.

Hoare-style "contracts" (pre- and post-conditions) are often implemented as runtime checks in programming languages such as Eiffel [20]. It is not obvious how to check "reverse contracts" at runtime. However, static verification of contracts is slowly becoming a reality through tools such as ESC/Java2 [7] and Spec# [4]. It should not be too difficult to extend these tools to support verification of reverse contracts too, although we have not attempted to do so. Some

of these tools do not support logic variables, amplifying the need for a reverse logic to reason about reachability.[5]

8 Conclusions

Reverse Hoare Logic can be used to give and prove reachability specifications. Compared to the alternatives, the specifications can be more abstract than in Hoare logic, and the corresponding proofs are simpler than when using a probabilistic logic. Reverse Hoare Logic naturally gives rise to the concept of a weakest postcondition, which we have used to show that the proof system is complete.

It would be worthwhile to attempt to combine standard and reverse Hoare logic, yielding a logic in which we can express both the reachability of good states and the non-reachability of bad states. An extension to functional languages, especially higher-order ones, would also be of interest.

The use of an infinitary logic enabled an elegant definition of the weakest postcondition calculus and hence made the completeness proof easier. However, a formulation using a finitary logic would be useful for an implementation of Reverse Hoare Logic in an automatic theorem prover.

Finally, it would be interesting to look at adaptation in the context of Reverse Hoare Logic. We believe that a simple adaptation rule such as equation (6) in [17] is sound for Reverse Hoare Logic (*mutatis mutandis*), but it is unclear at present if Reverse Hoare Logic can be made adaptation complete.

Acknowledgements. We would like to thank Colm Bhandal for an insightful discussion on the expressivity of Hoare triples in the presence of non-determinism, and to Hugh Gibbons for providing valuable references.

References

1. Apt, K.R.: Ten years of Hoare's logic: A survey—part I. ACM Trans. Program. Lang. Syst. 3, 431–483 (1981)
2. Apt, K.R.: Ten years of Hoare's logic: A survey—part II: Nondeterminism. Theoretical Computer Science 28(1-2), 83–109 (1983)
3. Bacchus, F., Yang, Q.: Downward refinement and the efficiency of hierarchical problem solving. Artificial Intelligence 71(1), 43–100 (1994)
4. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.S.: The spec# programming system: Challenges and directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)

[5] In some tools such as ESC/Java2 we can use "ghost variables" (ordinary variables appearing only in specifications and proofs) to express reachability properties; e.g. for the nondeterministic shuffle we can write $\{\gamma \in \Pi(\mathtt{a})\}$ shuffle $\{\mathtt{a} = \gamma \wedge \gamma = \mathtt{old}(\gamma)\}$. Again, deterministic implementations of shuffle do not satisfy this specification. Note that techniques to remove ghost variables (*cf.* [14]) do not apply here.

5. Chadha, R., Cruz-Filipe, L., Mateus, P., Sernadas, A.: Reasoning about probabilistic sequential programs. Theor. Comput. Sci. 379, 142–165 (2007)
6. Chen, W., Udding, J.T.: Program inversion: more than fun! Sci. Comput. Program. 15, 1–13 (1990)
7. Cok, D., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
8. Goldblatt, R.: Logics of time and computation. Center for the Study of Language and Information, Stanford, CA, USA (1987)
9. Harel, D.: Logics of programs: Axiomatics and descriptive power. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
10. den Hartog, J.: Verifying probabilistic programs using a Hoare like logic. In: Thiagarajan, P., Yap, R. (eds.) ASIAN 1999. LNCS, vol. 1742, pp. 790–790. Springer, Heidelberg (1999)
11. Heyer, T.: Semantic Inspection of Software Artifacts From Theory to Practice. Ph.D. thesis, Linköping University (2001)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12, 576–580 (1969)
13. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language PASCAL. Acta Informatica 2, 335–355 (1973)
14. Hofmann, M., Pavlova, M.: Elimination of ghost variables in program logics. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 1–20. Springer, Heidelberg (2008)
15. Honda, K., Yoshida, N., Berger, M.: An observationally complete program logic for imperative higher-order functions. In: LICS, pp. 270–279. IEEE, Los Alamitos (2005)
16. Kanig, J., Filliâtre, J.C.: Who: a verifier for effectful higher-order programs. In: ACM SIGPLAN Workshop on ML, pp. 39–48. ACM, New York (2009)
17. Kleymann, T.: Hoare logic and auxiliary variables. Formal Aspects of Computing 11, 541–566 (1999), http://dx.doi.org/10.1007/s001650050057, doi:10.1007/s001650050057
18. Kozen, D.: A probabilistic PDL. J. Comp. and Sys. Sc. 30(2), 162–178 (1985)
19. Kumar, A., Srivastava, B., Mittal, S.: Information modeling for end to end composition of semantic web services. In: Gil, Y., Motta, E., Benjamins, V.R., Musen, M.A. (eds.) ISWC 2005. LNCS, vol. 3729, pp. 476–490. Springer, Heidelberg (2005)
20. Meyer, B.: Object-oriented software construction, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (1997)
21. Morgan, C., McIver, A., Seidel, K.: Probabilistic predicate transformers. ACM Trans. Program. Lang. Syst., 325–353 (1996)
22. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare Type Theory. SIGPLAN Not. 41, 62–73 (2006)
23. Ramshaw, L.H.: Formalizing the analysis of algorithms. Ph.D. thesis, Stanford University (1979)
24. Reussner, R., Poernomo, I., Schmidt, H.: Reasoning about software architectures with contractually specified components. In: Cechich, A., Piattini, M., Vallecillo, A. (eds.) CBSQ 2003. LNCS, vol. 2693, pp. 287–325. Springer, Heidelberg (2003)
25. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society, Washington, DC, USA (2002)

26. ten Teije, A., van Harmelen, F.: Characterising approximate problem solving: by partially fulfilled pre- and postconditions. In: ECAI 1998. CEUR-WS, vol. 16, pp. 78–82 (1998)
27. Woodcock, J., Cavalcanti, A.: A Tutorial Introduction to Designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 40–66. Springer, Heidelberg (2004)
28. von Wright, J.: Program inversion in the refinement calculus. Information Processing Letters 37(2), 95–100 (1991)

Improving SAT Modulo ODE for Hybrid Systems Analysis by Combining Different Enclosure Methods*

Andreas Eggers[1], Nacim Ramdani[2], Nedialko Nedialkov[3], and Martin Fränzle[1]

[1] Carl von Ossietzky Universität, Oldenburg, Germany
{eggers,fraenzle}@informatik.uni-oldenburg.de
[2] Université d'Orléans, PRISME, 63 av. de Lattre de Tassigny, 18020 Bourges, France
nacim.ramdani@bourges.univ-orleans.fr
[3] McMaster University, Hamilton, Ontario, Canada
nedialk@mcmaster.ca

Abstract. Aiming at automatic verification and analysis techniques for hybrid systems, we present a novel combination of enclosure methods for ordinary differential equations (ODEs) with the iSAT solver for large Boolean combinations of arithmetic constraints. Improving on our previous work, the contribution of this paper lies in combining iSAT with VNODE-LP, as a state-of-the-art enclosure method for ODEs, and with bracketing systems which exploit monotonicity properties to find enclosures for problems that VNODE-LP alone cannot enclose tightly. We apply our method to the analysis of a non-linear hybrid system by solving predicative encodings of an inductive stability argument and evaluate the impact of different methods and their combination.

1 Introduction

The formal analysis of hybrid systems usually involves steps of (ideally safely) approximating their behavior to obtain models that can be handled by available tools, since practical engineering models often incorporate elements that no verification tool can handle in combination. Each of these approximations may cause a loss of precision in the model, e.g. when capturing non-linear behavior by a linear model. At the same time, these approximations often have to be done manually, and worse, have to be repeated when the original model changes. We are therefore convinced that it is highly desirable to develop tools that can handle as rich dynamics as possible, and hence allow model checking of hybrid systems in a direct way. In this paper, we will not present a comprehensive tool that achieves this goal, but we show that our improvement of Satisfiability modulo ODE solving is a promising step into this direction, though still of academic nature in the size of problems solvable.

* This work has been supported by the German Research Council DFG within SFB/TR 14 "Automatic Verification and Analysis of Complex Systems" (www.avacs.org) and by the Natural Sciences and Engineering Research Council of Canada.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 172–187, 2011.

The underlying idea of hybrid system analysis by Satisfiability (SAT) modulo ODE solving is to offer a constraint language, plus the corresponding solvers, featuring as its atomic constraints exactly the equations and inequalities arising in hybrid-system models, especially algebraic constraints between variables and non-linear ODEs. With such an expressive constraint language, predicative encoding of hybrid system dynamics becomes straightforward, rendering intricate encodings and approximations superfluous. Starting from a predicative encoding of a hybrid system, the task of the solver is to prove the absence of or search for a satisfying valuation of the variables, which encode snapshots of the system's state at points in time, connected by the transition relation that encodes the behavior of the system. In the case of bounded model checking (BMC), satisfying valuations represent trajectories of the modeled system, starting from an initial state, performing a bounded number of transitions (jumps and flows) and finally leading to a target state satisfying a property of interest. The basic principle of SAT modulo ODE solving is to handle directly ODEs as part of a constraint system by evaluating their consistency under the current partial assignment the solver is investigating and learning implied facts for future search.

ODE enclosures as propagation mechanisms have been applied previously in Constraint Programming [6] for conjunctive Constraint Satisfaction Problems as well as by Ishii et. al. [8] in a traditional Satisfiability Modulo Theories (SMT) scheme. In contrast to such an integration (i.e., a SAT solver selecting which theory atoms shall be satisfied, interleaved with theory solvers evaluating this conjunction of atoms), the iSAT [5] algorithm performs a search by splitting intervals and hence indirectly ruling out those atoms that become inconsistent under this valuation, and thus deducing that other arithmetic constraints must be satisfied for satisfaction of the entire formula. These constraints then participate in the search by means of interval constraint propagation (ICP): as they have to be satisfied, interval valuations for their variables can be narrowed by pruning off subintervals that cannot contain a solution. Such ICP deductions are well-known for algebraic constraints and narrow the search space very effectively.

Reasoning about ODEs can be directly integrated into this framework [4] using methods for safe interval enclosures of solutions of ODEs. These methods compute an interval cover for the states reachable from an interval box of initial states. Since their effectiveness in narrowing the overall search space of the constraint solver depends on the tightness of the enclosures provided by these methods, we have reconsidered the tools used for generating such enclosures, now incorporating the ODE solver VNODE-LP [12] and combining it with a second layer of reasoning about ODEs, which is only applicable under certain side-conditions, but may yield tighter enclosures. This additional layer generates bracketing systems [16] for monotonic segments of trajectories, thus reducing the problem of computing the image of a set of initial states to one of computing bounding trajectories.

In this paper, we describe the resulting algorithm and evaluate it on a classical nonlinear hybrid system, thereby comparing different combinations of the ODE enclosure mechanisms. The evaluation covers deep unwindings of BMC problems,

as traditionally covered by SMT methods, as well as a novel temporal induction scheme able to prove a form of stability of hybrid systems.

The exposition starts with an overview of the iSAT algorithm and its interplay with ODE constraints in Section 2. Section 3 describes the VNODE-LP solver, Section 4 explains the bracketing systems approach, and Section 5 discusses deducing trajectory directions. Section 6 reports experimental results obtained on benchmarks, followed by the conclusions presented in Section 7.

2 The iSAT Algorithm for SAT Modulo ODE

In this section, we overview briefly the basic iSAT algorithm (for details cf. [5]) and focus on aspects related to the integration of ODE enclosures.

Problem statement. Let Φ be a quantifier-free Boolean combination of arithmetic constraints over bounded real-, integer-, and Boolean-valued variables, simple bounds, and ODE constraints over real variables with the following properties:

- arithmetic constraints over variables x, y, and z are of the form $x \sim \circ(y, z)$ or $x \sim \circ(y)$, where $\sim$ is a relational operator from $\{<, \leq, =, \geq, >\}$, and $\circ$ is a total unary or binary operator from $\{+, -, \cdot, \sin, \cos, \mathrm{pow}_{\mathbb{N}}, \exp, \min, \max\}$;
- simple bounds are of the form $x \sim c$ with $\sim$ as above a relational operator, x a variable, and c a constant; and
- ODE constraints are time invariant and given by $\dot{x}_i = dx_i/dt = f(x_1, \ldots, x_n)$ with all occurring variables x_i themselves being defined by ODE constraints and f being a function composed of $\{+, -, \cdot, /, \mathrm{pow}_{\mathbb{N}}, \exp, \ln, \sqrt{}, \sin, \cos\}$. These ODE constraints must occur only under an even number of negations in the formula, allowing e.g. $m_1 \rightarrow ((\dot{x} = \sin(y)) \wedge (\dot{y} = -x))$, but forbidding e.g. $(\dot{x} = \sin(y)) \rightarrow m_1$to avoid subtleties in the semantics of the formula.

Additionally, Φ and the variables therein have the structure

$$\Phi = \mathrm{decl}[0] \wedge \cdots \wedge \mathrm{decl}[k] \wedge \mathrm{init}[0] \wedge \mathrm{trans}[0, 1] \wedge \cdots \wedge \mathrm{trans}[(k-1), k] \wedge \mathrm{target}[k]$$

arising from the k-fold unwinding of the transition system, where $\mathrm{decl}[i]$ is the i-th instantiation of the system variables' domain bounds, $\mathrm{init}[0]$ is the predicative encoding of the initial state applied to the 0-th variable instance, i.e. to the beginning of the trace, $\mathrm{trans}[i, i+1]$ is the application of the transition predicate to the i-th and $(i+1)$-th instances of the variables, e.g. instantiating $a' = a + 1$ to $a[3] = a[2] + 1$, and $\mathrm{target}[k]$ is the application of the target predicate to the last variable instance. ODE constraints occur only within the transition relation since they constrain the continuous flow behavior of the system.

Example. To illustrate this input, Figure 1 shows an encoding of a model from [6]. The problem can be stated as follows: find two points A and B on a circle with radius 1 around $(1, 0)$ and from the box $[-1, 1] \times [-1, 1]$, such that a trajectory of a harmonic oscillator around $(0, 0)$ with fixed temporal length (here, we choose 1), starting in A ends in a point X, forming an equilateral triangle A, B, X.

```
DECL
    float [−1, 1] ax, ay, bx, by;
    float [−10, 10] x, y;
    float [0, 10] time;
    float [1, 1] delta_time;
INIT
    −− A and B on circle around (1,0).
    (ay − 0)^2 + (ax − 1)^2 = 1^2;
    (by − 0)^2 + (bx − 1)^2 = 1^2;
    −− A and B must be distinct points.
    ax != bx or ay != by;
    −− Trajectory must start in A.
    x = ax; y = ay; time = 0;
TRANS
    −− A and B stay the same.
    ax' = ax; ay' = ay;
    bx' = bx; by' = by;
    −− Trajectory.
    (d.x / d.time = y);
    (d.y / d.time = −x);
    time' = time + delta_time;
```

```
TARGET
    −− Equilateral triangle: equal
    −− distances between A, B, and X.
      (ay − by)^2 + (ax − bx)^2
    = (ax − x)^2 + (ay − y)^2;
      (ay − by)^2 + (ax − bx)^2
    = (bx − x)^2 + (by − y)^2;
```

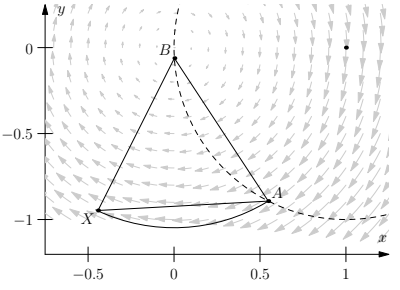


Fig. 1. Illustration of the solver input (before being automatically rewritten into the solver's internal format by its frontend) and a possible solution found by iSAT

Satisfiability. As usual, a valuation σ, which maps each variable to a point from its domain, satisfies Φ iff the constraints satisfied under σ satisfy the Boolean structure of Φ. Satisfiability is straightforward for simple bounds and arithmetic constraints, but requires some explanation in the case of ODE constraints.

As noted above, ODEs —describing the evolution of variables over continuous time— occur only in the transition relation, which constrains the pre-post relation between any two successive instances of variables in a trace. Semantically, a trace is a sequence of snapshots of a real-time trajectory of the hybrid system. Hence, ODE constraints describe the behavior of the system between two such snapshots, i.e. describe trajectories emerging from the pre-valuation, following the dynamics described by the ODE, and finally reaching the post-valuation. A valuation σ thus satisfies a definitionally closed system of ODE constraints (each occurring variable itself being defined by one of the component ODE constraints), iff there exists a solution trajectory starting with the pre-valuation and ending with the post-valuation after a duration equal to the temporal length of the flow, as provided by the value of a special variable `delta_time`.

More formally, given $\vec{x} = (x_1, \ldots, x_n)^T$ and ODE constraints defining $\vec{x}$:[1]

$$\dot{\vec{x}} = \big(f_1(\vec{x}), \ldots, f_n(\vec{x})\big)^T = \vec{f}(\vec{x}), \tag{1}$$

for two BMC unwinding depths i and $i+1$, the instantiations of $\vec{x}$ are given by $\vec{x}[i]$ and $\vec{x}[i+1]$, and their valuations $\sigma(\vec{x}[i])$ and $\sigma(\vec{x}[i+1])$[2] together with $\tau := \sigma(\texttt{delta\_time}[i])$ satisfy (1) iff there exists a solution function $\vec{y} : [0, \tau] \to \mathrm{dom}(\vec{x})$ such that $\vec{y}(0) = \sigma(\vec{x}[i])$, $\dot{\vec{y}}(t) = \vec{f}(\vec{y}(t))$ for all $t \in [0, \tau]$, and $\vec{y}(\tau) = \sigma(\vec{x}[i+1])$.

[1] We use explicit vector notation only where confusion with simple variables from a formula might otherwise occur.

[2] For simplicity, the valuation of a vector shall be the vector of its valuations.

Flow invariants. Currently, we do not support direct encoding of *mode* or *flow invariants*, i.e. of constraints on the states traversed during a continuous evolution. Such invariants can only be formulated within the pre-post relation. If in the example in Fig. 1, y should stay $\geq c$, we could add constraints like $y \geq c \wedge y' \geq c$ to the transition system. While for monotonic solutions, no additional behavior would be allowed by this notation, for the example system, the direction may change and thus a trajectory may start and end above c, satisfying the added constraint, but violating the flow invariant at a point of time in between. The constraint system would thus be an overapproximation of the original system, allowing spurious trajectories that can not be always removed.

Solving. The task of the solver is to find a valuation satisfying the formula or proving its unsatisfiability. Starting from an input formula like the one depicted in Fig. 1, a preprocessing step (see [5] for more details) introduces auxiliary variables to split complex arithmetic expressions into the format described above and to simplify the Boolean structure into a conjunction of clauses, which are themselves disjunctions of arithmetic atoms, simple bounds, and trigger variables representing ODE constraints. The latter are stored separately and are activated whenever their respective trigger variable becomes true.

Instead of point-valued valuations, the iSAT algorithm interprets the constraints over intervals. Initially, each variable receives its whole domain as an interval valuation. Akin to DPLL-based SAT solving [2,3], the three main ingredients of the solver are deduction, decision, and conflict resolution. However, constraints cannot only be satisfied or unsatisfied for all valuations from the interval box, but also contain a mixture of points satisfying or violating a constraint. For example, consider constraint $C: \; x = 2 \cdot y$ under the interval valuation $x \in [0, 10]$, $y \in [3, 6]$. No point with $x \in [0, 6)$ or $y \in (5, 6]$ satisfies C, while $x \in [6, 10], y \in [3, 5]$ contains points (x, y) like $(6, 3)$ satisfying and points like $(6.1, 3)$ violating C.

Clauses (disjunctions of constraints) that contain only one constraint that is potentially satisfiable under the current valuation are called *unit* and give rise to *unit propagation*: the last satisfiable constraint in a clause else containing only violated constraints must be propagated to retain a chance for satisfiability of the conjunction of all clauses. If the above example constraint c were such a last remaining atom of a clause, then *interval constraint propagation* would allow to prune away those ranges above identified as not containing any solutions, yielding a new valuation $x \in [6, 10], y \in [3, 5]$ and thus a reduced search space.

When no more propagations are possible or the newly deduced bounds have negligible progress with respect to the old ones, a *decision* is performed by selecting heuristically a variable and splitting its interval, i.e. introducing a new upper or lower bound at its midpoint. This bound may give rise to new deductions. If all of a clause's constraints are violated under the current valuation, e.g. due to a prior propagation step, a *conflict* is encountered, which is resolved by analyzing the reasons that caused it and generating a *conflict clause* that is a disjunction of the negated reasons. This clause is added to the formula and

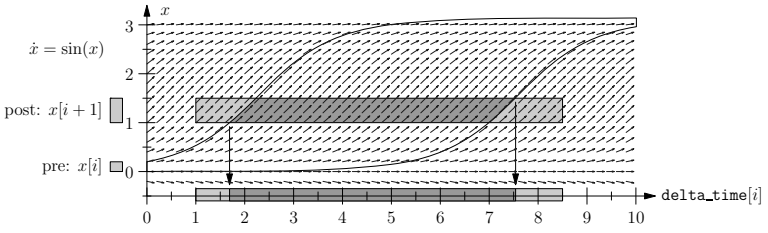


Fig. 2. An ODE deduction which allows to propagate tighter bounds for `delta_time`$[i]$

forces at least one of the offending bounds to be chosen differently in the future, effectively removing this part of the search space for the remainder of the search.

Termination. If the solver encounters a conflict from which it cannot recover, because no undoing of decisions would resolve it, it has successfully proven *unsatisfiability*. Due to the safe overapproximations used in all propagations (e.g. outward rounding for arithmetic evaluations) and always pruning non-solutions only, this unsatisfiability result is safe. The solver terminates with *unknown*, if it encounters a box whose maximum width is below a small, user-defined threshold and for which deduction cannot show inconsistency. This small box is a candidate solution box, which merits practical attention when encountered as a potential counter example to the safety of an engineered system. As the reported candidate solution boxes are very small, interval Newton methods may be able to verify that they contain an actual solution. While our algorithm currently does not contain such a check, Ishii et al. [8] have implemented it.

Deduction for ODE constraints. Having interval valuations for the variable instances occurring in ODEs, again requires lifting their original point-valued interpretation to intervals. For arithmetic constraints, we prune away only parts not containing any solutions. The very same idea applied to ODEs means that we may prune away all those points from the post-valuation that are not (forward) reachable when starting a trajectory from any point in the pre-valuation and staying on it for any duration contained in the interval valuation of the respective `delta_time` variable. Analogously, we can safely prune away those parts of the pre-valuation for which no trajectory can reach any point in the post-valuation with any of the possible durations (backward propagation). In addition, time points t from `delta_time` can be pruned when no trajectory starting from the pre-valuation reaches any point from the post-valuation at t (cf. Figure 2).

The essential ingredient in the deduction for ODE constraints is thus a method to safely enclose over a temporal interval all trajectories emerging from the pre-valuation, which is typically an interval box. While our original integration of such an ODE enclosure mechanism into the iSAT algorithm [4] was confined to embedding a relatively weak own implementation of a Taylor-series-based safe integrator, we base our current approach on VNODE-LP [12].

ODE deductions are performed in strict alternation with the other deduction mechanisms. After completing Boolean and interval constraint propagation as

described above, iSAT's ODE solving layer uses the current valuation of the trigger variables for each instance of the transition system to select the active ODE constraints. This signature of activated ODEs and the current interval valuation for the occurring variables together suffice to generate an enclosure. In contrast to normal deductions, whose results are stored only temporarily until they may be undone later by a backjump when recovering from a conflict, the results of ODE deductions are stored in clauses. This technique, similar to conflict clause learning, ensures that the same deduction does not have to be repeated since its results have been added persistently to the formula. Similarly to constraints replication [19], we add copies of the learned clauses for all isomorphic variable instances arising from the k-fold unwinding of the transition relation.

Before performing an ODE deduction, the algorithm checks whether the same query has been encountered before and rejects all duplicate queries. A second level of caching holds a limited number of intermediate results, which can be reused when enclosures for a subbox of the original box are requested since interval arithmetic's monotonicity property w.r.t.. set inclusion guarantees then that they are still valid (yet coarse) enclosures also for the current valuation. Using a stored solver run, whenever the currently examined valuation is only slightly smaller than the original box, partially avoids recomputations. Since the bounds deduced by the ODE solver are subsequently used in interval propagations, it is very likely to encounter kind of slightly changed query, providing this caching layer with a significant role in avoiding wasted computations.

Soundness. The correctness of the core algorithm has been detailed in [5]. Since our extension to deductions for ODE constraints is restricted to the pruning of non-solutions and storing all reasons involved in these deductions explicitly in the learned clauses, the same arguments hold here, too. An essential ingredient to soundness is the use of *validated computations*, i.e. outward rounding for interval computations, interval evaluation of remainder terms to capture truncation errors for the numerical enclosure method detailed in the following section, and detection of overflows during these computations. Technically, many of these issues are delegated to libraries, in our case the MPFR and filib++ libraries.[3]

3 Overview of VNODE-LP

In this section, we present an overview of VNODE-LP, Validated Numerical ODE through Literate Programming. More details can be found in [12,13].

Consider the initial-value problem, IVP (we omit the $\vec{\cdot}$ notation),

$$\dot{x}(t) = f(t, x), \quad x(t_0) = x_0, \qquad t \in \mathbb{R},\ x \in \mathbb{R}^n, \tag{2}$$

where $f : \mathbb{R} \times \mathbb{R}^n$ is sufficiently smooth (as a consequence, the code list of f should not contain e.g. branches, abs, or min).

[3] http://www.mpfr.org/ and http://www2.math.uni-wuppertal.de/~xsc/software/filib.html.

Denote the set of n-dimensional interval vectors by $\mathbb{IR}^n$. Given $\boldsymbol{x}_0 \in \mathbb{IR}^n$ and $t_{\text{end}} \neq t_0$ ($t_{\text{end}} \in \mathbb{R}$), VNODE-LP tries to compute an $\boldsymbol{x}_{\text{end}} \in \mathbb{IR}^n$ at t_{end} that contains the solution to (2) at t_{end} for all $x_0 \in \boldsymbol{x}_0$. If VNODE-LP cannot reach t_{end}, for example the bounds on the solution become too wide, bounds at some t^* between t_0 and t_{end} are returned.

This solver proceeds in a one-step manner from t_0 to t_{end}, where it computes bounds at (adaptively) selected points $t_j \in (t_0, t_{\text{end}}]$. To explain an integration step, denote by $x(t_j; t_0, x_0)$ the solution to (2) with an initial condition x_0 at t_0, and denote by $\boldsymbol{x}_j$ an enclosure of this solution at t_j. That is,

$$x(t_j; t_0, x_0) \in \boldsymbol{x}_j \quad \text{for all } x_0 \in \boldsymbol{x}_0.$$

On a step from t_j to t_{j+1}, VNODE-LP computes first a priori bounds $\widetilde{\boldsymbol{x}}_j$ such that $x(t; t_j, x_j) \in \widetilde{\boldsymbol{x}}_j$ for all $t \in [t_j, t_{j+1}]$ and all $x_j \in \boldsymbol{x}_j$. Then it finds tight bounds $\boldsymbol{x}_{j+1}$ at t_{j+1} such that $x(t_{j+1}; t_0, x_0) \in \boldsymbol{x}_{j+1}$ for all $x_0 \in \boldsymbol{x}_0$. For an illustration of a priori and tight bounds, see Fig. 3. To compute these bounds, we use interval arithmetic, Taylor series expansion of the solution to (2) at each integration point, and various interval techniques; for more details see [12,14].

VNODE-LP is based on Taylor series and the Hermite-Obreschkoff [14] methods. It is a fixed-order, variable-stepsize solver. The stepsize is varied such that an estimate of the *local excess* per unit step is below a user-specified tolerance. Typically efficient values for the order can be between 20 (default) and 30 [12].

Generally, VNODE-LP is suitable for computing bounds on the solution of an IVP ODE with point initial conditions or interval initial conditions with a sufficiently small width. If the initial condition set is not small enough and/or long time integration is desired, the COSY package [1] of Berz and Makino can produce tighter bounds than VNODE-LP. Alternatively, one can subdivide the initial interval vector (box) $\boldsymbol{y}_0$ into smaller boxes, perform integrations with them as initial conditions, and build an enclosure of the solution at t_{end}.

The COSY package bounds the solution using Taylor models, which consist of a high-order Taylor polynomial in the initial conditions plus an enclosure of the remainder term. On each integration step, such polynomial representations of the bounds are propagated, thus effectively reducing the wrapping effect. In contrast, VNODE-LP, expands the solution with respect to initial condition up to first order, and propagates parallelepipeds enclosing the solution, which are generally less effective for reducing the wrapping effect. However, COSY is computationally more expensive than VNODE-LP.

On each step from t_j to t_{j+1}, iSAT uses the a priori bounds and also computes tighter bounds over selected subintervals of $[t_j, t_{j+1}]$, in addition to the provided tight bounds by VNODE-LP at t_{j+1}, by calling VNODE-LP with initial point t_j and the interval to be refined as interval ending $\boldsymbol{t}_{\text{end}} \subset [t_j, t_{j+1}]$. These bounds are not computed efficiently by VNODE-LP, as currently it does not provide a facility for evaluating a representation of the solution between integration points; that is, a facility similar to having a continuous interpolant in standard ODE solving. Such a feature is presently being implemented.

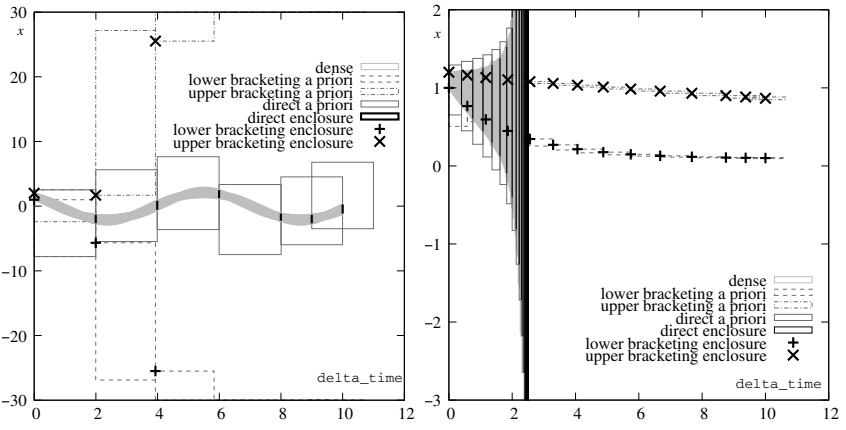


Fig. 3. Comparison of direct and bracketing enclosure. Left: x dimension of a harmonic oscillator $\dot{x} = y, \dot{y} = -x, x(0), y(0) \in [1, 2]$. Right: x dimension of $\dot{x} = -p_4x-(p_1x)/(1+p_2y)+p_3y+0.1)$, $\dot{y} = p_4x-p_3y$, all $\dot{p}_i = 0$, for $x(0) \in [1, 1.2]$, $y(0) \in [0.8, 1]$, $p_1 \in [0.8, 1]$, $p_2 \in [1.0, 1.2]$, $p_3 \in [0.3, 0.5]$, and $p_4 \in [0.20, 0.25]$. Dense enclosures have been obtained by direct application of VNODE-LP with small fixed stepsize.

4 Using Bracketing Systems as Enclosures

When the starting point of the IVP (2) is a wide interval vector, the enclosures returned by VNODE-LP may diverge after a few computation steps. One way to address this shortcoming, while deriving guaranteed results, is to use the bracketing approach introduced in [16,17], which relies on the classical Müller's existence theorem [11,10].

Given the IVP (2), the bracketing method analyzes the signs of the partial derivatives $\partial f_i/\partial x_l$, evaluated over the enclosure for all $t \in [t_j, t_{j+1}]$.

(i) Over each time interval $[t_j, t_{j+1}]$, where these signs remain constant, the method builds two dynamical systems that enclose the original uncertain dynamical system and thus bound the flow pipe between a minimal solution, i.e. a flow that is always lower than the solution flow pipe, and a maximal solution that is always larger. Since this bracketing system involves no more uncertainty, VNODE-LP can be efficiently used for the guaranteed computation of the minimal and maximal solutions, which start as points instead of intervals. Hence, the solution enclosure of the actual IVP is enclosed between a minimal and a maximal solution, obtained as the solution of a new system of coupled ODEs.

(ii) Over each time interval $[t_j, t_{j+1}]$, where the sign of at least one partial derivative changes, we merely use VNODE-LP on the original IVP.

In our implementation, the signs of the partial derivatives need not be analyzed over the enclosure set for all $t \in [t_j, t_{j+1}]$, but are only analyzed over $\boldsymbol{x}_j$, the tight enclosure at t_j. Once the bracketing systems are built and the solution set computed over the whole time interval, these signs are then checked a posteriori: if they remain constant for all $t \in [t_j, t_{j+1}]$, then it is proven that

the bracketing systems are valid [16], if not, then the bracketing systems are not valid over whole time interval. In this case the solution is enclosed using VNODE-LP on the original IVP with interval initial conditions.

Furthermore, our implementation of the bracketing approach is novel. Indeed, the bracketing systems are built automatically on the fly inside iSAT. This is done through the FADBAD++[4] automatic differentiation package, whereas previously they were built manually or using external symbolic algebra.

Figure 3 compares enclosures obtained using our implementation of the bracketing approach and the direct application of VNODE-LP. Clearly, both methods should be combined as their actual performances depend on the analyzed ODE. The performance of the bracketing approach, that is how tight are the computed enclosures when used with a given system, may in fact be known a priori. For monotone dynamical systems, those whose flows preserve a suitable partial ordering on states, hence on initial conditions, the computed bracketing systems are feasible instantiations of the dynamical system under study, hence exhibit the same convergence and stability properties as the original system. If the latter is convergent and stable, then should the bracketing systems. However, when the dynamical system is not a monotone one, the bracketing systems usually suffer from a hidden wrapping effect that provokes the derived enclosures to blow up. In spite of that, both experimental and theoretical evaluation show that when the original system exhibits very strong convergence (stability) properties, the latter property can overrule the wrapping effect making the bracketing approach effective. Finally, the bracketing approach performs badly when the system exhibits stable orbits or oscillatory behaviors. Nevertheless, we expect our implementation of bracketing systems within iSAT to simplify the thorough practical assessment of its actual performance in the future.

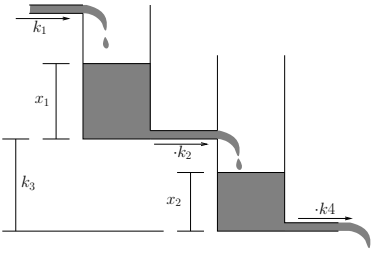
5 Deducing Trajectory Directions

In the case study shown in the following section, we encounter the problem of showing that a trajectory cannot stay at the point of its origin when at least an infinitesimal amount of time (`delta_time` > 0) has been spent. The enclosure schemes presented so far —powerful as they are— are unable to prove this. One reason for this is that even for point-valued initial conditions $\boldsymbol{x}_0$, the very first a priori enclosure for an interval $t \in (0, t_1]$ must also contain the enclosure $\boldsymbol{x}_0$ itself, since the solution trajectory is a continuous function.

The simple yet effective solution to this problem is to evaluate the ODEs' right-hand sides over a prefix `delta_time` $\in [0, t_p]$ of the already calculated enclosure. If this evaluation yields a strictly positive result, we can safely deduce `delta_time` $\in (0, t_p] \Rightarrow x' > x$, i.e. that the post-value is strictly greater than the pre-value for this prefix. Analogously, we can deduce `delta_time` $\in (0, t_p] \Rightarrow x' < x$, if the evaluation yields only values strictly less than zero.

A *direction deduction* performs an interval evaluation of the ODE's right-hand side over the first enclosure step and continues this computation for subsequent

[4] http://www.fadbad.com



For $x_2 > k_3$:

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}k_1 - k_2\sqrt{x_1 - x_2 + k_3}\\ k_2\sqrt{x_1 - x_2 + k_3} - k_4\sqrt{x_2}\end{pmatrix}$$

For $x_2 \leq k_3$:

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}k_1 - k_2\sqrt{x_1}\\ k_2\sqrt{x_1} - k_4\sqrt{x_2}\end{pmatrix}$$

Fig. 4. Structure and dynamics of the two tank hybrid system (from [20])

steps, as long as the calculated intervals do not contain zero. The upper bound of t_p is then at the end of either the entire enclosure or the last enclosure step for which the evaluation yielded a strictly positive or negative result.

6 Experiments

To evaluate the integrated tool and the influence of the different enclosure methods, we apply our solver to the two-tank model from [20], which has been frequently used as a case study for verification tools cf. e.g. [7,18]. This system comprises two tanks connected by a tube. The first tank has an inflow of constantly $k_1 = 0.75$ volume units, and its base is $k_3 = 0.5$ length units above the base of the second tank. The connecting tube is characterized by a constant factor $k_2 = 1$, which also characterizes the outflow of the system as $k_4 = 1$.

Figure 4 illustrates this setting and formalizes the dynamic behavior of the liquid's height x_1 and x_2 in the two tanks. The system's behavior switches between two dynamics, when x_2 reaches the outlet from tank 1 and therefore exerts a counter pressure against the incoming flow. Note that the model is implicitly bounded to the case that $x_2 \leq x_1 + k_3$, since it does not provide the dynamics for the inverse direction. To understand better the dynamics of this system and the proof obligations we encoded, Fig. 5 depicts simulated trajectories.

Similar to the introductory example in Fig. 1, we encode this model predictively using the above description directly as ODE constraints.[5]

Bounded reachability. To validate the model, we first check bounded reachability properties. As can be assumed from Fig. 5, there should not be any trajectory leading from region $D = [0.70, 0.80] \times [0.45, 0.50]$ to $E = [0.45, 0.50] \times [0.60, 0.65]$. This property has been verified by Henzinger et. al. using HyperTech [7].

We restrict the global time ≤ 100 and each step duration `delta_time` ≤ 10. To avoid unnecessary non-determinism in the model, all steps are explicitly enforced in the transition relation to take the maximum possible duration. They may be shorter only if they reach the switching surface at $x_2 = k_3$, if the time $= 100$, or if (x_1, x_2) reaches E. Our solver can prove unsatisfiability of this bounded property for up to 300 unwindings of the transition system within 3109.1 seconds.

[5] http://www.avacs.org/fileadmin/Benchmarks/Open/iSAT_ODE_SEFM_2011_models.tar.gz

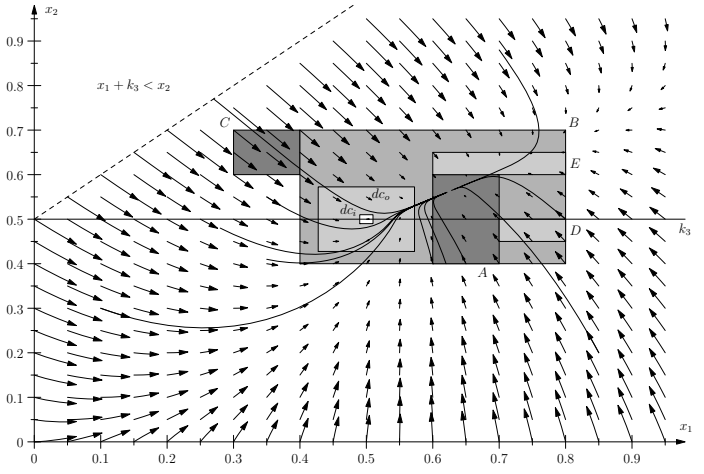


Fig. 5. Simulated trajectories for the two tanks system, inner and outer bounds of the don't care mode, and regions A - E used in the different verification conditions

Table 1 summarizes the runtimes on a 2.4 GHz AMD Opteron machine, which has been used for all runtime measurements. The solver is set to continue until it finds a solution and to keep learned clauses of previous BMC steps in the formula. The runtimes clearly indicate that the bulk of the problem lies in refuting the possibility of a trajectory with a low number of steps, while adding more unwindings of the formula does not make this problem harder to solve.

Unbounded trajectory containment. Although the formula structure is a bounded unwinding of the transition system, inductive arguments may be used to prove unbounded properties. One can easily see that region $A = [0.6, 0.7] \times [0.4, 0.6]$ contains an equilibrium point. However, the simulation also shows that there are trajectories leaving this region. We extend our model to show that trajectories can leave region A only on a bounded prefix, but thereafter stay in A forever.

First, we guess a $\tau > 0$ (supported by looking at some simulated trajectories). With $\mathcal{M}_l := \{\text{all trajectories of length} \geq l\}$, from showing that

$$\forall \vec{x} \in \mathcal{M}_{2\tau} : [0, 2\tau] \to \mathbb{R}^2 : (\vec{x}(0) \in A \Rightarrow \forall t \in [\tau, 2\tau] : \vec{x}(t) \in A) \tag{3}$$

follows by inductive application of (3), as facilitated by time invariance,

$$\Rightarrow \forall \vec{x} \in \mathcal{M}_{\infty} : [0, \infty) \to \mathbb{R}^2 : (\vec{x}(0) \in A \Rightarrow \forall t \in [\tau, \infty) : \vec{x}(t) \in A)$$

Table 1. CPU time (seconds) for the individual unwinding depths of the bounded reachability check from region D to E

depth k	1	2	3	4	5	6	7	8	9	10	...	100	200	300
time	6.6	183.7	344.3	268.8	167.4	15.7	12.7	16.5	10.3	3.7		3.8	7.9	15.0
total	6.6	190.3	534.5	803.4	970.7	986.4	999.1	1015.6	1025.8	1029.5		1308.9	1955.2	3109.1

Table 2. Column *all* shows results and CPU times (seconds) for checking unbounded containment in A using all enclosure methods combined. In the subsequent columns, one of the methods is disabled

depth	all	no bracketing	no direct	no direction
1	unknown, 111.9	unknown, 42.0	unknown, 61.5	unknown, 111.5
2	unknown, 467.5	unknown, 981.0	unknown, 346.3	unknown, 342.0
3	**UNSAT**, 674.0	**UNSAT**, 5011.6	**UNSAT**, 404.2	unknown, 478.8
4	**UNSAT**, 812.1	**UNSAT**, 1995.1	**UNSAT**, 499.1	unknown, 547.5
5	**UNSAT**, 986.0	**UNSAT**, 2432.0	**UNSAT**, 601.1	unknown, 682.4
6	**UNSAT**, 1126.1	**UNSAT**, 3303.4	**UNSAT**, 705.0	unknown, 834.2
7	**UNSAT**, 1277.2	**UNSAT**, 2486.8	**UNSAT**, 803.7	unknown, 982.5
8	**UNSAT**, 1451.4	**UNSAT**, 5273.3	**UNSAT**, 890.8	unknown, 1115.7
9	**UNSAT**, 1584.6	**UNSAT**, 4905.2	**UNSAT**, 966.5	unknown, 1235.8
10	**UNSAT**, 1706.6	**UNSAT**, 6396.1	**UNSAT**, 1053.2	unknown, 1356.0

Intuitively, we show that all trajectories of length 2τ stay in A for $\texttt{delta\_time} \in [\tau, 2\tau]$ (ignoring their behavior for $[0, \tau)$). All unbounded trajectories must have these trajectories of length 2τ as prefix. At τ, they are thus (again) in A. Due to time invariance, we can consider $(x_1, x_2)(\tau)$ as a new starting point. Since it lies in A, we have already proven that for $[\tau + \tau, \tau + 2\tau]$, the trajectory will lie in A again. For the time in between, we already know that it is in A. By repeating this process ad infinitum, we know that the trajectory can never leave A again.

Note that this proof is related to the idea of *region stability* [15] and can be thought of as a stabilization proof for an unknown (and maybe hard to characterize) sub-region $A_{\text{inv}} \subseteq A$ into which all trajectories from A stabilize, and which is an invariant region for the system.

Table 2 summarizes runtimes for this proof using iSAT and the different enclosure methods. Our model encodes the above proof scheme in the following way: if a trajectory exists that is shorter than 2τ or that reaches a point outside A in time $\in [\tau, 2\tau]$, this trajectory satisfies the model. The proof is successful when the solver finds an unwinding depth k of the transition system upon which the model becomes *unsatisfiable*. Here, an unwinding depth of 3 suffices to prove the desired property. Without the direction deduction presented in Sect. 5, the solver fails to prove unsatisfiability, because it always finds counter examples that stay on the switching surface, spending there only tiny amounts of time. These trajectories satisfy the target condition of having time $\leq 2\tau$ and do not allow proving (3). Direction deduction hence enables proving the property.

The runtimes show that the approach without the direct enclosure (using only bracketing enclosures and direction deductions) outperforms both, the restriction to the direct usage of VNODE-LP with direction deduction and the combination of all enclosure methods together on this benchmark.

Introducing artificial non-determinism and hysteresis. Trying a direct inductive proof for the region $B = [0.4, 0.8] \times [0.4, 0.7]$ (i.e. showing that B cannot be left with one step of the transition system) fails with our tool since B's corner at $(0.4, 0.4)$ cannot be represented exactly by floating-point numbers.

Table 3. Results and CPU times (seconds) for checking unbounded containment in B

depth	all	no bracketing	no direct	no direction
1	unknown, 17.7	unknown, 9.4	unknown, 12.9	unknown, 15.4
2	unknown, 163.9	unknown, 57.9	unknown, 81.9	unknown, 157.4
3	unknown, 198.9	unknown, 71.8	unknown, 126.9	unknown, 202.4
4	unknown, 666.6	unknown, 193.6	unknown, 146.7	unknown, 206.9
5	**UNSAT**, 2334.2	**UNSAT**, 3270.3	unknown, 183.4	unknown, 283.6
6	**UNSAT**, 4615.6	**UNSAT**, 1441.2	unknown, 182.2	unknown, 122.0
7	**UNSAT**, 2967.1	unknown, 1934.7	unknown, 144.1	unknown, 123.9
8	**UNSAT**, 2559.0	**UNSAT**, 2953.0	unknown, 201.6	unknown, 123.6
9	**UNSAT**, 2184.1	**UNSAT**, 4121.2	unknown, 135.2	unknown, 127.2
10	**UNSAT**, 5541.6	**UNSAT**, 7717.3	unknown, 272.5	unknown, 127.6

To compensate, B is overapproximated to capture rounding errors, hence includes points that lie slightly outside B. Using the same proof scheme as above can be expected to work, as the simulated trajectories point inwards from the border of B. Yet, applying this proof scheme, the solver finds trajectories that can chatter indefinitely at $P = (0.5, 0.5)$, since $\dot{x}_2 = 0$ in P. This chattering is a valid behavior, though irrelevant for the actually intended proof of B's invariance.

We therefore identify intersections of the switching surface with $\dot{x}_2 = 0$ (i.e. solutions to the constraint system $k_2\sqrt{x_1} - k_4\sqrt{x_2} = 0 \wedge x_2 = k_3$) and, finding only this one in P, add a *don't-care mode* around it —depicted in Fig. 5 as $dc_i = [0.49, 0.51] \times [0.49, 0.51]$. Since this region lies well inside B, we allow any trajectory that reaches it to jump immediately or after an arbitrary positive amount of time to the outer border of the don't-care mode, illustrated by dc_o, which is $\varepsilon = 0.0625$ away from dc_i. We also forbid any trajectory to enter dc_i. This modification trades in accuracy by introducing non-determinism for the benefit of an artificial hysteresis: trajectories which could formerly stutter in P can now jump to any point on the border of dc_o, but must then move along the system's dynamics again, consuming time.

With this modification, we can prove that B is left for less than $\tau = 0.0625$. Table 3 shows that the proof succeeds for depths $k \geq 5$ for all methods combined. Though bracketing enclosures are computed successfully, the direct method generates at least one deduction which is essential to prove unsatisfiability.

Further evaluation. We also applied the same proof scheme to region $C = [0.3, 0.4] \times [0.6, 0.7]$ again with unwinding depths 1 to 10. As expected, none of the resulting formulae was proven unsatisfiable. Runtimes were within 20.3 seconds for unwinding depth 1 without bracketing system usage and 617.6 seconds for unwinding depth 10 with all methods used in combination.

7 Conclusion

After exploring the feasibility of using ODE enclosures to solve SAT modulo ODE problems in [4], this paper extends and improves the abilities of the resulting

solver by combining enclosure methods. We have shown that the techniques presented in this paper have complementary strengths, and that our integrated approach is capable of handling different types of proof obligations for a nonlinear hybrid system. Our improvements are orthogonal to the application of interval Newton contractors in [6,8], and could be extended in the same way to gain the ability to prove existence of solutions.

One current weakness of our method is its inability to express directly *flow invariants*, which constrain variables over the entire duration of a flow. The resulting formula may thus have solutions that are spurious trajectories in terms of the original model. Our experiments show that proofs can be successfully obtained in spite of this overapproximation. However, a direct handling of flow invariants would remove the need to counteract such spurious trajectories.

Ishii et al. handle this issue in [9] by selecting the "first" intersection of an enclosure with a guard condition. However, they discard an enclosure if it contains the initial value set under the assumption that this initial point and the next intersection with the guard are distinct. It is unclear whether this suffices to guarantee that the first intersection of a trajectory (after its starting point) is chosen. One focus of our future work will be to handle flow invariants by pruning the enclosures directly.

To accelerate our tool, we plan on extending VNODE-LP to produce enclosures over intervals of time by allowing re-evaluations of the Taylor series between computed steps, which will be significantly faster than the current evaluation scheme. Little effort has so far been invested in good decision heuristics to select likely solutions earlier in the search. We will also explore ways to build the bracketing systems when off-diagonal Jacobian elements change sign.

Acknowledgment. We would like to thank Stefan Ratschan, Christian Herde, Tino Teige, Jens Oehlerking, and Corina Mitrohin for discussions on the region-stability-related proof scheme utilized for the experiments in this paper and our other colleagues from AVACS H1/2 for the joint development of the iSAT core. Additionally, we are grateful to the reviewers for their detailed comments.

References

1. Berz, M.: COSY INFINITY version 8 reference manual. Tech. Rep. MSUCL–1088, National Superconducting Cyclotron Lab., Michigan State University, USA (1997)
2. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Commun. ACM 5, 394–397 (1962)
3. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)
4. Eggers, A., Fränzle, M., Herde, C.: SAT modulo ODE: A direct SAT approach to hybrid systems. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 171–185. Springer, Heidelberg (2008)
5. Fränzle, M., Herde, C., Ratschan, S., Schubert, T., Teige, T.: Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. JSAT Special Issue on Constraint Programming and SAT 1(3-4), 209–236 (2007)

6. Goldsztejn, A., Mullier, O., Eveillard, D., Hosobe, H.: Including ordinary differential equations based constraints in the standard CP framework. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 221–235. Springer, Heidelberg (2010)
7. Henzinger, T., Horowitz, B., Majumdar, R., Wong-Toi, H.: Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In: Lynch, N., Krogh, B. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 130–144. Springer, Heidelberg (2000)
8. Ishii, D., Ueda, K., Hosobe, H.: An interval-based SAT modulo ODE solver for model checking nonlinear hybrid systems. International Journal on Software Tools for Technology Transfer (STTT), 1–13 (March 2011)
9. Ishii, D., Ueda, K., Hosobe, H., Goldsztejn, A.: Interval-based solving of hybrid constraint systems. In: Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems, pp. 144–149 (2009)
10. Kieffer, M., Walter, E., Simeonov, I.: Guaranteed nonlinear parameter estimation for continuous-time dynamical models. In: Proceedings 14th IFAC Symposium on System Identification, Newcastle, Aus, pp. 843–848 (2006)
11. Müller, M.: Über das Fundamentaltheorem in der Theorie der gewöhnlichen Differentialgleichungen. Mathematische Zeitschrift 26, 619–645 (1927)
12. Nedialkov, N.S.: VNODE-LP — a validated solver for initial value problems in ordinary differential equations. Tech. Rep. CAS-06-06-NN, Department of Computing and Software, McMaster University, Hamilton, Ontario, L8S 4K1 (2006), VNODE-LP http://www.cas.mcmaster.ca/~nedialk/vnodelp
13. Nedialkov, N.S.: Implementing a rigorous ODE solver through literate programming. In: Rauh, A., Auer, E. (eds.) Modeling, Design, and Simulation of Systems with Uncertainties, Mathematical Engineering, vol. 3, pp. 3–19. Springer, Heidelberg (2011)
14. Nedialkov, N.S.: Computing Rigorous Bounds on the Solution of an Initial Value Problem for an Ordinary Differential Equation. Ph.D. thesis, Department of Computer Science, University of Toronto, Toronto, Canada, M5S 3G4 (February 1999)
15. Podelski, A., Wagner, S.: Region stability proofs for hybrid systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 320–335. Springer, Heidelberg (2007)
16. Ramdani, N., Meslem, N., Candau, Y.: A hybrid bounding method for computing an over-approximation for the reachable space of uncertain nonlinear systems. IEEE Transactions on Automatic Control 54(10), 2352–2364 (2009)
17. Ramdani, N., Meslem, N., Candau, Y.: Computing reachable sets for uncertain nonlinear monotone systems. Nonlinear Analysis: Hybrid Systems 4(2), 263–278 (2010)
18. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. ACM Transactions in Embedded Computing Systems 6(1) (2007)
19. Shtrichman, O.: Tuning SAT checkers for bounded model checking. In: Emerson, E., Sistla, A. (eds.) CAV 2000. LNCS, vol. 1855, pp. 480–494. Springer, Heidelberg (2000)
20. Stursberg, O., Kowalewski, S., Hoffmann, I., Preußig, J.: Comparing timed and hybrid automata as approximations of continuous systems. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1996. LNCS, vol. 1273, pp. 361–377. Springer, Heidelberg (1997)

Verification of B$^+$ Trees: An Experiment Combining Shape Analysis and Interactive Theorem Proving

Gidon Ernst, Gerhard Schellhorn, and Wolfgang Reif

University of Augsburg, Germany
{ernst,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. Interactive proofs of correctness of pointer-manipulating programs tend to be difficult. We propose an approach that integrates shape analysis and interactive theorem proving, namely TVLA and KIV. The approach uses shape analysis to automatically discharge proof obligations for various data structure properties, such as "acyclicity". We verify the main operations of B$^+$ trees by decomposition of the problem into three layers. At the top level is an interactive proof of the main recursive procedures. The actual modifications of the data structure are verified with shape analysis. To this purpose we define a mapping of typed algebraic heaps to TVLA. TVLA itself relies on various constraints and lemmas, that were proven in KIV as a foundation for an overall correct analysis.

1 Introduction

Interactive theorem provers are powerful tools for formal verification. However, reasoning about pointer structures in the presence of destructive updates can be quite difficult with them. In contrast, fully automatic tools based on shape analysis, such as TVLA [14,2], are specifically designed to perform well in these situations, but can not deal with precise arithmetic and induction, for example.

B$^+$ trees [1] are ordered, balanced trees that are commonly used to implement indices in databases or file systems. They have several invariants regarding tree shape, sorting, balance and node sizes.

Verification of B$^+$ trees is a hard problem. We are aware of several efforts to verify them. Two pen-and-paper proofs are [4] and [15]. The first uses two refinements with an intermediate level of nested sets. The implementation is given as Pascal code. The other uses separation logic. Algorithms are given by transitions of an abstract machine specifically designed for the problem.

The only complete mechanized verification we are aware of is [10]. It uses the separation logic framework of the Coq theorem prover and a similar formalization as [15]. Although the authors state that a significant degree of automation was achieved by customized proof tactics, the effort is still high: approx. 5000 lines of proof script were needed. Their verification, however, considers some additional operations (e.g., efficient range queries) we have not verified.

Preliminary work in TVLA is [8]. It verifies some properties of B$^+$ trees, but is restricted to a statically bounded rank.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 188–203, 2011.

Our approach uses algebraic specifications and wp-calculus as a convenient framework for verification of the relevant algorithms. Shape analysis is used as a kind of decision procedure that discharges some of the proof goals automatically. However, the proposed integration differs from common approaches with similar goals (that use for example SAT-solvers): shape analysis is parameterized with constraints that are specific to the problem domain. These constraints are used as unvalidated assumptions to guide the automatic proof. To ensure a correct analysis they have to be verified interactively.

In this work, we perform a conceptual integration by translating manually between the two worlds. We evaluate how the rather different high-level specification style used in algebraic specifications can be mapped to the shape graphs and logic of TVLA.

We use the theorem prover KIV [11], but the approach should be applicable with other interactive theorem provers, too. By combining KIV and TVLA, we have verified that our implementation of the insertion and deletion operations on B$^+$ trees maintains the invariants for tree shape, balance, sorting and node sizes and that they are a refinement of their counterparts on algebraic sets. We ensure correctness of the shape analysis specifications and – where possible – abstract from B$^+$ trees as the concrete data structure, so that many generic constraints can be reused for other pointer structures. The proofs done in KIV as well as the TVLA input files are available online at [3].

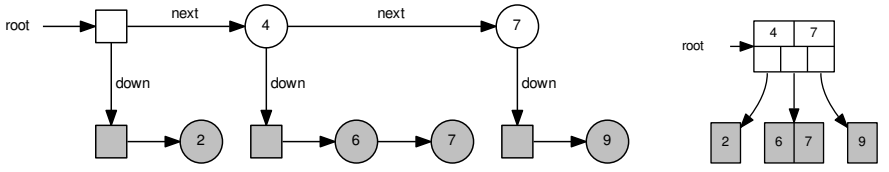
This work is organized as follows: Section 2 introduces B$^+$ trees, our verification approach and an algebraic specification of pointer structures. Section 3 briefly describes the shape analysis implemented by TVLA. Section 4 formalizes B$^+$ tree invariants and explains how these can be tracked with shape analysis. Sec. 5 summarizes our experiences, and Sec. 6 draws conclusions.

2 B$^+$ Trees and Approach

B$^+$ trees are ordered, balanced N-ary trees. They are used to implement large sets of keys (or key-value maps). They maintain several invariants to guarantee logarithmic-time operations. The main operations on B$^+$ trees are lookup, insertion and deletion. In a B$^+$ tree of rank N, a node is either a *branch*, that stores $N \leq k \leq 2N$ keys and $k+1$ downward pointers, or it is a *leaf*, that stores between N and $2N$ keys (for sets) or key-value mappings. There is an exception to this rule for the root, which must contain at least one key instead of N. A total order on keys is required. The actual *content* of the B$^+$ tree only consists of the information in the leaves, the keys in the branch nodes organize efficient access (in contrast to B-trees, that store content in internal nodes, too). A B$^+$ tree is *balanced* if all leaves are on the same level, and *sorted* if in each node the keys are sorted in increasing order, while subtrees only contain keys between adjacent keys in the parent.

We use linked lists instead of arrays for the representation of nodes. An encoding of arrays in TVLA has been developed [5,6], but unfortunately the modified TVLA versions are not available, so we remain with the core concepts in this case study.

Figure 1 shows an example B$^+$ tree in this model, compared to an equivalent array-based one. Graphical nodes displayed as boxes serve as representatives of entire B$^+$ tree nodes (subsequently called *heads*), while the round nodes (subsequently called *entries*) store the keys. The edge labels next and down indicate the names of the corresponding selectors of the respective objects. This B$^+$ tree represents the set $\{2, 6, 7, 9\}$.



(a) pointer-based **(b)** array-based

Fig. 1. B^+ tree representations

2.1 Algorithms

Both the insertion and deletion algorithm on B^+ trees essentially follow the same strategy: recursively traverse the tree down to a leaf node that is responsible for holding the given key k and perform the desired modification locally on that leaf. This may cause an underflow or overflow with respect to the node size invariant, which is restored by restructuring the tree. For example, an overfull leaf is split into halves, and an additional down-pointer and key are added to the parent. Therefore, restructuring may cascade upwards along the path the recursive descent has taken, possibly leading to growth and shrinking at the root.

```
insert_node(k, rn) {
  if rn.leaf?
  then insert_leaf(rn, k)
  else if k ≤ rn.next.key
    then insert_node(rn.down, k)
         if overfull(rn.down)
         then split_node(rn ; rt)
    else insert_bentry(rn.next, k)
}
```

```
insert_bentry(k, rbe) {
  if   rbe.next = null
     ∨ k ≤ rbe.next.key
  then insert_node(rbe.down, k)
       if overfull(rbe.down)
       then split_bentry(rbe ; rt)
  else insert_bentry(rbe.next, k)
}
```

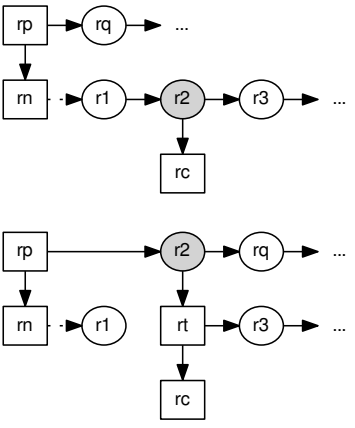
Fig. 2. Mutually recursive insertion routines

Figure 2 lists two mutually recursive subroutines of the insertion algorithm, given in the abstract program syntax used in KIV. They receive the current key as a parameter k and the current node in rn and rbe respectively – insert_node descends at a node head (displayed as boxes in Fig. 1), while insert_bentry performs similar actions at branch entries (displayed as circles). The top-level routine insert, which is not shown here, is very similar to insert_node but additionally has to deal with an empty tree as well as growth and shrinking. The actual modifications of the data structure are hidden inside the functions insert_leaf, which stores a key in a leaf, and split_node(rn ; rt) and split_bentry(rbe ; rt) that split the overfull down-child of rn resp. rbe at its median elements. The deletion algorithm is similar but more complicated because balance may be restored either by merging nodes or by transferring keys between adjacent nodes.

Figure 3 shows the implementation of split_node and its effect on the data structure, rn is the overfull node, its parent is rp and r1 denotes the entry just before the median r2. The newly allocated node is returned in rt (which is required to specify the contract).

```
split_node(rp ; rt) {
  rn := rp.down
  compute r1
  r2 := r1.next
  r3 := r2.next
  r1.next := null
  rc := r2.down
  rt := new Branch
  r2.down := rt
  rt.down := rc
  rt.next := r3
  rq := rp.next
  rp.next := r2
  r2.next := rq
}
```

(a) Code



(b) Structure before/after modification

Fig. 3. Restructuring routine split_node

2.2 Verification Approach

The verification of B$^+$ trees must establish two properties of the implementation: 1) insertion and deletion retain the B$^+$ tree invariants and 2) the operations correspond to their set-theoretic counterparts. Invariants are collected in a predicate $\mathsf{btree}(r)$ (formalized in Sec. 4) that specifies r as the root of a proper B$^+$ tree. The set of elements that a B$^+$ tree with root r represents is denoted by $\mathsf{elts}(r)$ in the following. Formally,

$$\mathsf{btree}(r) \wedge e = \mathsf{elts}(r) \rightarrow \mathbf{wp}(\mathsf{insert}(k;\, r),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \cup \{k\}) \tag{1}$$

$$\mathsf{btree}(r) \wedge e = \mathsf{elts}(r) \rightarrow \mathbf{wp}(\mathsf{delete}(k;\, r),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \setminus \{k\}) \tag{2}$$

must be proved, where $\mathbf{wp}(\alpha, \varphi)$ denotes the weakest precondition of program α with postcondition φ. e denotes the elements of the initial tree. Programs insert and delete are called with the key k to insert or delete. The root r of the tree is passed by reference as it may change.

The correctness criteria for the verification of B$^+$ trees can be decomposed into three layers along the structure of the algorithms. The top level consists of interactive proofs of the recursive insertion and deletion algorithms. These depend on shape analysis to verify subroutines that perform actual modifications to the data structure, such as split_node, split_bentry and insert_leaf, forming the intermediate layer. The basis for the verification is an algebraic specification of B$^+$ trees as pointer structures. It also serves for consistency proofs of the constraints and theorems required for shape analysis.

The interactive proofs in KIV are performed by symbolic execution of the program source code. KIV implements the wp-calculus in the form of Dynamic Logic [7], but any prover that supports Hoare calculus would be sufficient. Calls to subroutines which are verified with shape analysis are dispatched via their contracts, so the interactive verification does not have to deal with the implementation of these subroutines at all. These contracts form the interface between the top and intermediate layer.

Subroutines can be classified into restructuring, such as split_node, and content modifications consisting of insert_leaf and delete_leaf. There are 16 restructuring routines (split, merge, transfer a key to left and right sibling) for leaf-level and internal operations that also differ in whether they affect a node head. The contracts of subroutines of the same class are very similar, concrete examples are given in (3)[1] and (4) where r denotes the root of the tree. In the precondition, $\mathsf{lok}(rl, k)$ ("leaf of key") specifies that key k actually belongs into the leaf rl which is required to establish sorting in the result.

$$\begin{aligned} &\mathsf{btree}(r) \wedge \mathsf{reachable}(r, rp) \wedge e = \mathsf{elts}(r) \qquad (3)\\ &\quad \rightarrow \mathbf{wp}(\mathsf{split\_node}(rp; rt),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e)\\ &\mathsf{btree}(r) \wedge \mathsf{reachable}(r, rl) \wedge e = \mathsf{elts}(r) \wedge \mathsf{lok}(rl, k) \qquad (4)\\ &\quad \rightarrow \mathbf{wp}(\mathsf{insert\_leaf}(k, rl),\ \mathsf{btree}(r) \wedge \mathsf{elts}(r) = e \cup \{k\}) \end{aligned}$$

The main proof for the insert algorithm is concerned with the mutually recursive procedures insert_node and insert_bentry (see Fig. 2). We combine these into one proof obligation, so that recursive calls from one function to the other are covered by the induction hypothesis. The induction is carried out over the number of nodes in (sub)trees. The critical proof step is to establish $\mathsf{lok}(r.\mathsf{next}, k)$ resp. $\mathsf{lok}(r.\mathsf{down}, k)$ given that $\mathsf{lok}(r, k)$ holds for the current node r and key k – which follows from the key comparisons in the algorithm.

An alternative to this decomposition scheme is to verify the top-level with TVLA as well, for example with the technique presented in [13], which automatically computes the contracts of subroutines. However, the interactive proof also shows termination and the effort for the recursion is reasonably low.

2.3 Algebraic Formalization of Pointer Structures

Pointer structures consist of *objects* that live inside a *heap* and are accessed indirectly via typed *references*. The heap H is a partial, polymorphic function $H : \mathsf{ref}[T] \rightharpoonup T$ that maps ("dereferences") allocated references $r \in \mathsf{dom}(H)$ with $r : \mathsf{ref}[T]$ to objects $o = H(r)$ of corresponding type T. With this scheme, heap access is statically type-checked within the logic's type system.

We model objects as instances of free data types. For B^+ trees we obtain three sorts: Node for branch and leaf heads and BEntry, LEntry for their respective entries. Let refn abbreviate ref[Node] and let rn denote variables of type refn in the following (similar conventions for refbe, rbe and refle, rle).

data Node = Branch(next: refbe; down: refn) | Leaf(next: refle)
data BEntry = BEntry (key: Key; next: refbe; down: refn)
data LEntry = LEntry (key: Key; next: refle)

Node, for example, is a type freely generated by the constructors Branch and Leaf. Overloaded selector functions next and down retrieve the respective constructor arguments and are applied in postfix notation similar to Java fields, e.g., if $o = \mathsf{Leaf}(r)$

[1] This contract ignores the node sizes, see (22) for the full contract.

then $o.\mathsf{next} = r$. Additionally, predicates such as $o.\mathsf{leaf?}$ are provided to test by which constructor an object has been built.

To bridge the gap to the untyped logic of TVLA, we define supersorts/sum-types $\mathsf{ref}, \mathsf{object}$ and the enumeration of selectors sel

$$\begin{aligned} \mathsf{ref} &= \mathsf{refn} + \mathsf{refbe} + \mathsf{refle} \\ \mathsf{object} &= \mathsf{Node} + \mathsf{BEntry} + \mathsf{LEntry} \\ \mathsf{sel} &= \mathsf{next} \mid \mathsf{down} \end{aligned}$$

We assume a constant $\mathsf{null} : \mathsf{ref}$ that is never allocated in a Heap. A predicate $\mathsf{wt} : \mathsf{object} \times \mathsf{sel}$ such that $\mathsf{wt}(o, s)$ iff $o.s$ is well typed allows us to define heap properties without referring to concrete types of the case study, for example paths and treeness (see Sec. 4.1).

As opposed to TVLA, algebraically specified heaps can contain *dangling pointers* (references pointing outside the heap). A consistent heap requires that whenever an object is stored in the heap, all of its reference selectors are either null or point inside the heap again. In the remainder of this text, we assume that all heaps are consistent.

$$\begin{aligned} \mathsf{consistent}(H) \leftrightarrow \quad & \forall r \in \mathsf{dom}(H), s : \mathsf{sel}. \\ & \mathsf{wt}(H(r), s) \rightarrow (H(r).s = \mathsf{null} \vee H(r).s \in \mathsf{dom}(H)) \end{aligned}$$

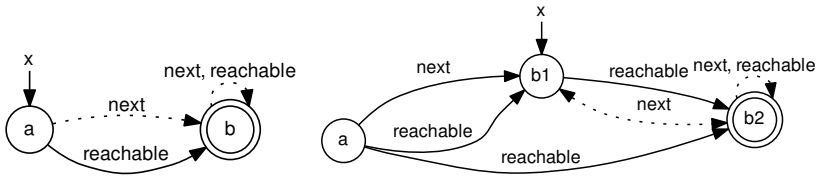
3 Introduction to Parametric Shape Analysis

Parametric shape analysis [14] is an instance of abstract interpretation. The actual computations are performed over an approximation of concrete states. A fully automatic analysis is achieved by keeping the abstract state space finite, so that it can be explored exhaustively. The analysis is conservative, i.e., proofs sometimes fail even if the program is correct, but not the other way round. Parametric shape analysis is a generic framework. The user controls abstraction, the encoding of data structure properties and the program statement semantics. The approach is implemented in the TVLA (Three-Valued Logic Analyzer) tool.

Parametric shape analysis is based on untyped first-order logic with transitive closure, but without function symbols (which are encoded as predicate symbols). The logic is three-valued with the domain of truth values $\mathbb{B}_3 = \{0, \frac{1}{2}, 1\}$. The third value $\frac{1}{2}$ denotes unknown truth (sometimes called "indefinite") and aggregates contradicting truth values in abstract states. An abstraction function maps the infinite concrete domain of possible data structures to finitely many bounded abstract structures.

The key idea is to partition objects allocated in the heap into finitely many equivalence classes, so that all objects in the same class are indistinguishable by a set of user-definable properties of interest. These properties are given by unary *abstraction predicates*. Typically, there are singleton classes for objects pointed to by program variables. Classes with more than one element are called *summary nodes*. A single summary node can represent a whole subtree, for example.

As an example, Fig. 4a shows a *shape graph* for a heap which contains a singly-linked list. Node a represents a single object, while all other memory cells are grouped



(a) original structure **(b)** after splitting b into b1 and b2

Fig. 4. Splitting of a summary node

into the (doubly circled) summary node b. The partitioning is according to the fact that program variable x points to a.

The program signature (program variables and selectors) is encoded as part of the logical signature. A program variable x becomes an unary predicate symbol $\mathsf{x}(r)$ that is constrained to hold for one node only: the one x points to. Selectors become binary predicate symbols, for example $H[r_1].\mathsf{next} = r_2$ is encoded as $\mathsf{next}(r_1, r_2) = 1$. These predicates are constrained to be partial functions. The predicate symbols arising from the program signature are called *core predicates*.

In Fig. 4a the dotted arrow indicates that $\mathsf{next}(\mathsf{a}, \mathsf{b}) = \frac{1}{2}$, since the next selector of node a points to some node in class b, but not to all. All nodes summarized by b are definitely reachable from a, as indicated by the solid arrow.

The program itself is represented by a finite transition system between states, each state corresponding to a specific value of the program counter. For each state shape analysis computes the set of shape graphs that approximate all heap structures that are possible at that program point. This can be done by a fixpoint computation, since the number of states as well as the number of shape graphs is finite.

To compute the fixpoint, for each transition a *precondition* (%p in TVLA) and an *update formula* must be defined. The precondition encodes tests of conditionals or loops, the update formula must specify the effects of assignments for all core predicates.

As an example x := y is represented by the update formulas $\varphi^{\mathsf{x}}_{\mathsf{x}:=\mathsf{y}}(r) = \mathsf{y}(r)$ for x and $\varphi^{\mathsf{p}}_{\mathsf{x}:=\mathsf{y}}(r) = \mathsf{p}(r)$ for all other predicates p. A transition is executed by evaluating each update formula in the old state yielding the predicate's values in the new state. The statement x.next := y is represented as $\varphi^{\mathsf{next}}_{\mathsf{x.next}:=\mathsf{y}}(r_1, r_2) = \mathsf{next}(r_1, r_2) \vee \mathsf{x}(r_1) \wedge \mathsf{y}(r_2)$.[2] [14] defines preconditions and update formulas for standard statements such as assignments, selector access and case distinctions. All selector assignment statements x.sel := y assume, that x.sel = null, so that edges are either added or removed but not both in one step. Therefore when translating KIV programs to TVLA, these assignments have to be rewritten into x.sel := null; x.sel := y.

During the run of a program, the abstraction is dynamically adjusted with every statement. Suppose, the statement x := x.next should be executed in Fig. 4a and recall that each program variable should point to a singleton node: for x we have to get hold of the object a.next in this case. Parametric shape analysis splits summary nodes as necessary with an operation called *materialization*, as shown in Fig. 4b. Here, b1 = a.next is the

[2] For x : refbe, this corresponds to $H := H[\mathsf{x} \mapsto \mathsf{BEntry}(H(\mathsf{x}).\mathsf{key}, \mathsf{y}, H(\mathsf{x}).\mathsf{down})]$ in KIV.

direct successor of a and b2 represents the remaining elements of b.[3] Technically, the analysis ensures that a given set of *focus formulas* yields definitive values for all objects r. For the assignment x := x.next the focus formulas are $\mathsf{x}(r)$ and $\exists r_1.\, \mathsf{x}(r_1) \wedge \mathsf{next}(r_1, r)$. Additional focus formulas can be given to cause extra splits of summary nodes. We will need these in Sec. 4.3.

Switching to a finite domain cannot preserve all information available in the infinite domain. To preserve more information, two strategies are possible, the *instrumentation* and the *guard* strategy. The first explicitly defines additional *instrumentation predicates*. Predicates reachable and step, defined by

$$\mathsf{reachable}(r_1, r_2) \leftrightarrow \mathsf{step}^*(r_1, r_2) \quad \text{and} \quad \mathsf{step}(r_1, r_2) \leftrightarrow \bigvee_{s \in \mathsf{sel}} s(r_1, r_2)$$

are such instrumentation predicates (step* is the reflexive transitive closure of step). The guard strategy uses global invariants INV that hold at all times during the execution of an algorithm. Formally, these are defined by *consistency rules* (`%r` in TVLA). For example, the following rule expresses that an algorithm never creates cyclic structures

$$\mathsf{reachable}(r_1, r_2) \rightarrow \neg \mathsf{step}(r_2, r_1)$$

The value of instrumentation predicates is explicitly stored in shape graphs. By default, executing a transition reevaluates the defining formula in the new state. However, this may lose the definite information that an instrumentation predicate stores. For the structure resulting from the assignment shown in Fig. 4b we have $\mathsf{reachable}'(\mathsf{b1}, \mathsf{b2}) = \mathsf{next}'^*(\mathsf{b1}, \mathsf{b2}) = \frac{1}{2}$. To prevent this, parametric shape analysis allows explicit update formulas for instrumentation predicates. Assuming that $\mathsf{x.sel} = \mathsf{null}$ and $\mathsf{y} \neq \mathsf{null}$ the update formula for reachable and an assignment x.sel := y is

$$\varphi^{\mathsf{reachable}}_{\mathsf{x.sel}:=\mathsf{y}}(r_1, r_2) \;= \mathsf{reachable}(r_1, r_2) \;\vee\; \mathsf{reachable}(r_1, \mathsf{x}) \wedge \mathsf{reachable}(\mathsf{y}, r_2)$$

This update formula preserves definite reachability of b2 from b1. However, update formulas that do not comply with the definitions of instrumentation predicates lead to an unsound analysis. To ensure soundness, we verify in KIV that given an update formula $\varphi^{\mathsf{p}}_{stm}$ for statement *stm* and predicate p

$$INV \wedge H_0 = H \rightarrow \mathbf{wp}\left(stm, \tau\left(\varphi^{\mathsf{p}}_{stm}\right)(r_1, \ldots, r_n, H_0) \leftrightarrow \mathsf{p}(r_1, \ldots, r_n, H)\right) \qquad (5)$$

holds, i.e., the update formula evaluated in the old heap H_0 must equal the instrumentation predicate evaluated in the heap H after *stm* has been executed. Here, $\tau(\varphi)$ denotes the translation of formula φ to KIV's logic (see appendix A). Note that the proof obligation may assume the global invariants established by the guard strategy to yield stronger update formulas. To establish such global invariants, *guards* (`%message` in TVLA) are attached to transitions, that ensure that the invariant is preserved (therefore the name). TVLA stops the analysis whenever it finds that an input shape graph for a transition violates its guard. To ensure soundness, we verify in KIV

$$INV \wedge \tau(\psi) \rightarrow \mathbf{wp}(stm, INV) \qquad (6)$$

for each assignment *stm* and its guard formula ψ.

[3] A second shape graph (not shown) is necessary for the special case where b1 has no successors.

When defining a global invariant is possible, then checking guards is typically more efficient than the definition of INV as an additional instrumentation predicate, that has to be tracked to be valid in all intermediate states. It is also easier to verify guards in KIV, since update formulas for invariants tend to be rather complex, while guards can often be simplified by making them stronger than strictly necessary to prove (6), see (11) for an example.

4 Formalization and Verification of B$^+$ Tree Invariants

In this section, we formalize B$^+$ tree invariants. We start with the intuitive definitions as used in KIV and adapt them to shape analysis by using instrumentation predicates, consistency rules, guards and update formulas. We focus on critical aspects, so this section is not exhaustive – additional instrumentation predicates and constraints are often required to achieve a precise analysis result.

The B$^+$ tree invariants are collected in the predicate btree, the set of keys $\mathsf{elts}(r)$ that a B$^+$ tree with root r represents is axiomatized as shown. The predicate root restricts the heap to contain only the tree pointed to by r. Predicates in this section have an implicit heap parameter H and $H[r].s$ is abbreviated as $r.s$.

$$\begin{aligned}
\mathsf{btree}(r,[r_1,\ldots,r_m]) \;\leftrightarrow\;& \mathsf{root}(r) \wedge \mathsf{tree}(r) \\
& \wedge \forall r'.\quad \mathsf{reachable}(r,r') \rightarrow \mathsf{balanced}(r') \wedge \mathsf{sorted}(r') \\
& \qquad \wedge r' \notin \{r_1,\ldots,r_m\} \rightarrow \mathsf{oksize}(r') \\
\text{where}\quad \mathsf{root}(r) \;\leftrightarrow\;& \forall r'.\, \mathsf{reachable}(r,r') \\
k \in \mathsf{elts}(r) \;\leftrightarrow\;& \exists r'.\, \mathsf{reachable}(r,r') \wedge r'.\mathsf{lentry?} \wedge r'.\mathsf{key} = k
\end{aligned} \tag{7}$$

Predicate btree has an optional list of nodes $r_1,\ldots,r_m$ whose size may be out of bounds, which is used in contracts of restructuring subroutines.

Quantifiers range over allocated references, and by convention, free variables are universally quantified. The following subsections specify predicates tree (has tree shape), balance and sorted (tree is balanced and sorted), elts and oksize, and show the difficulties of encoding them in TVLA.

4.1 Tree Shape

We characterize trees as follows: a node is the root of a tree if there is at most one path from the root to every node in that tree. A path starts with some reference r_1 and follows a sequence of applicable selectors xs : list[sel] to another reference r_2 ([] denotes the empty list, + list concatenation).

$$\begin{aligned}
\mathsf{tree}(r) \leftrightarrow\;& r \neq \mathsf{null} \wedge \forall x_1,x_2,r_1,r_2. \\
& \qquad \mathsf{path}(r,x_1,r_1) \wedge \mathsf{path}(r,x_2,r_2) \rightarrow (x_1 = x_2 \leftrightarrow r_1 = r_2) \\
\mathsf{path}(r_1,[\,],r_2) \leftrightarrow\;& r_1 \neq \mathsf{null} \wedge r_1 = r_2 \\
\mathsf{path}(r_1,s+xs,r_2) \leftrightarrow\;& r_1 \neq \mathsf{null} \wedge \mathsf{path}(r_1.s,xs,r_2) \wedge \mathsf{wt}(H(r_1),s)
\end{aligned}$$

These definitions serve as an intuitive formalization and are used in the algebraic specification for various consistency proofs. For shape analysis though, an alternative characterization is required that does not use recursive definitions or an explicit representation of paths. We employ the guard strategy, as the algorithms preserve tree shape in all intermediate structures. It is sufficient to prohibit cyclic and converging paths in general, similar to [9]. Converging paths are excluded by consistency rules (8) and (9), cycles are excluded by (10), forming the global invariant for tree shape. Guard (11) is used for assignments x.sel := y.

$$r_1.\mathsf{next} = r_2.\mathsf{down} \rightarrow r_1.\mathsf{next} = \mathsf{null} \tag{8}$$

$$r_1.s = r_2.s \wedge r_1.s \neq \mathsf{null} \rightarrow r_1 = r_2 \qquad \text{for } s \in \{\mathsf{next}, \mathsf{down}\} \tag{9}$$

$$r_1.s = r_2 \rightarrow \neg\mathsf{reachable}(r_2, r_1) \tag{10}$$

$$\neg\mathsf{reachable}(\mathsf{y}, \mathsf{x}) \wedge \neg\exists r.\ (\mathsf{reachable}(r, \mathsf{y}) \wedge r \neq \mathsf{y}) \tag{11}$$

We have proven that these constraints are equivalent to $\forall r.\ \mathsf{tree}(r)$ under the assumption that there is some r with $\mathsf{root}(r)$. The latter is an instrumentation predicate that shape analysis can prove easily to be true in the final states of the subroutines. $\mathsf{root}(r)$ cannot be used as an invariant, since routines like split_node have intermediate states where the tree is split into several parts.

4.2 Balance

We characterize balance as follows: A B$^+$ tree is balanced, if each node fulfills the constraint that its down successor is the root of a subtree of height one less than the subtree of its next successor. The height of a node is determined by the maximum number of down selectors on a path to a leaf starting at that node.

$$\mathsf{height}(r) = \begin{cases} 0 & \text{if } r = \mathsf{null} \\ \max\left[\mathsf{height}(r.\mathsf{next}),\ \mathsf{height}(r.\mathsf{down}) + 1\right] & \text{otherwise} \end{cases} \tag{12}$$

$$\begin{aligned} \mathsf{balanced}(r) \leftrightarrow\ & r.\mathsf{next} \neq \mathsf{null} \wedge r.\mathsf{down} \neq \mathsf{null} \\ & \rightarrow \mathsf{height}(r.\mathsf{next}) = \mathsf{height}(r.\mathsf{down}) + 1 \end{aligned} \tag{13}$$

This as well as the following definitions assume that $\mathsf{tree}(r)$ holds for all relevant references r. Note that for arbitrary heaps with cyclic structures they would be inconsistent.

These definitions are hard to reproduce in shape analysis as they are based on arithmetic. Therefore we use different definitions in TVLA based on two binary predicates eqh ("equal height") and olh ("one-less height") defined by (14) and (15) that do a *local* comparisons of heights. (16) is a definition of balanced in terms of these predicates that can be proven to be equivalent to (13).

$$\mathsf{eqh}(r_1, r_2) \leftrightarrow \mathsf{height}(r_1) = \mathsf{height}(r_2) \tag{14}$$

$$\mathsf{olh}(r_1, r_2) \leftrightarrow \mathsf{height}(r_1) + 1 = \mathsf{height}(r_2) \tag{15}$$

$$\begin{aligned} \mathsf{balanced}(r) \leftrightarrow\ & (r.\mathsf{next} \neq \mathsf{null} \rightarrow \mathsf{eqh}(r.\mathsf{next}, r)) \\ \wedge\ & (r.\mathsf{down} \neq \mathsf{null} \rightarrow \mathsf{olh}(r.\mathsf{down}, r)) \end{aligned} \tag{16}$$

As the height function is not available during shape analysis, eqh and olh must be specified as core predicates. Several constraints compensate the missing definitions and propagate height comparisons (transitively) between nodes, such as $\mathsf{eqh}(r_1, r_2) \rightarrow \neg\mathsf{olh}(r_1, r_2)$ and $\mathsf{olh}(r_1, r_3) \wedge \mathsf{olh}(r_2, r_3) \rightarrow \mathsf{eqh}(r_1, r_2)$.

Specified as core predicates, eqh and olh do not automatically reflect changes to the height of nodes arising from modifications, so they must be updated explicitly. The critical statements are null assignments to selectors.

We demonstrate the strategy for x.next := null. There are two cases: If x.down = null the height of x is reduced to one, possibly changing the heights of its ancestors, too. We model this by forgetting the height relations of the affected nodes. If x.down is non-null and x is balanced, then the height of x remains unchanged. This implies that the height of all other nodes, in particular its ancestors, is unaffected too. This is expressed by lemma (17), which implies that relative comparisons eqh are also unaffected (second case of (18)).

$$\begin{aligned} \mathsf{x.next} \neq \mathsf{null} \ \wedge h = \mathsf{height}(r) \wedge \forall r_0.\ \mathsf{tree}(r_0) \wedge \mathsf{balanced}(r_0) \\ \rightarrow \mathbf{wp}\,(\mathsf{x.down} := \mathsf{null},\ h = \mathsf{height}(r)) \end{aligned} \tag{17}$$

Formula (18) shows the update of eqh. We ensure that x is actually balanced with an appropriate guard.

$$\mathsf{eqh}'(r_1, r_2) \leftrightarrow \begin{cases} \frac{1}{2} & \text{if } \mathsf{reachable}(r_1, \mathsf{x}) \vee \mathsf{reachable}(r_2, \mathsf{x}) \\ & \text{and } \mathsf{x.down}_1 = \mathsf{null} \\ \mathsf{eqh}(r_1, r_2) & \text{otherwise} \end{cases} \tag{18}$$

Updates for the first case immediately destroy balance information at ancestors. To avoid this problem, the statements are rearranged to ensure that no ancestor is affected at all by first detaching the node in question from its parent. For example, the underlined statement r1.next := null in Fig. 3 is therefore placed before any heap modification.

Height information is recovered when the first child is attached to a node: the height of a node with exactly one non-null selector is determined by its (single) child, as expressed by the following constraints:

$$\begin{aligned} r.\mathsf{down} = \mathsf{null} \wedge r.\mathsf{next} \neq \mathsf{null} &\rightarrow \mathsf{eqh}(r.\mathsf{next}, r) \\ r.\mathsf{down} \neq \mathsf{null} \wedge r.\mathsf{next} = \mathsf{null} &\rightarrow \mathsf{olh}(r.\mathsf{down}, r) \end{aligned}$$

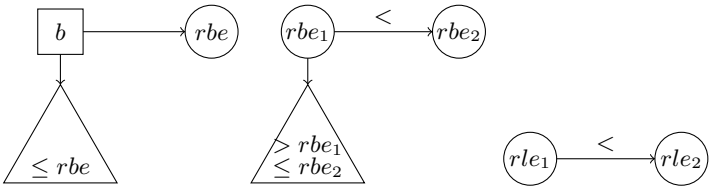
4.3 Sorting

A B^+ tree is sorted if all of its nodes obey the constraints graphically given in Fig. 5. Sorting is maintained with the guard strategy. Equation (19) formalizes the sorting constraint shown in Fig. 5b for branch entries rbe_1 that is preserved in all intermediate states of the algorithm.

$$\forall r.\ \mathsf{reachable}(rbe_1.\mathsf{down}, r) \rightarrow rbe_1 <_\mathsf{k} r \wedge r \leq_\mathsf{next} rbe_1 \tag{19}$$

where $r \leq_\mathsf{next} rbe_1 \leftrightarrow rbe_1.\mathsf{next} \neq \mathsf{null} \rightarrow r \leq_\mathsf{k} rbe_1.\mathsf{next}$

and $r_1 \leq_\mathsf{k} r_2 \leftrightarrow r_1.\mathsf{key} \leq r_2.\mathsf{key}$ ($<_\mathsf{k}$ is defined similarly)



(a) branches **(b)** branch entries **(c)** leaf entries

Fig. 5. Sorting Constraints. Branch-nodes are shown as boxes, entries as circles.

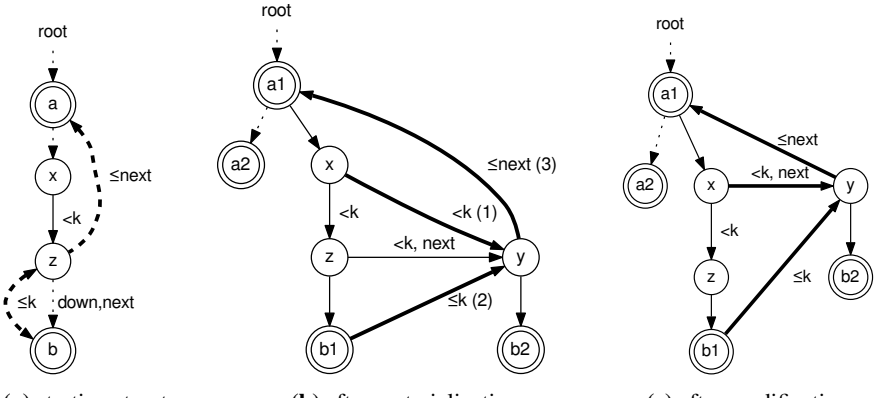
The guard for the assignment x.next := y is

$$\mathsf{x} <_\mathsf{k} \mathsf{y} \wedge (\forall r.\ \mathsf{reachable}(\mathsf{x.down}, r) \ \rightarrow r \leq_\mathsf{k} \mathsf{y}) \wedge \tag{20}$$
$$\forall r_1, r_2.\ \mathsf{reachable}(r_1.\mathsf{down}, \mathsf{x}) \wedge \mathsf{reachable}(\mathsf{y}, r_2) \rightarrow r_2 \leq_\mathsf{next} r_1$$

The first conjunct ensures that keys in the linked list of entries remain ordered. The second conjunct checks elements in the down subtree of x to conform to y. The third conjunct checks that nodes in the attached subtree conform to all ancestors r with a down-pointer towards x, where $\mathsf{reachable}(r_1.\mathsf{down}, \mathsf{x})$ determines these ancestors. The guard for x.down := y is similar.

The hard problem in TVLA is to ensure that the guard definitely holds when such statements are executed. Fig. 6 shows the execution of the typical sequence y := z.next; z.next := null; x.next := y starting with Fig. 6a. The critical relations are depicted as thick arrows in Fig. 6b, each corresponds to one of the conjuncts. When these relations evaluate to definite values, the guard holds, as shown in Fig. 6c.

The first conjunct x $<_\mathsf{k}$ y follows by transitivity over z and is established in (b). To derive the other two conjuncts, we focus on nodes r such that $\mathsf{reachable}(\mathsf{z.next}, r)$ when executing y := z.next, and we focus on nodes r such that $\mathsf{reachable}(r.\mathsf{down}, \mathsf{x})$ when executing z.next := null. These have the effect of splitting b and a respectively. b1 now represents the subtree that must be checked in the second conjunct and a1 gives exactly



(a) starting structure **(b)** after materializations **(c)** after modification

Fig. 6. Tracking sorting through modifications

the ancestors that are covered by the third conjunct. The necessary relations are then derived from the sorting invariant *before* the statement z.next := null is executed, explicitly stored in the structure and thus available when the guard is evaluated.

Note that in order to prevent nodes that are materialized from being merged back, we have to employ several derived abstraction predicates, e.g., reachable-from-x$(r) \leftrightarrow$ reachable(x, r) for program variables x. Deriving unary (reachability) predicates from binary ones with respect to program variables is a common idiom in TVLA.

4.4 Elements

In [12], the set elts of elements a pointer structure represents is tracked by explicitly labeling objects whose key is in the set in the initial state with an additional (core) predicate. The final state is then related to this predicate. For our case this formalization has the drawback that leaf entries must be kept distinct from other objects, so that rle.key $\in$ elts(r) always yields definite values. Instead, we mark a leaf entry when its keys changes, or when it is allocated or deallocated. For insert_leaf we establish that it allocates at most one leaf entry, and changes no key. This is expressed as postcondition (21), where H and H' are the initial and the final heap. The modifications of elts can be derived from this condition in KIV. A similar postcondition is proved for delete_leaf.

$$\begin{aligned} \exists rle. \quad & H'[rle].\mathsf{key} = k \wedge \mathsf{dom}(H') = \mathsf{dom}(H) \cup \{rle\} \\ & \wedge \forall rle_1 \in \mathsf{dom}(H).\ H[rle_1].\mathsf{key} = H'[rle_1].\mathsf{key} \end{aligned} \tag{21}$$

4.5 Node Sizes

The size of a node rn is determined by the number of its entries r, i.e., those reachable by following next selectors only. These entries are collected in a set, extensionally defined as $r \in \mathsf{nset}(rn) \leftrightarrow \mathsf{next}^*(rn, r)$. Let N denote the rank of the B$^+$ tree, then

$$\mathsf{oksize}(r) \leftrightarrow (r.\mathsf{node?} \rightarrow (\mathbf{if}\ \mathsf{root}(r)\ \mathbf{then}\ 1\ \mathbf{else}\ N) \leq |\mathsf{nset}(r)| \leq 2N)$$

Node sizes are verified by a strategy similar to [6]. There, the sets of concrete individuals represented by summary nodes are tracked, as well as the cardinalities of these sets. [6] is an extension to TVLA that seems capable to directly verify the node size invariant. However, the prototype implementation is not available, so we imitate the strategy. As an example, for split_node$(rp; rt)$, we prove the following contract with TVLA:

$$\begin{aligned} rn = rp.\mathsf{down} \wedge \mathsf{btree}(r, [rn]) \wedge \mathsf{reachable}(r, rp) \wedge e = \mathsf{elts}(r) \wedge \mathsf{median}(rn, r_1.\mathsf{next}) \\ \rightarrow\ \mathbf{wp}(\mathsf{split\_node}(rp; rt),\ \ & \mathsf{btree}(r, [rp, rn, rt]) \wedge e = \mathsf{elts}(r) \\ & \wedge \mathsf{nset}(rp) = \mathsf{nset}_0(rp) \cup \{r_1.\mathsf{next}\} \\ & \wedge \mathsf{nset}_0(rn) = \mathsf{nset}(rn) \cup \{r_1.\mathsf{next}\} \cup \mathsf{nset}(rt)) \end{aligned} \tag{22}$$

where $\mathsf{nset}_0(r)$ denotes the set of entries of r in the initial state. nset-membership is encoded as binary predicates in TVLA. From (22) we prove in KIV that if $|\mathsf{nset}_0(rn)| = 2N + 1$ then both rn and rt have now size N and satisfy oksize, implying btree$(r, [rp])$.

5 Results and Experiences

To make TVLA usable as a decision procedure we had to solve two problems: the first was to bridge the gap between explicit, typed algebraic heaps specified as partial functions and the implicit view of heaps encoded as the domain of predicates defined in untyped logic. The solution caused some overhead in KIV, to support switching between the generic specification and its instance for B$^+$ trees. It is however a generic solution that allows us to verify the constraints shape analysis uses for generic predicates such as tree shape or acyclicity once and for all. The second problem was to determine (5) and (6) as the right proof obligations for the instrumentation and the guard strategy.

The overall effort of the case study was around six person-months. The first month was necessary to get familiar with TVLA's user interface, which is very low level. A simple script (available on the website) that removes superfluous information from the output and colorizes the shape graphs was invaluable. Another script was used to generate TVLA transition systems from code.

The main task then was to translate the natural definitions of the B$^+$ tree invariants into suitable TVLA constraints. It roughly took three person-months to iteratively figure out the right instrumentation predicates, update formulas and consistency rules given in Sec. 4 for the B$^+$ tree invariants by analyzing failed TVLA proofs.

The remaining two months were spent on setting up the KIV specifications (including the generic theory), proving correctness of update formulas/guards and the interactive proofs of the main recursion.

The main proofs for the recursive programs were easy using the lemmas established by shape analysis. The most expensive consistency proofs are for update formulas like (17) and guards like (20). Some of them still required some dozen interactions. This agrees with our expectations that interactive reasoning about pointer manipulations is difficult. However, we have found that these proofs are required, many of the more complex constraints we used in TVLA were initially wrong.

TVLA proofs for most of the subroutines required run times below one minute on a 2.8 GHz CPU equipped with 8 Gb of main memory running 64 bit Linux. Consumption of main memory is high, usually between 500 Mb and 1 Gb, supposedly caused by the high number of predicates (around 30 binary and over 160 unary predicates). A few subroutines, such as rotations in the middle of the tree, took up to 5 minutes.

From our experience, attempting an analysis of the whole insert and delete algorithms with the final specification with TVLA seems feasible. Initial attempts, however, indicate that running TVLA on the composed code requires further optimizations. In particular, the strategy for sorting creates too many structures when traversing the full tree. We also think that it is not practical to develop the specification using TVLA on the full program, since the number of shape graphs grows rapidly with the length of the program, up to several thousands. These would have to be analyzed to find out where exactly the analysis goes wrong. For the subroutines the number was much lower, typically around one hundred.

6 Conclusion

We have verified an implementation of the main algorithms for B^+ trees using a combination of interactive theorem proving and automated shape analysis.

Our results indicate that the combination of both techniques is a significant improvement compared to using one approach alone. Automation using Shape Analysis has been significantly better than if we would have used KIV exclusively. Soundness of the shape analysis results would have been rather doubtful without proving the more complex constraints with an interactive theorem prover.

The case study has also shown how to bridge the gap between an abstract, typed algebraic approach used by almost all interactive theorem provers and the untyped approach of TVLA in general. Based on these results it is clear now how to implement an automated translation of KIV programs, predicates and constraints to TVLA (which remains work to do).

We must however concede that shape analysis is not as easily usable as a decision procedure would be. There is still a lot of specific knowledge of the internal working of TVLA required to define the right instrumentation predicates (for example $\leq_{\mathsf{next}}$), and (even more) to analyze failed proof attempts from TVLA. Getting meaningful counterexamples from failed proof attempts to analyze whether a proof failed since the goal was wrong or due to overapproximation is still one of the most time-consuming tasks, and a topic for further work.

Acknowledgments. We thank Alexander Knapp for valuable feedback.

References

1. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. Acta Informatica 1, 173–189 (1972)
2. Bogudlov, I., Lev-Ami, T., Reps, T., Sagiv, M.: Revamping TVLA: Making Parametric Shape Analysis Competitive. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 221–225. Springer, Heidelberg (2007)
3. Ernst, G.: KIV and TVLA proofs for B^+-Trees (2011), http://www.informatik.uni-augsburg.de/swt/projects/btree.html
4. Fielding, E.: The specification of abstract mappings and their implementation as B+ trees. Technical report, Oxford University, PRG-18 (1980)
5. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. In: Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, POPL, pp. 338–350. ACM, New York (2005)
6. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Proc. of the 36th ACM SIGPLAN-SIGACT Symp Principles of programming languages, POPL, pp. 239–251. ACM, New York (2009)
7. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
8. Herter, J.: Towards shape analysis of B-trees. Master's thesis, Universität Saarbrücken (2008)
9. Loginov, A., Reps, T., Sagiv, M.: Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)

10. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Proc. of the 37th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, POPL, pp. 237–248. ACM, New York (2010)
11. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, pp. 13–39. Kluwer, Dordrecht (1998)
12. Reineke, J.: Shape analysis of sets. In: Workshop "Trustworthy Software". IBFI (2006)
13. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural Shape Analysis for Cutpoint-Free Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 284–302. Springer, Heidelberg (2005)
14. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24, 217–298 (2002)
15. Sexton, A., Thielecke, H.: Reasoning about B+ trees with operational semantics and separation logic. Electron. Notes Theor. Comput. Sci. 218, 355–369 (2008)

A Translation from KIV to TVLA

This appendix sketches the formal definition of the translation between KIV and TVLA, which is in essence a standard construction for a homomorphism.

In KIV the semantics of a specification $SPEC = (\Sigma, Ax)$ with many-sorted signature $\Sigma = (S, F, P)$ and Axioms Ax is the class of all algebras $\mathcal{A} = ((A_s)_{s\in S}, f^{\mathcal{A}}, p^{\mathcal{A}}))$ with carrier sets A_s for every sort $s \in S$, functions $f^{\mathcal{A}}$ for $f \in F$ and predicates $p^{\mathcal{A}}$ for $p \in P$ that satisfies the axioms. A valuation v maps variables to appropriate elements of the carrier sets. In particular $v(H)$ for the heap H is a partial function from references to objects.

The corresponding signature used in TVLA contains predicate symbols s for every selector function $.s$, a unary predicate x for every program variable and predicates q for heap dependent predicates from KIV (like btree) dropping the heap parameter. All other arguments of these predicates are of reference type. A pair of an algebra $\mathcal{A}$ and a valuation v can be translated to an untyped model $\mathcal{U} := \rho(\mathcal{A}, v)$ of TVLA. The carrier set U of $\mathcal{U}$ is defined as the domain of the heap: $U := \{a : a \in dom(v(H))\}$. Selectors are interpreted as

$$s^{\mathcal{U}} := \left\{(a, b) : a \in dom(v(H)) \wedge b = v(H)(a).s^{\mathcal{A}} \wedge b \in dom(v(H))\right\}$$

and other predicates q are interpreted as

$$q^{\mathcal{U}}(a_1, \ldots, a_n) \text{ iff } q^{\mathcal{A}}(a_1, \ldots, a_n, v(H))$$

The semantic translation ρ of algebras corresponds to a syntactic translation τ of TVLA formulas to KIV formulas. For example,

$$\tau(s(r_1, r_2)) = (H[r_1].s = r_2) \text{ and } \tau(q(r_1, \ldots, r_n)) = q(r_1, \ldots, r_n, H)$$

It is easy to prove (by induction over the formula) that for any TVLA formula φ

$$\rho(\mathcal{A}, v) \models \varphi \text{ iff } \mathcal{A}, v \models \tau(\varphi)$$

Therefore to prove that a formula φ is valid in TVLA we prove $\tau(\varphi)$ in KIV.

Semantically, assignments stm in KIV map a valuation v to a modified valuation v'. Proof obligation (5) guarantees that for an update formula φ^p_{stm}

$$\rho(\mathcal{A}, v') \models p(r_1, \ldots, r_n) \text{ iff } \rho(\mathcal{A}, v) \models \varphi^p_{stm}(r_1, \ldots, r_n)$$

i.e., the semantics of p in the state $\rho(\mathcal{A}, v')$ after the assignment is as predicted by φ^p_{stm}.

Runtime Verification of Component-Based Systems*

Yliès Falcone[1], Mohamad Jaber[2], Thanh-Hung Nguyen[2],
Marius Bozga[2], and Saddek Bensalem[2]

[1] INRIA, Rennes - Bretagne Atlantique, France
[2] VERIMAG, Université Grenoble I, France
Firstname.Lastname@inria.fr, Firstname.Lastname@imag.fr

Abstract. Verification of component-based systems still suffers from limitations such as state space explosion since a large number of different components may interact in an heterogeneous environment. Those limitations entail the need for complementary verification methods such as *runtime verification* based on dynamic analysis and prone to scalability. In this paper, we integrate runtime verification into the BIP (Behavior, Interaction, and Priority) framework. BIP is a powerful component-based framework for the construction of heterogeneous systems. Our method augments BIP systems with monitors checking a user-provided specification. This method has been implemented in RV-BIP, a prototype tool that we used to validate the whole approach on a robotic application.

1 Introduction

A component-based approach consists in building complex systems by composing components (building blocks). This confers numerous advantages (e.g., productivity, incremental construction, compositionality) that allow to deal with complexity in the construction phase. Component-based systems (CBS) are desirable because they allow reuse of sub-systems as well as their incremental modification without requiring global changes. Their development requires methods and tools supporting a concept of architecture which characterizes the coordination between components. An architecture structures a system and involves components and relationships between the externally visible properties of those components. The global behavior of a system can, in principle, be inferred from the behavior of its components and its architecture. Component-based design is based on the separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis on coordination constraints between components even if local computations on components are not visible (i.e., components are "black boxes").

BIP (Behavior Interaction Priority). BIP is a general framework supporting rigorous design. It uses a dedicated language and an associated toolset supporting the design flow. The BIP language allows building complex systems by coordinating the behavior of a set of atomic components. Behavior is described with Labelled Transition Systems (LTS) extended with data and functions written in C. The description of coordination

* This work is partially supported by the FP7 IP ASCENS.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 204–220, 2011.

between components is layered. The first layer describes the interactions between components. The second layer describes dynamic priorities between the interactions and is used also to express scheduling policies. The combination of interactions and priorities characterizes the overall architecture of a system. It confers BIP strong expressiveness that cannot be matched by other existing formalism dedicated to CBS [1]. Moreover, BIP has a rigorous operational semantics: the behavior of a composite component is formally described as the composition of the behaviors of its atomic components. This allows a direct relation between the underlying semantic model and its implementation.

Runtime-verification (RV) [2,3,4] is an effective technique to ensure, at runtime, that a system meets a desirable behavior. It can be used in numerous application domains, and more particularly when integrating together unreliable software components. In RV, a run of the system under scrutiny is analyzed incrementally using a decision procedure: a *monitor*. This monitor may be generated from a user-provided high level specification (e.g., a temporal formula, an automaton). This monitor aims to detect violation or satisfaction w.r.t. the given specification. Generally, it is a state machine processing an execution sequence (step by step) of the monitored program, and producing a sequence of verdicts (truth-values taken from a truth-domain) indicating specification fulfillment or violation. Recently [4] a new framework has been introduced for runtime verification. This expressive framework, leveraged by a finite-trace semantics and an expressive truth-domain, allows to monitor all specifications expressing a linear temporal behavior.

The proposed approach (informal overview). We introduce a complementary validation technique for CBS in general and BIP systems in particular. We leverage the BIP framework by integrating a component-based version of the runtime verification framework introduced in [4]. Given a specification, our method uniformly integrates a monitor as an additional component in a BIP system that is able to runtime check the satisfaction or violation of the specification. The whole method is implemented in a prototype tool, RV-BIP, that automatically instrument BIP systems with monitors. Thanks to the code generator of BIP, the generated self-monitoring system can be directly translated into an actual C module embedded in the global system whose behavior is checked at runtime against the specification. The whole approach has been evaluated on a real robotic application and our experiments validate the relevance of our method.

Paper Organization. The paper is structured as follows. In Section 2 we give a minimal introduction to the BIP framework. Section 3 defines an abstract RV framework for CBS described in BIP. Section 4 shows how the abstract RV framework is implemented for BIP systems. Section 5 describes RV-BIP, a prototype implementation of our method, used to evaluate our method on a robot application. Section 6 is dedicated to related work. Finally, Section 7 raises some concluding remarks and open perspectives.

Notations. In this paper, we use the following notations. For two domains of elements E and F, we note $[E \to F]$ (resp. $[E \rightharpoonup F]$) the set of functions (resp. partial functions) from E to F. When elements of E depends on the elements of F, we note $\{e \in E\}_{f \in F'}$, where $F' \subseteq F$, for $\{e \in E \mid f \in F'\}$ or $\{e\}_{f \in F'}$ when clear from context.

2 BIP - Behavior Interaction Priority

In this section we recall the necessary concepts of the BIP framework [5]. BIP is a component-based framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. The *behavior* layer consists of a set of atomic components represented by transition systems. The *interaction* layer models the collaboration between components. Interactions are described using sets of ports and connectors between them [6]. The *priority* layer is used to enforce scheduling policies applied to the interaction layer, given by a strict partial order on interactions.

2.1 Component-Based Construction

BIP offers primitives and constructs for modeling and composing complex behaviors from atomic components. Atomic components are Labeled Transition Systems (LTS) extended with C functions and data. Transitions are labeled with sets of communication ports. Composite components are obtained from atomic components by specifying connectors and priorities.

Atomic Components. An atomic component is endowed with a set of local variables X taking values in a domain $Data$. Atomic components synchronize and exchange data with other components through the notion of *port*.

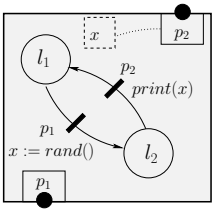
Definition 1 (Port). *A port $p[X']$, where $X' \subseteq X$, is defined by a port identifier p and some data variables in a set X' (referred as the support set).*

Definition 2 (Atomic component). *An atomic component B is defined as a tuple $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$, where:*

- *(P, L, T) is an LTS over a set of ports P. L is a set of control locations and $T \subseteq L \times P \times L$ is a set of transitions.*
- *X is a set of variables.*
- *For each transition $\tau \in T$:*
  - *g_τ is a Boolean condition over X: the guard of τ,*
  - *$f_\tau \in \{x := f^x(X) \mid x \in X\}^*$: the computation step of τ, a list of statements.*

For $\tau = (l, p, l') \in T$ a transition of the internal LTS, l (resp. l') is referred as the source (resp. destination) location and p is a port through which an interaction with another component can take place. Moreover, a transition $\tau = (l, p, l') \in T$ in the internal LTS involves a transition in the atomic component of the form $(l, p, g_\tau, f_\tau, l')$ which can be executed only if the guard g_τ evaluates to `true`, and f_τ is a computation step: a set of assignments to local variables in X. In the rest of this article, we use the dot notation to denote the elements of atomic components, e.g., $B.P$ denotes the set of ports of an atomic component B.

Example 1 (Atomic component). The figure below shows an example of atomic component with two ports p_1, p_2, a variable x, and two control locations l_1, l_2.



At location l_1, the transition labelled by the port p_1 is possible (the guard evaluates to `true` by default). When an interaction through p_1 takes place, a random value is assigned to the variable x through $x := rand()$. From the control location l_2, the transition labelled by the port p_2 is possible, the variable x is not modified, the value of x is printed and exported through p_2.

Semantics of Atomic Components. The semantics of an atomic component is an LTS over configurations and ports, formally defined as follows:

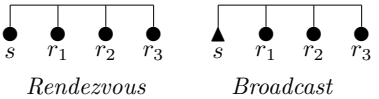
Definition 3 (Semantics of Atomic Components). *The semantics of the atomic component* $(P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ *is an LTS* (P, Q, T_0) *s.t.*

- $Q = L \times [X \rightarrow Data]$,
- $T_0 = \{((l, v), p, (l', v')) \in Q \times P \times Q \mid \exists \tau = (l, p, l') \in T : g_\tau(v) \wedge v' = f_\tau(v)\}$.

A configuration is a pair $(l, v) \in Q$ where $l \in L$ is a control location, and $v \in [X \rightarrow Data]$ is a valuation of the variables in X. The evolution of configurations $(l_1, v) \overset{p(v_p)}{\rightarrow} (l_2, v')$, where v_p is a valuation of variables attached to port p, is possible if there exists a transition $(l_1, p[x_p], g_\tau, f_\tau, l_2)$, s.t. $g_\tau(v) = \texttt{true}$. As a result, the valuation v of variables is modified to $v' = f_\tau(v[x_p \leftarrow v_p])$.

Creating composite components. Assuming some available atomic components B_1, $\ldots$, B_n, we show how to connect $\{B_i\}_{i \in I}$ with $I \subseteq [1, n]$ using *connectors*.

A connector γ is used to specify possible interactions, i.e., the sets of ports that have to be jointly executed. Two types of ports (*synchron*, *trigger*) are defined, in order to specify the feasible interactions of a connector. A *trigger* port is active: it can initiate an interaction without synchronizing with other ports. It is graphically represented by a triangle. A *synchron* port is passive: it needs synchronization with other ports for initiating an interaction. It is graphically represented by a circle. A feasible interaction of a connector is a subset of its ports s.t. either it contains some trigger, or it is maximal.



The figure on the left shows two connectors: *Rendezvous* (only the maximal interaction $sr_1r_2r_3r_4$ is possible), *Broadcast* (all the interactions containing the trigger port s are possible).

Formally, a connector is defined as follows:

Definition 4 (Connector). *A connector* γ *is a tuple* $(\mathcal{P}_\gamma, t, G, F)$*, where:*

- $\mathcal{P}_\gamma = \{p_i[x_i] \mid p_i \in B_i.P\}_{i \in I}$ *s.t.* $\forall i \in I : \mathcal{P}_\gamma \cap B_i.P = \{p_i\}$,
- $t : \mathcal{P}_\gamma \rightarrow \{\texttt{true}, \texttt{false}\}$ *s.t.* $t(p) = \texttt{true}$ *if* p *is trigger (and* `false` *if synchron),*
- G *is a Boolean expression over the set of variables* $\cup_{i \in I}\, x_i$ *(the guard),*
- F *is an update function defined over the set of variables* $\cup_{i \in I}\, x_i$.

$\mathcal{P}_\gamma$ is the set of connected ports called the support set of γ. The ports in $\mathcal{P}_\gamma$ are tagged with function t indicating whether they are trigger or synchron. Moreover, for each $i \in I$, x_i is a set of variables associated to the port p_i.

A communication between the atomic components of $\{B_i\}_{i \in I}$ through a connector $(\mathcal{P}_\gamma, G, F)$ is defined using the notion of *interaction*:

Definition 5 (Interaction). *A set of ports $a = \{p_j\}_{j \in J} \subseteq \mathcal{P}_\gamma$ for some $J \subseteq I$ is an interaction of γ if one of the following conditions holds: (1) there exists $j \in J$ s.t. p_j is trigger; (2) for all $j \in J$, p_j is synchron and $\{p_j\}_{j \in J} = \mathcal{P}_\gamma$.*

An interaction a has a guard and two functions G_a, F_a, respectively obtained by projecting G and F on the variables of the ports involved in a. We denote by $\mathcal{I}(\gamma)$ the set of interactions of γ. Synchronization through an interaction involves two steps. First, the guard G_a is evaluated, then the update function F_a is applied. If there are several possible interactions inside a connector, we choose the interaction involving the maximum[1] number of ports. One can also add priorities to reduce non-determinism whenever several interactions are enabled. Then, the interaction with the highest priority is chosen.

Definition 6 (Composite Component). *A composite component is defined from a set of available atomic components and a set of connectors. The connection of the $\{B_i\}_{i \in I}$ using the set Γ of connectors is denoted $\Gamma(\{B_i\}_{i \in I})$.*

Note that a composite component obtained by composition of a set of atomic components can be composed with other components in a hierarchical and incremental fashion using the same operational semantics.

Definition 7 (Semantics of Composite Components). *A state q of a composite component $C = \Gamma(B_1, \ldots, B_n)$, where Γ connects the B_i's for $i \in I$, is an n-tuple $q = (q_1, \ldots, q_n)$ where $q_i = (l_i, v_i)$ is a state of B_i. Thus, the semantics of C is precisely defined as a transition system $(Q, A, \longrightarrow)$, where:*

- *$Q = B_1.Q \times \ldots \times B_n.Q$,*
- *$A = \cup_{\gamma \in \Gamma}\{a \in \mathcal{I}(\gamma)\}$ is the set of all possible interactions,*
- *$\longrightarrow$ is the least set of transitions satisfying the following rule:*

$$\frac{\exists \gamma \in \Gamma : \gamma = (P_\gamma, G, F) \qquad \exists a \in \mathcal{I}(\gamma) \qquad G_a(v(X)) \qquad \forall i \in I :\ q_i \xrightarrow{p_i(v_i)}_i q_i' \wedge v_i = F_{ai}(v(X)) \qquad \forall i \notin I :\ q_i = q_i'}{(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')}$$

where $a = \{p_i\}_{i \in I}$, X is the set of variables attached to the ports of a, v is the global valuation of variables, and F_{ai} is the partial function derived from F restricted to the variable associated to p_i.

The meaning of the above rule is the following: if there exists an interaction a s.t. all its ports are enabled in the current state and its guard ($G_a(v(X))$) evaluates to `true`, then we can fire the interaction. When a is fired, not involved components stay in the same state, and, involved components evolve according to the interaction.

Notice that several distinct interactions can be enabled at the same time, thus introducing non-determinism in the product behavior, possibly restricted using priorities.

Definition 8 (Priority). *Let $C = (Q, A, \longrightarrow)$ be the behavior of the composite component $\Gamma(B_1, \ldots, B_n)$. A* priority model *π is a strict partial order on the set of interactions A. Given a priority model π, we abbreviate $(a, a') \in \pi$ to $a \prec a'$. The component $\pi(C)$ is defined by the behavior $(Q, A, \longrightarrow_\pi)$, where $\longrightarrow_\pi$ is the least set of transitions satisfying the following rule:*

[1] If there are several maximal interactions, the choice between them is at random.

$$\frac{q \xrightarrow{a} q' \qquad \nexists a' \in A, \nexists q'' \in Q : a \prec a' \wedge q \xrightarrow{a'} q''}{q \xrightarrow{a}_{\pi} q'}$$

An interaction is enabled in $\pi(C)$ only if it is enabled in C, and, it is maximal according to π among the active interactions in C.

Finally, we consider systems defined as a parallel composition of components together with an initial state.

Definition 9 (System). *A system $\mathcal{S}$ is a pair $(B, Init)$ where B is a component and $Init$ is the initial state of B.*

3 An RV Framework for Component-Based Systems

We adapt classical RV frameworks dedicated to monitoring of sequential monolithic programs to CBS in general, and, to BIP systems in particular. We consider a composite component $C = \Gamma(B_1, \ldots, B_n)$ and a priority model π, where the runtime semantics of $\pi(C)$ is an LTS $(Q, A, \longrightarrow_{\pi})$ as introduced in Definitions 7 and 8.

3.1 Specifications for Component-Based Systems (CBS)

Considered specifications of CBS are state-based specifications expressing some desired behavior. We do not assume any particular specification formalism except that we require it expresses a subset of the possible linear behaviors of CBS. In order to make our approach as general as possible, we only describe the events of the possible specification language. We also assume the existence of a monitor synthesis algorithm from this specification formalism (see Section 3.2). For this purpose, the existing solutions (e.g., [7]) provided by the research efforts in RV can be easily adapted.

We follow a classical approach where events are built over a set of atomic propositions $Prop$. Intuitively, an atomic proposition is a Boolean expression over the states of the components (e.g., "in the component B_1, the variable x should be positive if in the component B_2 the variable y is negative"). More formally, an event of $\pi(C)$ is defined as a state formula over the atomic propositions expressed on components involved in $\pi(C)$. The set of events is defined with the following grammar:

$$\begin{aligned}
\Sigma(\pi(C)) &: \text{Atom} \vee \text{Atom} \mid \text{Atom} \wedge \text{Atom} \mid \text{Atom} \Rightarrow \text{Atom} \mid \neg\ \text{Atom} \\
\text{Atom} &: \text{component.var} == \text{val} \mid \text{component.var} \geq \text{val} \\
&\quad \mid \text{component.loc} == \text{a_location} \mid \text{component.port} == \text{a_port} \\
\text{component.var} &: \cup_{i \in [1,n]} B_i.X \\
\text{val} &: v \in Data \\
\text{a_location} &: s \in \cup_{i \in [1,n]} B_i.L \\
\text{a_port} &: p \in \cup_{i \in [1,n]} B_i.P
\end{aligned}$$

Let us note $Prop(e)$ the set of atomic propositions used in an event $e \in \pi(C)$. For $ap \in Prop(e)$ we define $used(ap)$ as the set of pairs made of a component and variables that are used to define ap:

used(ap) = match (ap) with
component.var == val → (component,var)
component.var ≥ val → (component,var)
component.loc == a_location → (component,loc)
component.port == a_port → (component,port)

3.2 Verification Monitors [4]

A monitor is a procedure consuming events fed by a BIP system and producing an appraisal on the sequence of events read so far. We follow a general approach considering verification monitors as deterministic finite-state machines producing a truth-value (a verdict) in an expressive 4-valued truth-domain $\mathbb{B}_4 \stackrel{\text{def}}{=} \{\bot, \bot_c, \top_c, \top\}$, introduced in [3] and used in [4]. $\mathbb{B}_4$ consists of the possible evaluations of a sequence of events and its possible futures relatively to the specification used to generate the monitor:

- The truth-value $\top_c$ (resp. $\bot_c$) denotes "currently true" (resp. "currently false") and expresses the satisfaction (resp. violation) of the specification "if the system execution stops here".
- The truth-value $\top$ (resp. $\bot$) is a definitive verdict denoting the satisfaction (resp. violation) of the specification: the monitor can be stopped.

We define the notion of monitor for a specification defined relatively to a set of events Σ expressed on a composite component.

Definition 10 (Monitor). *A* monitor $\mathcal{A}$ *is a tuple* $(\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$. *The finite set* $\Theta^{\mathcal{A}}$ *denotes the control states and* $\theta^{\mathcal{A}}_{\text{init}} \in \Theta^{\mathcal{A}}$ *is the initial state. The complete function* $\longrightarrow_{\mathcal{A}}$: $\Theta^{\mathcal{A}} \times \Sigma \to \Theta^{\mathcal{A}}$ *is the transition function. In the following we abbreviate* $\longrightarrow_{\mathcal{A}} (\theta, a) = \theta'$ *by* $\theta \stackrel{a}{\longrightarrow}_{\mathcal{A}} \theta'$. *The function* $ver^{\mathcal{A}} : \Theta^{\mathcal{A}} \to \mathbb{B}_4$ *is an output function, producing verdicts (i.e., truth-values) in* $\mathbb{B}_4$ *from control states.*

Such monitors are independent from any specification formalism used to generate them and are able to check any specification expressing a linear temporal specification [4]. Intuitively, runtime verification of a specification with such monitors works as follows. An execution sequence is processed in a lock-step manner. On each received event, the monitor produces an appraisal on the sequence read so far. For a formal presentation of the semantics of the monitor and a formal definition of sequence checking, we refer to [4]. In the remainder, we consider a monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$.

3.3 Runs and Traces of BIP Systems

Runs of BIP systems. Each state $q \in Q$ in the LTS of a component can be seen as an environment mapping variables used in the specification over an alphabet Σ to values.

Definition 11 (Environments in a component). *The set of possible environments in* $\pi(C)$ *is* $Env \stackrel{\text{def}}{=} \cup_{i\in[1,n]}(\{state_i \to Q^i\} \cup [B_i.X \to Data])$. *The environment defined by a state* $q = (q_1, \ldots, q_n)$, *where* $q_i = (l_i, v_i)$ *for each* $i \in [1, n]$, *is* $[\![q]\!] \in Env$ *s.t.* $[\![q]\!] \stackrel{\text{def}}{=} \cup_{i\in[1,n]}(\cup_{x_i \in B_i.var}\{x \mapsto v_i(x)\} \cup \cup_{i\in[1,n]}\{state^i \mapsto l_i\}$.

After an interaction bringing the component in a state q, an event e is fired if the state-formula associated to e holds, noted $q \vDash e$, i.e., when e evaluates to true in $[\![q]\!]$, i.e.,

$[\![q]\!](e) = \texttt{true}$. Let us note that, after reaching a state of the LTS corresponding to the runtime behaviors of a BIP component, it is always possible to determine whether an event is fired or not, i.e., whether the corresponding state-formula holds or not.

We present the notion of *run* of a composite component and how it is monitored.

Definition 12 (Run of a composite component). *A run of length m of a system $(\pi(C), Init)$ is the sequence of environments $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ s.t.: $q^0 = Init$, and, $\forall i \in [0, m-1] : q^i \in Q \wedge \exists a_i \in A : q^i \xrightarrow{a_i}_{\pi} q^{i+1}$.*

Definition 13 (Monitoring a run of a system). *The verdict $[\![\mathcal{A}]\!](q^0 \cdot q^1 \cdots q^m)$ stated by $\mathcal{A}$ for a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ is $ver^{\mathcal{A}}(\theta^m)$ where $\forall i \in [0, m-1] : \theta_i \xrightarrow{e}_{\mathcal{A}} \theta_{i+1}$ where e is the unique event enabled in θ_i s.t. $q^{i+1} \models e$.*

Building a trace from a run. As one of the challenges in RV is to lower the performance impact on the target program, we should take care of minimizing the information sent to the monitor. Making the monitor processing directly the run of the target program directly is not a reasonable solution because it would yield prohibiting overhead. Our proposal is to send a relevant abstraction of the run to the monitor that we call a *trace*. Intuitively, given a run, the obtained trace is its minimal abstraction (information wise) that permits to evaluate the specification as if the run was not abstracted. Given $Spec(\Sigma)$, a specification defined over a vocabulary of events Σ, we design an abstraction function $\downarrow_{\alpha}^{\Sigma}$ building this minimal abstraction. We thus define a notion of informativeness of environments built from states. Intuitively, an environment ρ_1 is less informative than an environment ρ_2 if it has less variables defined, i.e., $\rho_1 \sqsubseteq \rho_2$ if $Dom(\rho_1) \subseteq Dom(\rho_2)$ and $\forall x \in Dom(\rho_1) : \rho_1(x) = \rho_2(x)$. When monitoring a CBS our aim will be to dynamically build the least informative environment so that the monitoring activity of the system amounts to monitoring with the global state.

Definition 14 (Abstraction function). *The abstraction function $\downarrow_{\alpha}^{\Sigma}$: $Q \rightarrow Env$ is the least function s.t.: $\forall q \in Q : \downarrow_{\alpha}^{\Sigma}(q) = \rho$ and ρ is s.t.: $\forall x \in Dom([\![q]\!])$:*

$$\rho(x) = \begin{cases} [\![q]\!](x) & \text{if } \exists e \in \Sigma, \exists ap \in Prop(e) : used(ap) = (B_i, x), \text{ with } x \in B_i.X; \\ undef & \text{otherwise.} \end{cases}$$

Property 1 (Abstraction preserves event evaluation). The previous abstraction function adheres to the two following principles:

- soundness: $\forall e \in \Sigma, \forall q \in Q : \downarrow_{\alpha}^{\Sigma}(q) \models e \Leftrightarrow q \models e$,
- completeness: $\forall e \in \Sigma, \forall q \in Q : \downarrow_{\alpha}^{\Sigma}(q) \models e \vee \downarrow_{\alpha}^{\Sigma}(q) \not\models e$.

Soundness states that the concrete and abstracted evaluations are the same. Completeness states that evaluation of all specification events remains possible: abstraction does not erase the needed information from the environment.

Definition 15 (Trace of a composite component). *The trace defined from a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$ through an abstraction function $\downarrow_{\alpha}^{\Sigma}$ is the sequence of environments defined as $\downarrow_{\alpha}^{\Sigma}(q^0) \cdot \downarrow_{\alpha}^{\Sigma}(q^1) \cdots \downarrow_{\alpha}^{\Sigma}(q^m)$.*

The notion of trace evaluation by a monitor directly follows from the notion of run evaluation. Moreover, the following theorem, which is a direct consequence of Property 1, states that, for runtime verification, there is no difference regarding property evaluation to process the trace instead of the run.

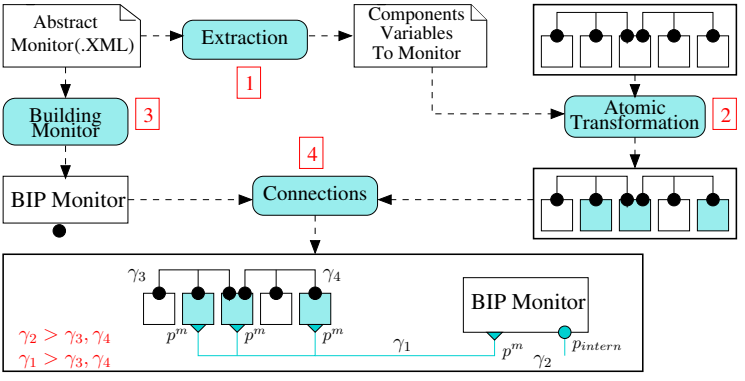


Fig. 1. Overview of the work-flow

Theorem 1 (Trace evaluation vs run evaluation by a monitor). *For $\mathcal{A}$ defined on Σ, the abstraction function $\downarrow_\alpha^\Sigma$, and a run $[\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]$, we have:*

$$[\![\mathcal{A}]\!]([\![q^0]\!] \cdot [\![q^1]\!] \cdots [\![q^m]\!]) = [\![\mathcal{A}]\!](\downarrow_\alpha^\Sigma(q^0) \cdot \downarrow_\alpha^\Sigma(q^1) \cdots \downarrow_\alpha^\Sigma(q^m))$$

In the next section, we will instrument BIP systems in such a way that, given a specification, the minimal abstraction function (information-wise) is dynamically generated.

4 Verifying the Runtime Behavior of BIP Systems

This section presents how we instrument and integrate an abstract monitor $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$ into a BIP system made of a composite component $C = \Gamma(B_1, \ldots, B_n)$ and priority rules π. The work-flow is as follows (see Fig. 1):

1. From the input abstract monitor we extract the list of components and their corresponding variables used by the monitor (Section 4.1).
2. For each component and its corresponding variables extracted from the monitor we instrument the selected components so as to observe them (Section 4.2).
3. From the monitor we generate the corresponding atomic component. Then, we add the generated component (a monitor in BIP) to the input composite component (Section 4.3).
4. Finally, we add the new connections between the instrumented atomic components and the monitor in BIP (Section 4.4).

4.1 Extraction of Needed Information

The first step is to retrieve from the monitor the set of components and their corresponding variables that should be monitored. For each selected component, transitions are instrumented to observe the just needed set of variables. For a specification expressed over $\Sigma(\pi(\Gamma(B_1, \ldots, B_n)))$ and its monitor, $comp(\Sigma)$ is the subset of $\cup_{i\in[1,n]}\{B_i\}$ corresponding to the set of components that should be monitored. We also define $occur(\Sigma)$

to be the subset of $\{B_i.loc \mid i \in [1,n]\} \cup \{B_i.port \mid i \in [1,n]\} \cup \cup_{i\in[1,n]} B_i.X$ denoting the set of variables used in the specification. Then from $occur(\Sigma)$, we sort the variables according to the component B_i (where $B_i \in comp(\Sigma)$) they are related to: $c\_v() = [1,n] \rightarrow \{B_i.loc\}_{i\in[1,n]} \cup \{B_i.port\}_{i\in[1,n]} \cup \cup_{i\in[1,n]} B_i.X$ s.t. $c\_v(i)$ is the set of variables related to component B_i.

4.2 Instrumentation of Atomic Component

For a composite component $\Gamma(B_1,\ldots,B_n)$, we transform each atomic component B_i, $i \in [1,n]$, so that it is able to interact with the monitor, if necessary.

Definition 16 (Instrumenting atomic components). *Given* $B = (P, L, T, X, \{g_\tau\}_{\tau\in T}, \{f_\tau\}_{\tau\in T})$ *s.t.* $B = B_i \in \{B_1,\ldots,B_n\}$, *we define a new atomic component*

$$B^m = \begin{cases} B & \text{if } B \notin comp(\Sigma) \\ (P^m, L^m, T^m, X^m, \{g_\tau\}_{\tau\in T^m}, \{f_\tau\}_{\tau\in T^m}) & \text{otherwise} \end{cases}$$

where, $(P^m, L^m, T^m, X^m, \{g_\tau\}_{\tau\in T^m}, \{f_\tau\}_{\tau\in T^m})$ *is defined as follows:*

- $X^m = X \cup \{loc \mid B_i.loc \in c\_v(i)\} \cup \{port \mid B_i.port \in c\_v(i)\}$;
- $P^m = P \cup \{p^m[c\_v(i)]\}$,
- $L^m = L \cup \{l_\tau\}_{\tau\in inst(T)}$, *where* $inst(T)$ *is defined as follows:*

$$inst(T) = \begin{cases} T & \text{if } \{B_i.loc, B_i.port\} \cap c\_v(i) \neq \emptyset \\ \{\tau \in T \mid c\_v(i) \cap var(f_\tau) \neq \emptyset\} & \text{otherwise} \end{cases}$$

- $T^m = T \setminus inst(T) \cup \{in(\tau) \mid \tau \in inst(T)\} \cup \{out(\tau) \mid \tau \in inst(T)\}$, *where,*
  - $in(\tau) = (l, p, f_{in(\tau)}, g_\tau, l_\tau)$, *where*

$$f_{in(\tau)} = \begin{cases} f_\tau & \text{if } B_i.loc \notin c\_v(i) \wedge B_i.port \notin c\_v(i) \\ f_\tau; [loc := \text{“}l\text{”}] & \text{if } B_i.loc \in c\_v(i) \wedge B_i.port \notin c\_v(i) \\ f_\tau; [port := \text{“}p\text{”}] & \text{if } B_i.loc \notin c\_v(i) \wedge B_i.port \in c\_v(i) \\ f_\tau; [loc := \text{“}l\text{”}; port := \text{“}p\text{”}] & \text{if } B_i.loc \in c\_v(i) \wedge B_i.port \in c\_v(i) \end{cases}$$

  - $out(\tau) = (l_\tau, p^m, f_{out(\tau)}, \texttt{true}, l')$, *where* $f_{out(\tau)} = []$.

We note $B^m = Instrum(B)$. In X^m, *loc* and *port* are variables containing a location name and a port name respectively. In P^m, p^m designates the port created for interacting with the monitor. Moreover, $inst(T)$ is the set of transitions that should be instrumented: we instrument atomic components whose variables are needed by the monitor. T^m designates the transitions in the instrumented atomic component. We instrument the transitions in the corresponding atomic component that are modifying a variable involved with the monitor. If the state or the port of an atomic component is needed, all transitions are instrumented. For each transition $\tau \in inst(T)$ that should be instrumented we add a new transition to interact with the monitor. Transitions are also instrumented by adding new statements to save the state and the port name, if necessary.

Example 2 (Instrumentation of an atomic component). Figure 2 illustrates the instrumentation of the atomic component depicted on the left-hand side into the instrumented component on the right-hand side. For instance, supposing that the state should be monitored, from the transition $\tau_0 = (l_0, p_1, f_{\tau_0}, \texttt{true}, l_1)$ with $f_{\tau_0} = [\text{done} := 0]$, we create a new state l_{τ_0} and the transitions $in(\tau_0) = (l_0, p_1, f_{in}, \texttt{true}, l_{\tau_0})$ with $f_{in} = [done := 0; loc := \text{“}l0\text{”}; port := \text{“}p_1\text{”}]$, and $out(\tau_0) = (l_{\tau_0}, p_1, [], \texttt{true}, l_1)$.

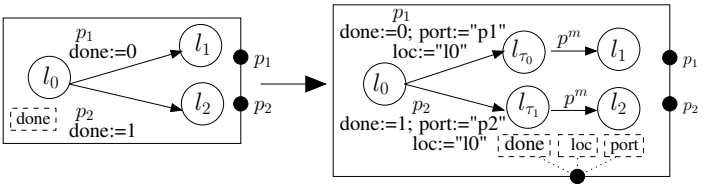


Fig. 2. Instrumentation of an atomic component

4.3 Creating an Atomic Component from a Monitor

From an abstract monitor (cf. Definition 10) given as an XML file, we construct the corresponding atomic component in BIP that interacts with the instrumented atomic components and produces verdicts following the behavior of the original monitor.

Definition 17 (Building monitors in BIP). *From a monitor* $\mathcal{A} = (\Theta^{\mathcal{A}}, \theta^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \mathbb{B}_4, ver^{\mathcal{A}})$*, we define the corresponding atomic component* $M^{\mathcal{A}} = (P, L, T, X, \{g_\tau\}_{\tau \in T}, \{f_\tau\}_{\tau \in T})$ *as an atomic component implementing its behavior:*

- $P = \{p^m[X], p_{intern}[]\}$,
- $L = \Theta^{\mathcal{A}} \cup \{q_{mi}\}_{q_i \in \Theta^{\mathcal{A}}}$,
- $T = T_1 \cup T_2$, *where*
  - $T_1 = \{(q_i, p^m, [], \texttt{true}, q_{mi}) \mid q_i \in \Theta^{\mathcal{A}}\}$,
  - $T_2 = \{(q_{mi}, p_{intern}, a, \texttt{print}(ver^{\mathcal{A}}(q'_i)), q'_i) \mid q_i \xrightarrow{a}_{\mathcal{A}} q'_i \wedge (q_i, p_M, q_{mi}) \in T_1\}$,
- $X = occur(\Sigma)$.

We note $M^{\mathcal{A}} = BuildMon(\mathcal{A})$ and call $M^{\mathcal{A}}$ a BIP monitor. T_1 denotes the set of transitions interacting with the composite component. T_2 is the set of transitions used to display verdicts following the behavior of the original monitor $\mathcal{A}$. The set of variables of the monitor is the set of variables used in the specification (as in Section 4.1).

Example 3 (Transforming an abstract monitor into a BIP monitor). Fig. 3 illustrates the transformation of Definition 17. The atomic component in Figure 3(a) is transformed into the BIP monitor in Figure 3(b).

4.4 Connections

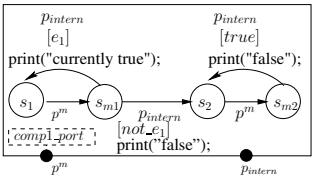
The next step of our transformation is to define the connectors between the transformed components B^m and the BIP monitor $M^{\mathcal{A}}$.

Definition 18 (Connections). *Given* $\mathcal{A}$ *and* $\pi(\Gamma(B_1, \ldots, B_n))$*, the monitored composite component is* $\pi^m(\Gamma^m(B_1^m, \ldots, B_n^m, M^{\mathcal{A}}))$*, where:*

- $B_i^m = Instrum(B_i)$*, for* $i \in [1, n]$*, (see Definition 16);*
- $M^{\mathcal{A}} = BuildMon(\mathcal{A})$*, (see Definition 17);*
- $\Gamma^m = \gamma \cup \{\gamma_1 = (P_{\gamma_1}, \texttt{true}, F_{\gamma_1}), \gamma_2 = (M^{\mathcal{A}}.p_{intern}, \texttt{true}, \emptyset)\}$*, where,*
  - $P_{\gamma_1} = \{B_i.p^m[X_i^m]\}_{B_i \in comp(\Sigma)}\} \cup \{M^{\mathcal{A}}.p^m\}$*, where all ports are synchron;*
  - F_{γ_1}*, the update function, is the identity data transfer from the variables in the ports of the interacting components* B_i *(*$i \in 51, n$*]) to the corresponding variables in the monitor port;*
  - *the type of the port* $M^{\mathcal{A}}.p_{intern}$ *in the connector* γ_2 *is synchron (one and only one interaction is defined by this connector:* γ_2*, see Definition 5);*

```
<VerificationMonitor>
   <State id="s1" initial="true">
      <Transition event="e1" nextState="s1" output="currently_true"/>
      <Transition event="not_e1" nextState="s2" output="false"/>
   </State>
   <State id="s2">
      <Transition event="true" nextState="s2" output="false"/>
   </State>
</VerificationMonitor>
```

(a) Abstract Monitor



(b) BIP Monitor

Fig. 3. Transforming an abstract monitor into a BIP Monitor

– $\pi^m = \pi \cup \{a \prec a' \mid a \in \cup_{\gamma \in \Gamma} \mathcal{I}(\gamma) \wedge a' \in \mathcal{I}(\gamma_1) \cup \mathcal{I}(\gamma_2)\}$.

The interactions defined by γ_1 and γ_2 have more priority than those defined by Γ (illustrated in Fig. 1). It ensures that, after execution of an interaction by the involved components, the monitor produces verdict before involving other interactions.

4.5 Summary and Discussion

We propose a 4-stage approach to introduce runtime verification for CBS. Our method directly integrates an abstract monitor in a CBS. Thanks to the BIP framework, monitoring of a specification can be taken into account at design stage. Moreover, the actual system, automatically generated from the augmented BIP model, is runtime-checked.

The correctness proof is omitted due to the lack of space, and, relies on the following informal arguments. Our transformations do not modify the data nor the behavior induced by the initial interactions. No deadlock is introduced because the synthesized BIP monitor is always ready to receive events from the instrumented components. Finally, the priorities introduced when connecting the instrumented components to the BIP monitor (Section 4.4) guarantee that the monitor always receives fresh data, i.e., the latest system state.

5 Implementation and Evaluation

5.1 RV-BIP: A Tool for Runtime Verification of BIP Systems

RV-BIP is a Java implementation ($\sim$ 2500 LOC) of the transformations described in Section 4, and, is part of the BIP distribution. RV-BIP takes as input a BIP system and an abstract monitor (an XML file) and then outputs a new BIP system whose behavior is monitored. It uses the following modules (see Fig. 1):

– *Extraction:* this module extracts the components and the corresponding variables used in the monitor. It takes as input an abstract monitor and then outputs a list of components with their corresponding variables,
– *Atomic Transformation:* this module instruments the atomic components selected from the extraction module. It takes as input the output of the *Extraction* module and a BIP file containing the original BIP system,
– *Building Monitor:* this module takes as input an abstract monitor and then outputs the corresponding atomic component,
– *Connections:* this module constructs the new composite component whose behavior is monitored. It takes as input the output from the *Atomic Transformation* and *Building Monitor* modules and then outputs a new composite component.

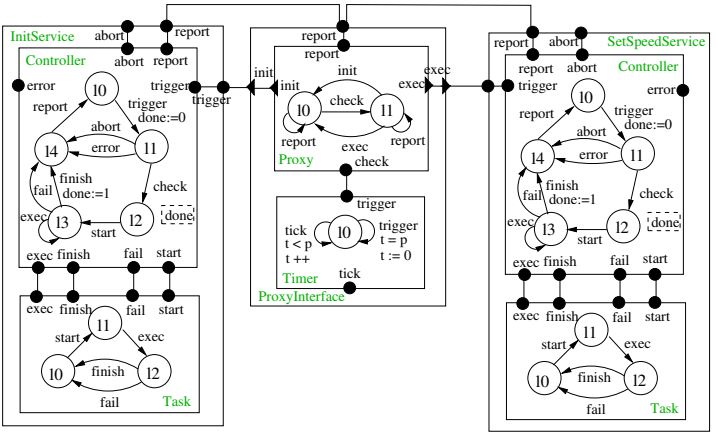


Fig. 4. Two services involving the ordering specification

5.2 Case Study: A Robotic Application

We experimented RV-BIP on a robotic application modeled in BIP: Dala robot [8,9]. The Dala robot is a large and realistic interactive system. It is an infinite system (in terms of states and transitions) that cannot be model-checked.

The functional level of the Dala robot consists of a set of modules. A module is composed of a set of services corresponding to different tasks and a set of posters where the produced data is stored and exchanged between different modules. In this section, due to the lack of space, we present a simplified model of the modules with only the services related to two properties among those we runtime checked.

Execution order: Figure 4 shows a simplified model of Dala. It consists of 3 components: *ProxyInterface*, *InitService* and *SetSpeedService*. *ProxyInterface* communicates with the control layer using the mailbox by executing the transition *check*. *InitService* is responsible for the initialization of the module and *SetSpeedService* performs the main task of the module. According to the received request, *Proxy* triggers either *InitService* or *SetSpeedService*. Each service has a status variable *done*: value 1 means that the corresponding task has been successfully executed. A service can be triggered through the port *trigger*, then it executes its task by taking the transition *start* and finally it returns to the initial location by the transition *finish* when the task is done. The execution order of some services are important. In this module, *InitService* initializes the robot and should be successfully executed before *SetSpeedService* sets the speed parameter of the robot. This requirement is formalized as "φ_1 and φ_2", see Table 1.

Data freshness: In Dala, the modules communicate by a set of posters. Data generated by a module is written in a poster that can be accessed by another module. The behavior of the robot might depend on this data, therefore it is necessary that the data is up to date: the data read by a service of a module (called $Reader$) must be fresh enough compared to the moment it has been written (by a service called $Writer$). If t_1 and t_2 respectively are the moments of reading and writing actions, then the difference between t_1 and t_2 must be less than a specific duration δ, i.e., $|t_2 - t_1| \leq \delta$.

Table 1. Formalization of the requirements for the Dala robot

φ_1: $(e_1)^*$, where, e_1: $(SetSpeedService.port == \text{“}trigger\text{”} \wedge ProxyInterface.port == \text{“}exec\text{”}) \Rightarrow (InitService.done == 1)$	
φ_3: $(e_1)^*$, where, e_1: $(Reader.port == \text{“}read\text{”} \wedge poster.port == \text{“}read\text{”} \wedge Clock.port == \text{“}getTime\text{”})$ $\Rightarrow (Clock.time - poster.wrtime \leq 2)$	
φ_2: $(e_1 e_2)*$, where, e_1: $InitService.port == \text{“}finish\text{”}$ e_2: $SetSpeedService.port == \text{“}trigger\text{”}$	φ_4: $(e_1(\epsilon + e_2 + e_2 e_2) e_3)^*$, where, e_1: $Writer.port == \text{“}write\text{”}$ e_2: $Clock.port == \text{“}tick\text{”}$ e_3: $Reader.read == \text{“}read\text{”}$

This requirement is formalized as "φ_3 and φ_4", see Table. 1. In the model, the time counter is implemented by a component $Clock$, and the $tick$ transition occurs every second.

Experiments: Table 2 reports results on checking the ordering and freshness properties of the Dala robot. *Ordering violated* and *Ordering guaranteed* correspond to the model presented in Fig. 4: the first one might have the violation of the ordering specification whereas the second one always guarantees it. Each consists of an *InitService*, a *SetSpeedService* and ten other services (corresponding to different tasks). It is similar for *Data freshness violated* and *Data freshness guaranteed*: the first might have the violation of the freshness specification whereas the second always guarantees it. We consider two modules: the first has a service responsible for writing data and five other services; the second has a service responsible for reading data produced by the first module and also five other services. In Table 2, *time-no-monitor* indicates the execution time without monitoring; *specification* is the monitored specification; the *optimized* column reports the execution time and the overhead obtained with the monitor that interacts only with the two components involved in the specification; and the *not-optimized* column reports the execution time and the overhead obtained with a monitor that observes all components of the system (even the ones that are not involved in the specification).

The results substantiate our claim that if we monitor only components involved in the specification, using the abstraction technique defined in Section 3 and implemented in Section 4, the overhead is reduced significantly.

6 Related Work

Static verification of component-based system. With the growing demand of scalability and complexity for systems, verification techniques should be used to determine whether a designed system meets its requirements. Static formal verification [10,11,12] is based on mathematical techniques to prove or disprove the correctness of a design w.r.t. a given formal specification. It searches for input patterns which lead to violations of the desired properties and prove the correctness when such violations do not exist. Existing formal verification methods for component-based systems are based on either static analysis or on model-checking [13,14,15].

Approaches based on static analysis consist in computing specific invariants in order to abstract the state space. Though this kind of approaches is less sensitive to state explosion, it still suffers from some limitations. First these techniques are rather limited in terms of the properties they can check: they are mostly limited to safety properties and

thus some interesting behavioral properties remains out of the scope of these techniques. Moreover, since these approaches rely on abstraction and over approximation of the state space, they yield several false positives.

Behavioral approaches such as model-checking are based on an exhaustive exploration of the state space of the model obtained from the operational semantics of the specification language. For large systems, this exploration leads to a very large number of states (the well-known state explosion problem). Despite recent advances in model-checking, the state-explosion problem is far from being solved and refrain the use of these methods in component-based systems where the state space tends to become huge due to the number of possible configurations and interactions between components. On the other hand, techniques based on compositional verification [16,17,18] (less sensitive to state explosion) are not applicable when the behavior of some parts of the system is unknown - as it can be the case in BIP when using external C functions.

A compositional verification method based on invariants for checking safety properties in component-based systems is provided in [19,20]. Although the method has been successfully applied to large-scale and complex systems, the use of invariants can deal only with safety properties and might produce many false positive counter examples.

Another compositional approach is design-by-contract [21,22] that considers a property provided by a component as a contract between this component and its environment. For instance [23] provides a method that searches an implementation model that satisfies a given contract. Although the experimental results are promising, it is not always possible to find an implementation model that satisfies a given property. Moreover, the composition of contracts in concurrent systems can be very expensive.

Dynamic verification of component-based systems. Specification and verification of the behavior of CBS have received some research endeavor. A first series of approaches specify the behavior of components in terms of pre and post-conditions (e.g., with JML) or assertions (e.g., using Eiffel). More recently and closer to our work is the LIME specification language [24] that allows runtime monitoring of temporal properties for component interfaces. Components are seen as black boxes and LIME specifications describe how components should interact with an external application by describing a desired behavior on the calls and returns over the interface.

Comparison with our approach. The limitations of static validation techniques lead us to investigate the use of *runtime verification* as an alternative and complementary technique to validate CBS. Compared to previous dynamic techniques, our approach offer several advantages. First, it uses the latest advances in runtime verification using an expressive 4-valued truth-domain allowing our monitor to be generated using any monitor synthesis framework. Note also that the proposed RV framework only uses information about the events used in the specification. The monitors presented in this paper are bounded to regular properties, however, the expressiveness of the BIP language confers our monitors a potential to be Turing-complete. Moreover, our approach is not limited to monitoring component interfaces. It is often the case that components come with an abstract behavioral model, i.e., components are gray boxes instead of black boxes. Our monitoring framework supports both kinds of approaches. Furthermore, the specifications considered for BIP systems use locations spanning over several components allowing the specification of global behaviors of the system in composition.

Table 2. Results of monitoring the requirements Execution order and Data freshness

	time-no-monitor	specification	optimized		not-optimized	
			time (s)	ovhd (%)	time (s)	ovhd (%)
Ordering violated	1.896	φ_1	2.045	7.8	9.163	383
		φ_2	1.953	3	9.192	384
Ordering guaranteed	1.836	φ_1	1.984	8.0	8.9	384
		φ_2	1.889	2.8	8.896	384
Data freshness violated	1.638	φ_3	1.684	2,8	4.337	164
		φ_4	1.682	2,6	3.773	130
Data freshness guaranteed	1.634	φ_3	1.678	2,6	4.383	168
		φ_4	1.690	3,4	3.782	131

7 Conclusion and Future Work

This paper introduces runtime verification as a complementary validation technique for component-based systems written in the BIP framework. Our technique is based on a general and expressive runtime verification framework. It dynamically builds a minimal abstraction of the current runtime state of the system so as to lower the performance impact. By generating monitors directly as BIP components, we are able to generate actual monitored C programs. Our approach has been implemented in RV-BIP that smoothly integrate in the existing BIP tool-set. Finally, experimental evaluations on a robotic application substantiate our claims and the feasibility of our approach.

Several research perspectives can be considered. A first direction is to combine the recent advances in RV that use static analysis (see e.g., [25]). In RV, using static analysis techniques may reduce the overhead induced by a monitor by disabling unnecessary runtime checks. Also related to overhead reduction, a dynamic instrumentation technique, enabling the monitor to remove connectors when they are not needed anymore, would reduce the overhead even more. Another possible direction is to extend the proposed framework for runtime enforcement [26]. Runtime enforcement is an extension of RV aiming at circumventing property violation and provides better confidence in system behaviors. A more practical direction is to connect RV-BIP to the various existing monitor synthesis tools available within the RV community.

References

1. Bliudze, S., Sifakis, J.: A Notion of Glue Expressiveness for Component-Based Systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 508–522. Springer, Heidelberg (2008)
2. Runtime Verification (2001-2010), http://www.runtime-verification.org
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. J. Log. Comput. 20, 651–674 (2010)
4. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: [27], pp. 40–59
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: 4th IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12 (2006)

6. Bliudze, S., Sifakis, J.: The algebra of connectors—structuring interaction in BIP. IEEE Transactions on Computers 57, 1315–1330 (2008)
7. Stolz, V.: Temporal assertions with parametrised propositions. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 176–187. Springer, Heidelberg (2007)
8. Fleury, S., Herrb, M., Chatila, R.: GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In: Proceedings of Intelligent Robots and Systems, IROS 1997, pp. 842–848. IEEE, Los Alamitos (1997)
9. Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., Nguyen, T.H.: Toward a more dependable software architecture for autonomous robots. IEEE Robotics and Automaton Magazine, Special issue on Soft. Engineering for Robotics 16, 67–77 (2008)
10. Umrigar, Z.D., Pitchumani, V.: Formal verification of a real-time hardware design. In: DAC 1983: Proceedings of the 20th Design Automation Conference, pp. 221–227. IEEE Press, Los Alamitos (1983)
11. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
12. Clarke, E.M., Emerson, E.A.: Synthesis of synchronisation skeletons for branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
13. McMillan, K.: Symbolic model checking. Kluwer Academic Publishers, Boston (1993)
14. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the 5th Syposium on Logic in Computer science, pp. 428–439 (1990)
15. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. 19, 7–34 (2001)
16. Clarke, E., Long, D., McMillan, K.: Compositional model checking. In: Proceedings of the 4th Annual Symposium on LICS, pp. 353–362. IEEE Computer Society Press, Los Alamitos (1989)
17. Chang, E., Manna, Z., Pnueli, A.: Compositional verification of real-time systems. In: Symposium on Logic in Computer Science. IEEE, Los Alamitos (1994)
18. Long, D.E.: Model Checking, Abstraction, and Compositional Reasoning. PhD thesis, Carnegie Mellon (1993)
19. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. Software Journal, Special Issue on Automated Compositional Verification 4, 181–193 (2010)
20. Bensalem, S., Bogza, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: FMCAD (2010)
21. Meyer, B.: Applying design by contract. Computer 25, 40–51 (1992)
22. Abadi, M., Lamport, L.: Composing specifications. ACM Transaction on Programming Languages and Systems 15, 73–132 (1993)
23. Ben-Hafaiedh, I., Graf, S., Quinton, S.: Reasoning about safety and progress using contracts. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 436–451. Springer, Heidelberg (2010)
24. Kähkönen, K., Lampinen, J., Heljanko, K., Niemelä, I.: The lime interface specification language and runtime monitoring tool. In: [27], pp. 93–100
25. Bodden, E., Lam, P., Hendren, L.J.: Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In: [28], pp. 183–197
26. Falcone, Y.: You should better enforce than verify. In: [28], pp. 89–105
27. Bensalem, S., Peled, D. (eds.): RV 2009. LNCS, vol. 5779. Springer, Heidelberg (2009)
28. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.): RV 2010. LNCS, vol. 6418. Springer, Heidelberg (2010)

Translating Alloy Specifications to UML Class Diagrams Annotated with OCL

Ana Garis[1], Alcino Cunha[2], and Daniel Riesco[1]

[1] Universidad Nacional de San Luis, San Luis, Argentina
{agaris,driesco}@unsl.edu.ar
[2] DI-CCTC, Universidade do Minho, Braga, Portugal
alcino@di.uminho.pt

Abstract. Model-Driven Engineering (MDE) is a Software Engineering approach based on model transformations at different abstraction levels. It prescribes the development of software by successively transforming models from abstract (specifications) to more concrete ones (code). Alloy is an increasingly popular lightweight formal specification language that supports automatic verification. Unfortunately, its widespread industrial adoption is hampered by the lack of an ecosystem of MDE tools, namely code generators. This paper presents a model transformation between Alloy and UML Class Diagrams annotated with OCL. The proposed transformation enables current UML-based tools to also be applied to Alloy specifications, thus unleashing its potential for MDE.

Keywords: MDE, Alloy, UML, OCL.

1 Introduction

Model-Driven Engineering (MDE) is a promising Software Engineering approach using models at different abstraction levels. Software is developed by successively transforming models from abstract to more concrete ones.

UML and OCL have been successfully adopted in the MDE context through the Model-Driven Architecture (MDA) initiative [15]. In order to support UML and OCL in MDE, different tools have been developed such as code generators and reverse engineering tools. Due to the informality and ambiguity of UML semantics it also has been combined with formal methods to increase the confidence in the software development process. Formal methods use mathematics for specification and design of models helping to discover inconsistencies in informal requirements. The main disadvantage of formal languages is that they require a learning effort and thus are frequently avoided by software engineers responding to time and cost constraints.

Alloy [12] is a lightweight formal language with a simple notation, easy to learn, easy to use, that includes a friendly Validation and Verification (V&V) tool. Its denotational language is based on first-order relational logic, with an object-oriented notation similar to UML and OCL [16]. The automatic Alloy

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 221–236, 2011.

Analyzer allows the generation of snapshots showing instances of the model as well as the execution of operations and assertion checking.

Although very few UML software developers are familiar with formal methods, Alloy could be easily adopted by UML practitioners due to its simplicity and its resemblance with UML. Both Alloy and UML can benefit if two-way transformations are developed between them. On the one hand, from the UML practitioners point of view, Alloy Analyzer could be exploited as a model verification tool in a MDE context. On the other hand, from the Alloy practitioners point of view, a myriad of UML tools could be used in order to unleash Alloy potential for MDE. Specifically, there exist multiple code generators to different platforms and programming languages, such as JEE, CORBA, Java, C, C++, C# and Python, that could be used to refine Alloy specifications.

Further benefits could be achieved by developers familiarized with both Alloy and UML. They could be combined in the software development process: start by using UML Class Diagrams to specify requirements at high abstraction level, then translate them to Alloy and formally specify invariants and operations, perform model validation and verification using Alloy Analyzer, and finally translate back to UML+OCL in order to use the aforementioned code generation tools.

This paper presents a model transformation from Alloy specifications to UML Class Diagrams annotated with OCL. Although the semantic correspondence between elements of UML and Alloy was already analyzed in [1], the translation from Alloy to UML+OCL has not been considered yet. This translation opens new challenges since several Alloy expressions do not have a direct equivalent in UML or OCL. Additionally, it requires us to explore the different Alloy idioms in order to identify a specification style compatible with UML+OCL models. Therefore, we define a subset of the Alloy language which includes UML+OCL compatible expressions, and we study the semantics of the syntactic elements of Alloy, UML Class Diagrams and OCL. We redefine the EBNF of Alloy grammar to recognize expressions in this subset and specify the transformation rules. Our approach is illustrated with a case study.

The rest of the document is structured as follows. Next section introduces preliminary concepts related to Alloy. After discussing some related work, the transformation from Alloy to UML+OCL is presented. Last section presents the conclusions and future work.

2 Alloy

Alloy is a formal language based on first-order relational logic [12]. It is supported by a SAT solver that enables model V&V. Alloy Analyzer is inspired by model checkers, but it is implemented as a solver. An Alloy module consists of a module header, a set of imports and zero or more paragraphs. The *module header* is a name of the module where signatures, constraints, assertions and commands are defined. An *import* allows to include additional modules. Furthermore, a *paragraph* can either be a signature declaration, a constraint, an assertion or a command.

A *signature declaration* represents a set of atoms. An atom is a unity with three fundamental properties: it is indivisible, immutable and uninterpreted. Optionally, a signature declaration can introduce *fields*. Fields represent sets of tuples of atoms and are interpreted as relations between signatures. *Constraints* are defined by *facts*, *predicates* and *functions*. Facts are invariants; i.e, their associated constraints always hold. Predicates are named constraints, which can be used in diverse contexts. The difference between a fact and a predicate is that the first one always holds while the second one only holds when invoked. Finally, functions describe named expressions, which can be also reused in diverse contexts. *Assertions* allow the expression of properties that are expected to hold as consequence of the stated facts. *Commands* are instructions to perform particular analysis. Alloy provides two commands for analysis: `run` and `check`. Command `run` gives instructions to analyzer to search for an instance of a given predicate, and command `check` to search for a counterexample of a given assertion.

Alloy's logic is quite generic and does not commit to a particular specification style. For example, since atoms are immutable there is no standard way to model the dynamic behavior of operations, and several idioms have been proposed to address this issue. One of the most popular is to introduce a signature denoting the overall state of the system, and model operations as predicates that specify the relationship between pre- and post-states. Two variants of this idiom are possible, known respectively as *global state* and *local state*. In the former all mutable fields are defined in the global state signature. In the later, the state signature is added locally as an extra column at the end of each mutable field. The local state idiom is more modular, since fields are still declared in the signature they naturally belong to. On the other hand, the global state idiom forces all dynamic fields to be artificially grouped together. The designation *local state* can be misleading, since the state is also global - the "local" concerns only the location it appears in field declarations.

In this paper we will assume the local state idiom to specify operations. Note that Alloy models conforming to the global state idiom can be easily converted to the local one using a simple refactoring [9]. Without loss of generality, we will also assume the distinguished state signature to be denoted as `Time` and all fields to be dynamic. An operation `op` will be specified using a predicate `pred op[...,t,t':Time] {...}` with two special parameters `t` and `t'` denoting, respectively, the pre- and post-state. Functions will be used to model queries that do not change the state. As such, only one of those special parameters is needed.

Figure 1 presents an example of an Alloy model conforming to the local state idiom. It is a variant of the address book model first presented in [12]. The `addr` field is a mutable relation that maps names to targets. A target is either an address or another name. Names are either groups or aliases. For each book, the first fact forces all names in the `addr` relation to be registered in the respective `names` relation.

In Alloy *everything is a relation*. For example, variables are just unary singleton relations. As such, the relational composition operator can be used for

```
module addressBook

sig Time {}

abstract sig Target { }
sig Addr extends Target { }
abstract sig Name extends Target { }
sig Alias, Group extends Name { }

sig Book {
   names: Name set -> Time,
   addr: Name -> some Target -> Time }

fact { all t:Time | all b:Book | b.addr.t.Target in b.names.t }

fact { all t:Time | all b:Book | no n: Name | n in n.^(b.(addr.t)) }

fact { all t:Time | all b:Book | all a: Alias | lone a.(b.addr.t)  }

pred add [b: Book, n: Name, a: Target, t,t': Time] {
   n in b.names.t
   b.addr.t' in b.addr.t + n->a }

fun lookup [b: Book, n: Name, t: Time] : set Addr {
   n.^(b.(addr.t)) & Addr }
```

Fig. 1. Address book example

various purposes. In particular, `b.addr.t` denotes the value of the relation `addr` of book `b` at instant `t`. Note that the relation `b.addr.t` has type `Name -> some Target`. If we compose it with the `Target` set, we get all names in the domain of that relation. The second fact uses the transitive closure operator to ensure that the `addr` relation is acyclic. The last fact limits the addresses of aliases to at most one target.

In the body of operations, constraints that do not refer to `t'` can be seen as pre-conditions. For example, `n in b.names.t` requires names to be registered before adding or removing new targets. Otherwise we have post-conditions. For example, expression `b.addr.t' = b.addr.t + n->a` states that, after executing operation `add`, relation `b.addr` should have at least one additional tuple. `lookup` models a query that returns the set of addresses of a given name. Again, transitive closure is used to recursively traverse relation `addr`. A desirable assertion in this model could be:

```
assert lookupYields {
   all t:Time, b:Book, n:b.names.t | some lookup[b,n,t]
}
check lookupYields for 4 but 1 Book
```

Unfortunately it does not hold, and the `check` command will produce a counter-example.

3 Alloy and UML+OCL Integration in MDE

The translation from Alloy to UML could foster the usage of Alloy in the MDE context. As mentioned before, a lot of UML tools have been developed to support MDA. However, even though OCL tools have improved in the last years, they still have several limitations regarding model V&V. Formal methods have been proposed as a valuable alternative to improve these limitations. In fact, they have been successfully integrated in the MDA, as shown in HOL-OCL [3], USE [11] or UML2Alloy [1].

Due to its suitability for V&V, Alloy is a better candidate for the early modeling phase of the software development process. After the validation process with Alloy, models can be translated to UML Class Diagrams and OCL in order to enable MDA-UML tools as well as MDA-OCL tools. Most of the UML tools allow transformations from UML Class Diagrams to different platforms and programming languages, such as JEE, CORBA, Java, C, C++, C# and Python. Additionally, there exist OCL tools for code generation, such as OCLtoSQL [4] and DresdenOCL [5].

3.1 Related Work

The relationship between UML and OCL with Alloy has been extensively studied by Anastaskis et al. [1], resulting in a prototype named UML2Alloy. This translation considers only the basic elements of UML Class Diagrams: classes, attributes and associations; and excludes interfaces, dependencies and signals.

Massoni et al. also propose a UML+OCL to Alloy translation in [13]. For classes, attributes and associations, they propose the same approach as Anastaskis et al. [1], but they also consider the translation of UML interfaces. The semantics of UML cannot be fully preserved since Alloy cannot represent some UML Class Diagram features such as visibility of attribute's properties.

UML has also been mapped to Alloy for model V&V of particular case-studies. We present three examples: the first one uses the Alloy Analyzer for formal security evaluation in a methodology called Aspect-Oriented Risk-Driven Development (AORDD) [7]; the second one describes a proposal for Alloy specification from Aspect-UML models, a UML Profile for extending UML with Aspect-oriented concepts [14]; the third one explains an approach to translate UML models, specified with OntoUML, for model validation using Alloy [2]. These examples make evident Alloy potential to V&V, but like in [13,1]; they consider the translation from UML+OCL to Alloy, not from Alloy to UML.

Shah et al. present a model transformation from Alloy to UML [18]. Specifically, they convert instances generated by Alloy Analyzer to a UML Object Diagram. This approach is possible in case UML2Alloy tool has been used before to generate an Alloy specification. The transformation is based on the original UML Class Diagram so it assumes that Alloy specifications must not change. Even though Shah et al. have exposed a way to translate from Alloy to UML, they only map model instances to UML Object Diagrams. Namely, they do not consider the translation from Alloy to OCL either.

4 Model Transformation from Alloy to UML+OCL

Although model transformations from UML to Alloy have already been defined [1,13], the opposite translation has not. Moreover it is characterized by new challenges, not addressed previously. An important issue to solve is the characterization of source models, namely identifying a subset of Alloy that can faithfully be translated to UML+OCL. We will first define this subset formally, and then present the translation of both model declarations and constraints. The address book example presented in Figure 1 will be used to illustrate our proposal.

4.1 Characterizing Source Models

The Alloy subset accepted by our model transformation is defined in Figure 2. This subset restricts models to conform to the local state idiom, informally introduced in Section 2. In particular, the model must declare a distinguished signature denoted `Time`, the last column of field declarations must be `Time`, all predicates must have `t` and `t'` as parameters, and all functions must have `t` as parameter. Note that `Time`, `t`, and `t'` are reserved keywords not allowed in *sigId* and *varId*, respectively. To simplify the presentation, we are assuming all relations to be mutable. Immutable relations (not including `Time` in their declaration) could also be handled by adding frame conditions to all method post-conditions in OCL, stating that their value remains constant. Besides these structural restrictions, the syntax of formulas is further restricted to ensure a sound operational semantics [10]:

- Facts must be of the form `all t:Time |` ϕ, with `t` the only time variable that occurs in formula ϕ. This ensures that they act as invariants, instead of arbitrary temporal formulas, and can thus be represented in OCL.
- Every relational expression occurring in a formula must be *state-bound*, in the sense that each mutable relation identifier is within scope of a time variable. To simplify the translation, we ensure this restriction by forcing relation identifiers to be composed with either `t` or `t'`. However, a more relaxed syntax can be defined, where each occurrence of a relation identifier is required to be a subterm of either Φ`.t` or Φ`.t'`, where Φ denotes a relational expression.

Besides conforming to the local state idiom, an Alloy model must satisfy some additional restrictions due to the limitations of UML+OCL, as described in the UML Class Diagram metamodel [17] and the OCL metamodel [16]:

- Signature declarations can only be top-level or extend other signatures. Signature inclusion, where `in` is used instead of `extends`, is not allowed since it is not possible to specify using UML class diagrams that a class is a non-disjoint subset of another class.
- Field declarations must refer signature identifiers instead of arbitrary relational expressions. This ensures that the type of each column corresponds to a single signature, instead of an arbitrary disjunction of signatures, as

$$
\begin{aligned}
\mathit{module} &::= \texttt{module}\ \mathit{moduleId}\ \texttt{sig Time}\ \{\}\ \mathit{sigDecl}^{+}\ \mathit{paragraph}^{*}\\
\mathit{paragraph} &::= \mathit{factDecl} \mid \mathit{funDecl} \mid \mathit{predDecl}\\
\mathit{sigDecl} &::= [\texttt{abstract}]\ \texttt{sig}\ \mathit{sigId}\ [\texttt{extends}\ \mathit{sigId}]\ \mathit{sigBody}\\
\mathit{sigBody} &::= \{\ [\mathit{fieldDecl}\ (\texttt{,}\ \mathit{fieldDecl})^{*}]\ \}\\
\mathit{fieldDecl} &::= \mathit{setDecl} \mid \mathit{relDecl}\\
\mathit{setDecl} &::= \mathit{relId}\ \texttt{:}\ \texttt{set Time}\\
\mathit{relDecl} &::= \mathit{relId}\ \texttt{:}\ \mathit{sigId}\ [\mathit{mult}]\ \texttt{->}\ \texttt{Time} \mid \mathit{relId}\ \texttt{:}\ (\mathit{sigId}\ \texttt{->})^{+}[\mathit{mult}]\ \mathit{sigId}\ \texttt{->}\ \texttt{Time}\\
\mathit{mult} &::= \texttt{lone} \mid \texttt{one} \mid \texttt{some} \mid \texttt{set}\\
\mathit{factDecl} &::= \texttt{fact}\ \{\texttt{all t:Time |}\ \texttt{all}\ \mathit{varId}\texttt{:}\mathit{sigId}\ \texttt{|}\ \mathit{form}\}\\
\mathit{funDecl} &::= \texttt{fun}\ \mathit{funId}\texttt{[(}\mathit{varId}\texttt{:}\mathit{sigId}\texttt{,)}^{+}\texttt{t:Time] :}\ \texttt{set}\ \mathit{sigId}\ \{\ \mathit{expr}\ \}\\
\mathit{predDecl} &::= \texttt{pred}\ \mathit{predId}\texttt{[(}\mathit{varId}\texttt{:}\mathit{sigId}\texttt{,)}^{+}\texttt{t,t':Time]}\ \{\ \mathit{form}^{+}\ \}\\
\mathit{form} &::= \mathit{expr}\ \mathit{compOp}\ \mathit{expr} \mid \mathit{form}\ \mathit{logicOp}\ \mathit{form} \mid \mathit{form}\ \texttt{=>}\ \mathit{form}\ [\texttt{,}\ \mathit{form}] \mid\\
&\quad \texttt{!}\mathit{form} \mid \mathit{intExpr}\ \mathit{intCompOp}\ \mathit{intExpr} \mid \mathit{quant}\ \mathit{varId}\texttt{:}\mathit{sigId}\ \texttt{|}\ \mathit{form}\\
\mathit{logicOp} &::= \texttt{\&\&} \mid \texttt{||} \mid \texttt{<=>}\\
\mathit{compOp} &::= \texttt{=} \mid \texttt{in}\\
\mathit{intCompOp} &::= \texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=<} \mid \texttt{>=}\\
\mathit{quant} &::= \texttt{all} \mid \texttt{no} \mid \texttt{lone} \mid \texttt{one} \mid \texttt{some}\\
\mathit{expr} &::= \mathit{varId} \mid \mathit{sigId} \mid \mathit{relId}\texttt{.t} \mid \mathit{relId}\texttt{.t'} \mid \texttt{none} \mid \texttt{univ} \mid \texttt{iden} \mid \mathit{expr}\ \mathit{relOp}\ \mathit{expr} \mid\\
&\quad \texttt{~}\mathit{expr} \mid \texttt{\^{}((}\mathit{varId}\texttt{.)}^{*}\texttt{(}\mathit{relId}\texttt{.t))} \mid \texttt{\^{}((}\mathit{varId}\texttt{.)}^{*}\texttt{(}\mathit{relId}\texttt{.t'))} \mid\\
&\quad \texttt{*((}\mathit{varId}\texttt{.)}^{*}\texttt{(}\mathit{relId}\texttt{.t))} \mid \texttt{*((}\mathit{varId}\texttt{.)}^{*}\texttt{(}\mathit{relId}\texttt{.t'))} \mid \mathit{funId}\texttt{[(}\mathit{varId}\texttt{,)}^{+}\texttt{t]} \mid\\
&\quad \mathit{funId}\texttt{[(}\mathit{varId}\texttt{,)}^{+}\texttt{t']} \mid \{\mathit{varId}\texttt{:}\mathit{sigId}\texttt{(,}\mathit{varId}\texttt{:}\mathit{sigId}\texttt{)}^{+}\ \texttt{|}\ \mathit{form}\}\\
\mathit{relOp} &::= \texttt{.} \mid \texttt{+} \mid \texttt{-} \mid \texttt{\&} \mid \texttt{<:} \mid \texttt{:>} \mid \texttt{->} \mid \texttt{++}\\
\mathit{intExpr} &::= \mathit{integer} \mid \texttt{\#}\mathit{expr} \mid \texttt{int[}\mathit{expr}\texttt{]} \mid \mathit{intExpr}\ \mathit{intOp}\ \mathit{intExpr}\\
\mathit{intOp} &::= \texttt{+} \mid \texttt{-}
\end{aligned}
$$

Fig. 2. Subset of Alloy translated to UML+OCL

prescribed in the Alloy type system [6]. Since fields will be represented by attributes or associations in UML, without this restriction we might not be able to determine the type of attributes or association ends.

- Multiplicity constraints can only occur in the last column (not counting `Time`) of a field declaration. A field relating more than two signatures will be represented by a qualified association in UML, and those only support multiplicities in the association end.
- OCL requires a *context* (a class) for all invariants and methods. As such, Alloy facts must be further restricted to include at least one additional universally quantified variable besides the special time variable. The type of this variable will determine the OCL context. Moreover, functions and predicates are required to have at least one parameter besides the special time parameters. The type of the first parameter will determine the context of the target method.

- OCL does not natively support transitive closure. So far, we only managed to translate the transitive closure of a concrete relation, instead of an arbitrary relational expression. The syntax is restricted accordingly.
- Predicate call is not supported, since OCL constraints can only invoke side-effect free queries.
- Assertions and commands will not be considered, since they do not have a counterpart in OCL. In general they only make sense for model V&V, for which OCL is currently not well suited. We prescribe that model V&V should be performed using the Alloy Analyzer, so those constructs can safely be ignored when translating to OCL.

The grammar of Figure 2 also includes some additional minor restrictions, that do not limit the expressivity of the language. For example, we do not allow signature facts, neither atomic formulas of the form *mult expr*. These restrictions simplify the presentation of the translation, and can easily be lifted by means of trivial refactorings. For instance, the formula `lone a.(b.addr.t)` in Figure 1 can be refactored to `#(a.(b.addr.t))=<1`.

4.2 From Alloy to UML Class Diagrams

The relationship between UML class diagrams and Alloy declarations is straightforward, as noticed in [19,13,1]: in general, classes correspond to signatures (preserving the inheritance relation), associations and attributes to fields, and methods to predicates and functions. These relationships are essentially the same when translating from Alloy to UML class diagrams, but with the novelty that some fields may lead to non-binary associations.

As seen in Figure 2, excluding the mandatory `Time` column, fields can be of two kinds:

- Sets with type A, to be translated as attributes of class A with type `Boolean`.
- Relations with type A_1 `->` $\ldots$ `->` m A_n, to be translated as qualified associations between A_1 and A_n, with $A_2, \ldots, A_{n-1}$ as qualifiers. The multiplicity at the end of the association depends on m: `0..*` for `set`; `1..*` for `some`; `0..1` for `lone`; and `1..1` for `one`. If m is absent, the default is `set`.

If the relation is binary, with type A_1 `->` m A_2, and m is either `lone` or `one` it is more natural to encode it as attribute of A_1 with type A_2.

The UML class diagram corresponding to the address book example of Figure 1 is presented in Figure 3.

4.3 From Alloy to OCL

The model transformation from the Alloy subset described in Section 4.1 to OCL will be encoded using an embedding function $[\![\cdot]\!]$. To simplify the presentation, this function will accept and produce concrete syntax. The following convention will be followed for naming variables denoting the various grammar elements:

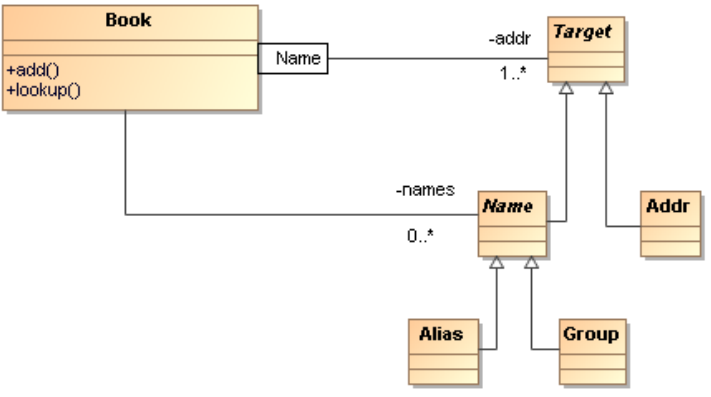


Fig. 3. UML class diagram corresponding to address book example

x, y, z for variable identifiers; A, B, C for signature identifiers and types in general; R, S, T for relation identifiers; ϕ, ψ, φ for formulas; Φ, Ψ, Υ for relational expressions; and α, β, γ for integer expressions.

We will assume the input model to be well-typed, according to the typing system described in [6]. This type system is very relaxed: an error occurs when a expression can be shown to always be empty at static time. For example, the composition $\Phi.\Psi$ is well-defined for any relational expressions Φ: A -> B and Ψ: C -> D, if the intersection of types B and C is non-empty. The type of a relational expression is itself a relation: a set of tuples of atomic signatures (i.e. signatures that are not further extended, such as `Book`, `Addr`, `Alias`, and `Group` in our running example). The type inference rules ensure that all the tuples in the type relation have the same length. Given a relational expression Φ of arity $|\Phi|$, we will denote the type of the n-th column as Φ^n (assuming $0 < n \leqslant |\Phi|$). The type of a column is guaranteed to be a set of atomic signatures, each corresponding to a concrete class in UML. In the translation we sometimes need to quantify over such types. To simplify the presentation, notation

$$\{A_1, \ldots, A_n\}\texttt{.allInstances->forAll(}x\texttt{|}\phi\texttt{)}$$
$$\{A_1, \ldots, A_n\}\texttt{.allInstances->exists(}x\texttt{|}\phi\texttt{)}$$

where $\{A_1, \ldots, A_n\}$ is an Alloy type, will be used as a shorthand for, respectively

$$A_1\texttt{.allInstances->forAll(}x\texttt{|}\phi\texttt{) and ... and }A_n\texttt{.allInstances->forAll(}x\texttt{|}\phi\texttt{)}$$
$$A_1\texttt{.allInstances->exists(}x\texttt{|}\phi\texttt{) or ... or }A_n\texttt{.allInstances->exists(}x\texttt{|}\phi\texttt{)}$$

The translation of an Alloy module is triggered by the following rule:

$$[\![\texttt{module } id \texttt{ sig Time } \{\} \ s_1 \ldots s_n \ p_1 \ldots p_m]\!] \equiv$$
$$\texttt{package } m \ [\![p_1]\!] \ldots [\![p_m]\!] \texttt{ endpackage}$$

Fact, Function and Predicate Declarations. Figure 4 details the transformations of fact, function and predicate declarations. Signature declarations are

$[\![$`fact {all t:Time | all` x`:`A `|` ϕ`}`$]\!] \equiv$

`context` A `inv:` $[\![\phi]\!]_x$

$[\![$`fun` f`[`x_1`:`A_1`,...,`x_n`:`A_n`,t:Time] : set` B `{` Φ `}`$]\!] \equiv$

`context` A_1`::`f`(`x_2`:`A_2`,...,`x_n`:`A_n`):Set(`B`)`

`post:` B`.allInstances->select(`y`|`$[\![\langle y \rangle \in \Phi]\!]_{x_1}$`)`

$[\![$`pred` f`[`x_1`:`A_1`,...,`x_n`:`A_n`,t,t':Time] {` $\phi_1 \ldots \phi_m$ `}`$]\!] \equiv$

`context` A_1`::`f`(`x_2`:`A_2`,...,`x_n`:`A_n`)`

`pre:` $[\![\phi_1]\!]_{x_1}$ if `t'` does not occur in ϕ_1
`post:` $[\![\phi_1]\!]'_{x_1}$ otherwise

...

`pre:` $[\![\phi_m]\!]_{x_1}$ if `t'` does not occur in ϕ_m
`post:` $[\![\phi_m]\!]'_{x_1}$ otherwise

Fig. 4. Translation of fact, function and predicate declarations

ignored in the OCL generation, and are only used in the UML class diagram generation detailed in the previous section.

In OCL, all invariants and method specifications must be defined in the context of a class. For Alloy facts, the type of the first universally quantified variable (appart from the mandatory `Time` one) will determine the context of the generated invariant. The translation of formulas must then be parametrized with the name of the variable that will denote the `self` object. For functions and predicates, the context is determined by the type of the first parameter. In a predicate, all formulas where `t'` does not occur will be translated as pre-conditions. Otherwise, they are translated as post-conditions.

Two slightly different formula translations will be defined, due to different meanings that variable `t` assumes in different contexts. In a post-condition, an association R`.t` should be translated as R`@pre`, since `t` denotes the pre-state. In invariants, functions and pre-states it denotes the only visible state, and thus the translation just outputs R. As such, we will use $[\![\phi]\!]$ to translate a formula ϕ that occurs in an invariant, function, or pre-condition; and $[\![\phi]\!]'$ to translate a formula ϕ that occurs in a post-condition.

Formulas. The translation of formulas is presented in Figure 5. We omit the definition of $[\![\cdot]\!]'$ because for formulas it is identical - it will only diverge when applied to relational expressions. Most logic operators have a direct counterpart in OCL and can thus be trivially translated. OCL does not support the non-standard quantifiers `no` and `one`, but they can be simulated by testing the cardinality of the subset of the type satisfying the quantified formula.

The trickiest part of the translation concerns the atomic formulas Φ `in` Ψ, where Φ and Ψ are arbitrary relational expressions. This formula cannot be encoded using set inclusion because $|\Phi|$ can be greater than 1, and, unlike Alloy, OCL does not support the construction of arbitrary relations as normal

$$
\begin{aligned}
[\![\Phi \texttt{ in } \Psi]\!]_x &\equiv \Phi^1\texttt{.allInstances->forAll(}y_1\texttt{|}\ldots \\
&\qquad \Phi^{|\Phi|}\texttt{.allInstances->forAll(}y_{|\Phi|}\texttt{|} \\
&\qquad\quad [\![\langle y_1,\ldots,y_{|\Phi|}\rangle \in \Phi]\!]_x \texttt{ implies } [\![\langle y_1,\ldots,y_{|\Phi|}\rangle \in \Psi]\!]_x\texttt{)}\ldots\texttt{)} \\
[\![\Phi \texttt{ = } \Psi]\!]_x &\equiv [\![\Phi \texttt{ in } \Psi]\!]_x \texttt{ and } [\![\Psi \texttt{ in } \Phi]\!]_x \\
[\![\phi \texttt{ \&\& } \psi]\!]_x &\equiv [\![\phi]\!]_x \texttt{ and } [\![\psi]\!]_x \\
[\![\phi \texttt{ || } \psi]\!]_x &\equiv [\![\phi]\!]_x \texttt{ or } [\![\psi]\!]_x \\
[\![\phi \texttt{ <=> } \psi]\!]_x &\equiv [\![\phi \texttt{ => } \psi]\!]_x \texttt{ and } [\![\psi \texttt{ => } \phi]\!]_x \\
[\![\phi \texttt{ => } \psi]\!]_x &\equiv [\![\phi]\!]_x \texttt{ implies } [\![\psi]\!]_x \\
[\![\phi \texttt{ => } \psi\texttt{,}\varphi]\!]_x &\equiv \texttt{if } [\![\phi]\!]_x \texttt{ then } [\![\psi]\!]_x \texttt{ else } [\![\varphi]\!]_x \texttt{ endif} \\
[\![\texttt{!}\phi]\!]_x &\equiv \texttt{not } [\![\phi]\!]_x \\
[\![\alpha \texttt{ = } \beta]\!]_x &\equiv [\![\alpha]\!]_x \texttt{ = } [\![\beta]\!]_x \\
&\quad\ldots \\
[\![\alpha \texttt{ >= } \beta]\!]_x &\equiv [\![\alpha]\!]_x \texttt{ >= } [\![\beta]\!]_x \\
[\![\texttt{all } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->forAll(}y\texttt{|}[\![\phi]\!]_x\texttt{)} \\
[\![\texttt{some } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->exists(}y\texttt{|}[\![\phi]\!]_x\texttt{)} \\
[\![\texttt{no } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->select(}y\texttt{|}[\![\phi]\!]_x\texttt{)->isEmpty()} \\
[\![\texttt{one } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->select(}y\texttt{|}[\![\phi]\!]_x\texttt{)->size() = 1}
\end{aligned}
$$

Fig. 5. Translation of formulas

first-order values. As such, relational expressions will be translated by building their standard first-order denotational semantics: a relational expression Φ will be translated by a formula that checks if a tuple $\langle y_1,\ldots,y_{|\Phi|}\rangle$ belongs to the denoted relation. The inclusion Φ `in` Ψ can thus be translated by a formula that checks if all tuples of the appropriate type that belong to Φ also belong to Ψ. Note that the type system ensures that the arity of Φ and Ψ are the same. Using the first-order semantics of relational logic to embed Alloy in other logical frameworks is kind of folklore. For example, a similar strategy was used recently to develop a tool for unbounded verification of Alloy models using SMT solvers [8].

Relational Expressions. The translation of relational expressions is presented in Figure 6. As explained above, this translation basically encodes the standard first-order semantics of relational operators. A brief explanation of the most interesting rules follows:

- Testing if a unary tuple is a member of a variable can be done with a simple equality test. Note that, as mentioned before, Alloy variables are singleton unary relations. If the variable denotes the `self` object then this identifier is used instead.
- Translation of field R`.t` membership depends on the arity of R: if it is a set we just check the value of the generated boolean attribute; otherwise we navigate the qualified association R.

- Translation of field membership requires an additional type checking since Alloy allows access to a field from any signature that includes the owner of the field. A reference like this could generate an undefined value in OCL. As such, we translate to `false` when the type of each variable y_i is not a sub-type of R^i. Translation of variables and signature membership assumes similar considerations. To simplify the presentation, these type-checkings are not included in Figure 6.
- The semantics of the relational composition $\Phi.\Psi$ leads to a new existential quantifier over the mediating type. We quantify over $\Phi^{|\Phi|} \cap \Psi^1$ because the composition only succeeds for values belonging to the intersection of both types. This optimization reduces the number of quantifiers in the outputted formula.
- Testing if $\langle y_1, \ldots, y_n \rangle$ is included in the relational overriding Φ `++` Ψ is reduced to a membership test over Ψ if $\langle y_1, \ldots, y_{n-1} \rangle$ belongs to its domain; otherwise a membership test over Φ is generated.
- In a relation defined by the comprehension $\{z_1 : A_1, \ldots, z_n : A_n \mid \phi\}$, the membership test is translated by just applying the predicate ϕ to the tuple variables $y_1, \ldots, y_n$ instead of $z_1, \ldots, z_n$.

The translation of closures is not straightforward since they are not finitely axiomatizable in first order logic, and OCL also does not support them natively. Fortunately, there are many ways to encode the transitive closure using recursive definitions. Given an arbitrary relation R`:` A_1 `->` ... `->` A_n, the transitive closure of the respective (partially applied) qualified association can be implemented as follows:

```
context A1
  def: RClosureAux(y2:A2,...,yn-1:An-1,s:Set(An)):Set(An) =
    let s':Set(An) = s->collect(x:A1 | yn-1.R[yn-2,...,y2,x] in
      if s->includesAll(s') then s
      else RClosureAux(y2,...,yn-1,s->union(s'))
      endif
  def: RClosure(y2:A2,...,yn-1:An-1):Set(An) =
    RClosureAux(y2,...,yn-1,yn-1.R[yn-2,...,y2,self]))
```

The translation of relational expressions occurring in post-conditions is almost identical, with the exception of the rules presented in Figure 7, where relation identifiers within scope `t` are evaluated in the pre-state. `RClosurePre` is an auxiliar definition similar to the one presented above, but with all occurrences of R replaced by R`@pre`.

The blind application of these translation rules usually results in obfuscated OCL specifications, mainly due to the introduction of quantifiers in the translation of the relational inclusion and composition. Fortunately, some first-order equivalences can be applied to the resulting order to simplify it, namely:

$$
\begin{aligned}
[\![\langle y\rangle \in z]\!]_x &\equiv \begin{array}{ll} y\texttt{=self} & \text{if } z = x \\ y\texttt{=}z & \text{otherwise}\end{array}\\
[\![\langle y\rangle \in A]\!]_x &\equiv A\texttt{.allInstances->includes(}y\texttt{)}\\
[\![\langle y_1,\ldots,y_n\rangle \in R\texttt{.t}]\!]_x &\equiv \begin{array}{ll} y_1\texttt{.}R\texttt{[}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n > 1 \\ y_1\texttt{.}R\texttt{()} & \text{otherwise}\end{array}\\
[\![\langle y\rangle \in \texttt{none}]\!]_x &\equiv \texttt{false}\\
[\![\langle y\rangle \in \texttt{univ}]\!]_x &\equiv \texttt{true}\\
[\![\langle y_1,y_2\rangle \in \texttt{iden}]\!]_x &\equiv y_1\texttt{=}y_2\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\texttt{.}\Psi]\!]_x &\equiv (\Phi^{|\Phi|}\cap\Psi^1)\texttt{.allInstances->exists(}y\texttt{|}\\
&\qquad [\![\langle y_1,\ldots,y_{|\Phi|-1},y\rangle \in \Phi]\!]_x\ \texttt{and}\ [\![\langle y,y_{|\Phi|},\ldots,y_n\rangle \in \Psi]\!]_x\texttt{)}\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{+}\ \Psi]\!]_x &\equiv [\![\langle y_1,\ldots,y_n\rangle \in \Phi]\!]_x\ \texttt{or}\ [\![\langle y_1,\ldots,y_n\rangle \in \Psi]\!]_x\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{-}\ \Psi]\!]_x &\equiv [\![\langle y_1,\ldots,y_n\rangle \in \Phi]\!]_x\ \texttt{and (not}\ [\![\langle y_1,\ldots,y_n\rangle \in \Psi]\!]_x\texttt{)}\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{\&}\ \Psi]\!]_x &\equiv [\![\langle y_1,\ldots,y_n\rangle \in \Phi]\!]_x\ \texttt{and}\ [\![\langle y_1,\ldots,y_n\rangle \in \Psi]\!]_x\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{<:}\ \Psi]\!]_x &\equiv [\![\langle y_1\rangle \in \Phi]\!]_x\ \texttt{and}\ [\![\langle y_1,\ldots,y_n\rangle \in \Psi]\!]_x\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{:>}\ \Psi]\!]_x &\equiv [\![\langle y_1,\ldots,y_n\rangle \in \Phi]\!]_x\ \texttt{and}\ [\![\langle y_n\rangle \in \Psi]\!]_x\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{->}\ \Psi]\!]_x &\equiv [\![\langle y_1,\ldots,y_{|\Phi|}\rangle \in \Phi]\!]_x\ \texttt{and}\ [\![\langle y_{|\Phi|+1},\ldots,y_n\rangle \in \Psi]\!]_x\\
[\![\langle y_1,\ldots,y_n\rangle \in \Phi\ \texttt{++}\ \Psi]\!]_x &\equiv \texttt{if}\ [\![\langle y_1,\ldots,y_{n-1}\rangle \in \Psi\texttt{.}\Psi^n]\!]_x\ \texttt{then}\ [\![\langle y_1,\ldots,y_n\rangle \in \Psi]\!]_x\\
&\qquad \texttt{else}\ [\![\langle y_1,\ldots,y_n\rangle \in \Phi]\!]_x\ \texttt{endif}\\
[\![\langle y_1,\ldots,y_n\rangle \in \texttt{\~{}}\Phi]\!]_x &\equiv [\![\langle y_n,\ldots,y_1\rangle \in \Phi]\!]_x\\
[\![\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}]\!]_x &\equiv\ y_1\texttt{.}R\texttt{Closure(}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{)->includes(}y_n\texttt{)}\\
[\![\langle y_1,y_n\rangle \in \texttt{*(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}]\!]_x &\equiv\ y_1\texttt{=}y_n\ \texttt{or}\ [\![\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}]\!]_x\\
[\![\langle y\rangle \in f\texttt{[}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{,t]}]\!]_x &\equiv y_1\texttt{.}f\texttt{(}y_2\texttt{,}\ldots\texttt{,}y_n\texttt{)->includes(}y\texttt{)}\\
[\![\langle y_1,\ldots,y_n\rangle \in \{z_1\texttt{:}A_1\texttt{,}\ldots\texttt{,}z_n\texttt{:}A_n\ \texttt{|}\ \phi\}]\!]_x &\equiv [\![\phi[y_1/z_1,\ldots,y_n/z_n]]\!]_x
\end{aligned}
$$

Fig. 6. Translation of relational expressions

$$
\begin{aligned}
A\texttt{.allInstances->exists(}y\ \texttt{|}\ y\texttt{=}z\ \texttt{and}\ \phi\texttt{)} &\equiv \phi[z/y]\\
A\texttt{.allInstances->forAll(}y\ \texttt{|}\ y\texttt{=}z\ \texttt{implies}\ \phi\texttt{)} &\equiv \phi[z/y]
\end{aligned}
$$

Integer expressions. Figure 8 presents the translation of Alloy integer expressions to OCL. Alloy expression $\texttt{\#}\Phi$ computes the size of a relational expression Φ of arbitrary arity. The presented rule only works for unary expressions. Using tuples it is trivial to generalize it to arbitrary relational expressions.

The Address Book Case Study. An excerpt of the OCL model obtained from the address book example is presented in Figure 9. It includes the first two invariants and the specification of operation `add`. Both simplification rules where applied to the result, which was then manually rendered for better comprehension.

$$[\![\langle y_1,\ldots,y_n\rangle \in R.\texttt{t}]\!]'_x \equiv \begin{array}{ll} y_1\,.\,R\texttt{@pre[}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n>1 \\ y_1\,.\,R\texttt{@pre()} & \text{otherwise}\end{array}$$

$$[\![\langle y_1,\ldots,y_n\rangle \in R.\texttt{t'}]\!]'_x \equiv \begin{array}{ll} y_1\,.\,R\texttt{[}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n>1 \\ y_1\,.\,R\texttt{()} & \text{otherwise}\end{array}$$

$$[\![\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.....}y_{n-1}\texttt{.}R\texttt{.t)}]\!]'_x \equiv\ y_1\,.\,R\texttt{ClosurePre(}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{)->includes(}y_n\texttt{)}$$

$$[\![\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.....}y_{n-1}\texttt{.}R\texttt{.t')}]\!]'_x \equiv\ y_1\,.\,R\texttt{Closure(}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{)->includes(}y_n\texttt{)}$$

$$[\![\langle y_1,y_n\rangle \in \texttt{*(}y_2\texttt{.....}y_{n-1}\texttt{.}R\texttt{.t)}]\!]'_x \equiv\ y_1\texttt{=}y_n\ \texttt{or}\ [\![\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.....}y_{n-1}\texttt{.}R\texttt{.t)}]\!]'_x$$

$$[\![\langle y_1,y_n\rangle \in \texttt{*(}y_2\texttt{.....}y_{n-1}\texttt{.}R\texttt{.t')}]\!]'_x \equiv\ y_1\texttt{=}y_n\ \texttt{or}\ [\![\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.....}y_{n-1}\texttt{.}R\texttt{.t')}]\!]'_x$$

$$[\![\langle y\rangle \in f\texttt{[}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{,t']}]\!]'_x \equiv\ y_1\,.\,f\texttt{(}y_2\texttt{,}\ldots\texttt{,}y_n\texttt{)->includes(}y\texttt{)}$$

Fig. 7. Translation of relational expressions in post-conditions

$$\begin{aligned}
[\![n]\!]_x &\equiv n \\
[\![\texttt{\#}\Phi]\!]_x &\equiv \Phi^1\texttt{.allInstances->select(}y\texttt{|}[\![\langle y\rangle \in \Phi]\!]_x\texttt{)->size()} \\
[\![\texttt{int[}\Phi\texttt{]}]\!]_x &\equiv \Phi^1\texttt{.allInstances->select(}y\texttt{|}[\![\langle y\rangle \in \Phi]\!]_x\texttt{)->sum()} \\
[\![\alpha\ \texttt{+}\ \beta]\!]_x &\equiv [\![\alpha]\!]_x\ \texttt{+}\ [\![\beta]\!]_x \\
[\![\alpha\ \texttt{-}\ \beta]\!]_x &\equiv [\![\alpha]\!]_x\ \texttt{-}\ [\![\beta]\!]_x
\end{aligned}$$

Fig. 8. Translation of integer expressions

```
context Book inv:
  Name.allInstances->forAll(v0 |
    Target.allInstances->exists(v1 | self.addr[v0]->includes(v1)
         and Target.allInstances ->includes(v1))  implies
    self.names->includes(v0))

context Book inv:
  Name.allInstances->select(n |
    n.addrClosure(self)->includes(n))->isEmpty()

context Book::add(n:Name,a:Target)
  pre: self.names->includes(n)
  post: Name.allInstances->forAll(v0 | Target.allInstances->forAll(v1 |
          self.addr[v0]->includes(v1)  implies
          self.addr@pre[v0]->includes(v1) or (v0=n and v1=a)))
```

Fig. 9. OCL specifications of the address book example

5 Concluding Remarks and Future Work

We have presented a model transformation from Alloy to UML class diagrams annotated with OCL. We have formally characterized the Alloy local state idiom accepted by the transformation. This idiom is sufficiently broad to encompass

most specifications. When compared to the previously developed transformations from UML+OCL to Alloy [1,13], this model transformation raised interesting new challenges, namely: the translation of relational expressions of arbitrary arity; dealing with the idiosyncrasies of Alloy's type-system; and the encoding of closures.

The transformation still has some limitations, most notably not allowing signature inclusion in the source Alloy model. Signature inclusion is mostly used in Alloy to overcome the single-inheritance limitation. We intend to extend our idiom to include such usages, and then translate it directly using multiple-inheritance. We also intend to extend it to other Alloy idioms that allow the specification of state-transition systems, in order to also generate UML state-chart diagrams.

We have implemented the proposed transformation in Haskell, generating syntax compatible with popular UML+OCL modeling applications. The tool is available for download at `http://sourceforge.net/projects/alloy2ocl`. It includes a couple of additional case-studies, but we intend to extend this set to further validate the transformation.

Acknowledgments. This research was partially supported by QREN (the Portuguese National Strategy Reference Chart) project 1621 – Evolve.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1), 69–86 (2008)
2. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. Innovations in Systems and Software Engineering 6(1-2), 55–63 (2010)
3. Brucker, A.D., Wolff, B.: HOL-OCL: a formal proof environment for UML/OCL. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 97–100. Springer, Heidelberg (2008)
4. Demuth, B., Hussmann, H., Loecher, S.: OCL as a specification language for business rules in database applications. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 104–117. Springer, Heidelberg (2001)
5. DresdenOCL website, `http://www.dresden-ocl.org/index.php/DresdenOCL`
6. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 189–199. ACM, New York (2004)
7. Georg, G., Anastasakis, K., Bordbar, B., Houmb, S.H., Toahchoodee, I.R.M.: Verification and trade-off analysis of security properties in UML system models. IEEE Transactions on Software Engineering 36(3), 338–356 (2010)
8. El Ghazi, A.A., Taghdiri, M.: Relational reasoning via SMT solving. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 133–148. Springer, Heidelberg (2011)
9. Gheyi, R., Massoni, T., Borba, P.: Formally introducing Alloy idioms. In: Proceedings of the Brazilian Symposium on Formal Methods, pp. 22–37 (2007)

10. Giannakopoulos, T., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Towards an operational semantics for alloy. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 483–498. Springer, Heidelberg (2009)
11. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and Systems Modeling 4(4), 386–398 (2005)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
13. Massoni, T., Gheyi, R., Borba, P.: Formal refactoring for UML Class Diagrams. In: Proceedings of the 19th Brazilian Symposium on Software Engineering, pp. 152–167 (2005)
14. Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling, pp. 41–48. ACM, New York (2007)
15. OMG: MDA Guide version 1.0.1 (2003)
16. OMG: Object Constraint Language, Version 2.2 (2010)
17. OMG: UML Superstructure, Version 2.3 (2010)
18. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation. ACM, New York (2009)
19. Vaziri, M., Jackson, D.: Some shortcomings of OCL, the Object Constraint Language of UML. In: Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems, pp. 555–562. IEEE, Los Alamitos (2000)

Safe Distribution of Declarative Processes

Thomas Hildebrandt[1], Raghava Rao Mukkamala[1], and Tijs Slaats[1,2] *

[1] IT University of Copenhagen,
Rued Langgaardsvej 7, 2300 Copenhagen, Denmark
{hilde,rao,tslaats}@itu.dk
http://www.itu.dk
[2] Exformatics A/S, 2100 Copenhagen, Denmark

Abstract. We give a general technique for safe distribution of a declarative (global) process as a network of (local) synchronously communicating declarative processes. Both the global and local processes are given as Dynamic Condition Response (DCR) Graphs. DCR Graphs is a recently introduced declarative process model generalizing labelled prime event structures to a systems model able to finitely represent ω-regular languages. An operational semantics given as a transition semantics between markings of the graph allows DCR Graphs to be conveniently used as both specification and execution model. The technique for distribution is based on a new general notion of projection of DCR Graphs relative to a subset of labels and events identifying the set of external events that must be communicated from the other processes in the network in order for the distribution to be safe. We prove that for any vector of projections that covers a DCR Graph that the network of synchronously communicating DCR Graphs given by the projections is bisimilar to the original global process graph. We exemplify the distribution technique on a process identified in a case study of an cross-organizational case management system carried out jointly with Exformatics A/S.

Keywords: formal specification, distributed synthesis, cross-organizational workflow, declarative processes, process composition.

1 Introduction

A model-driven software engineering approach to distributed information systems typically include both *global* models describing the collective behavior of the system being developed and *local* models describing the behavior of the individual peers or components.

The global and local descriptions should be consistent. If the modeling languages have formal semantics and the local model language support composition of individual processes, the consistency can be formally established, which we will refer to as the

* Authors listed alphabetically. This research is supported by the Danish Research Agency through a Knowledge Voucher granted to Exformatics (grant #10-087067, www.exformatics.com), the Trustworthy Pervasive Healthcare Services project (grant #2106-07-0019, www.trustcare.eu) and the Computer Supported Mobile Adaptive Business Processes project (grant #274-06-0415, www.cosmobiz.dk).

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 237–252, 2011.

consistency problem: Given a global model and a set of local models, is the behavior of the composition of the local models consistent with the global model? In order to support "top-down" model-driven engineering starting from the global model, one should address the more challenging *distributed synthesis problem*: Given a global model and some formal description of how the model should be distributed, can we synthesize a set of local processes with respect to this distribution which are consistent to the the global model?

In past work, briefly surveyed below, the result of the distributed synthesis have been a network of local processes described in an imperative process model, e.g. as a network of typed pi-calculus processes or a product automaton. The global process description has either been given declaratively, e.g. in some temporal logic, or imperatively, e.g. as a choreography or more generally a transition system.

In the present paper we address the distributed synthesis problem in a setting where both the global and the local processes are described *declaratively* as Dynamic Condition Response Graphs (DCR Graphs). DCR Graphs is a declarative workflow model introduced previously in [14, 15] as a generalization of the classical event structure model [47] allowing finite specification of infinite or iterative behavior (by allowing events to be executed more than once and replacing the symmetric conflict relation by asymmetric exclusion and (re-)inclusion relations) and specification of progress conditions (by replacing the causal order relation of event structures with two relations, respectively defining the conditions for and required responses to the execution of an event).

The motivation for introducing the DCR Graph model is to give, as part of the Trustworthy Pervasive Healthcare Services [13] project, a declarative model that can be used both as specification language and execution language for flexible workflow and business process. Indeed, the DCR Graphs model is inspired by and formalizes the core primitives of the process model employed by the industrial partner (Resultmaker) in the TrustCare project and is now being implemented in the workflow engine developed at Exformatics. As identified in e.g. [6, 43] declarative process languages make it easier to specify loosely constrained systems. Also, we believe the declarative approach is more promising when it comes to composition, and (dynamic) changes of processes which is one of the main objectives of the TrustCare project.

To safely distribute a DCR Graph we first define (Def. 3, Sec. 3.1) a new general notion of *projection* of DCR Graphs relative to a subset of labels and events. The key point is to identify the set of events that must be communicated from other processes in the network in order for the state of the local process to stay consistent with the global specification (Prop. 1, Sec. 3). To also enable the reverse operation, building global graphs from local graphs, we then define the composition of two DCR Graphs, essentially by gluing joint events. As a sanity check we prove (Prop. 2, Sec. 3.2) that if we have a collection of projections of a DCR Graph that cover the original graph (Def. 7, Sec. 3.2) then the composition yields back the same graph. We then finally proceed to the main technical result, defining networks of synchronously communicating DCR Graphs and stating (in Thm. 1, Sec. 3.3) the correspondence between a global process and a network of communicating DCR Graphs obtained from a covering projection (relying on Prop. 1). Throughout the paper we exemplify the distribution technique on a simple cross-organizational process identified within a case study carried out jointly

with Exformatics A/S using DCR Graphs for model-driven design and engineering of an inter-organizational case management system. We conclude in Sec. 4 and provide pointers to future work.

1.1 Related Work

There are many researchers [1, 20, 21, 40–42, 46] who have explicitly focussed on the problem of verifying the correctness of inter-organizational workflows in the domain of petri nets. In [41], message sequence charts are used to model the interaction between the participant workflows that are modeled using petri nets and the overall workflow is checked for consistency against an interaction structure specified in message sequence charts. In [20] Kindler et. al. followed a similar but more formal and concrete approach, where the interaction of different workflows is specified using a set of scenarios given as sequence diagrams and using criteria of local soundness and composition theorem, guaranteed the global soundness of an inter-organizational workflow. The authors in [40] proposed *Query Nets* based on predicate/transition petri nets to guarantee global termination, without the need for having the global specification. The work on workflow nets [1, 46] use a P2P (Public-To-Private) approach to partition a shared public view of an inter-organizational workflow over its participating entities and projection inheritance is used to generate a private view that is a subclass to the relevant public view, to guarantee the deadlock and livelock freedom. Further a more liberal and a weaker notion than projection inheritance, *accordance* has been used in [42] to guarantee the weak termination in the multiparty contracts based on open nets.

Modeling global behavior as a set of conversations among participating services has been studied by many researchers [2, 3, 11, 35, 48, 49] in the area business processes. An approach based on guarded automata studied in [11], for the realizability analysis of conversation protocols, whereas the authors in [49] used colored petri nets to capture the complex conversations. A framework for calculating and controlled propagation of changes to the process choreographies based on the modifications to partner's private processes has been studied in [35]. Similarly, but using process calculus to model service contracts, Bravetti-Zavattaro proposed conformance notion for service composition in [2] and further enhanced their correctness criteria in [3] by the notion of strong service compliance.

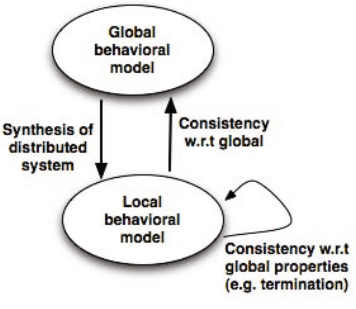


Fig. 1. Key problems studied in related work

Researchers [9, 19, 23, 29] in the web services community have been working on web service composition and decentralized process execution using BPEL [30] and other related technologies to model the web services. A technique to partition a composite web service using program analysis was studied in [29] and on the similar approach, [19] explored decomposition of a business process modeled in BPEL, primarily focussing on P2P interactions . Using a formal approach based on I/O automata

representing the services, the authors in [23] have studied the problem of synthesizing a decentralized choreography strategy, that will have optimal overhead of service composition in terms of costs associated with each interaction.

The derivation of descriptions of local components from a global model has been researched for the imperative choreography language WS-CDL in the work on structured communication-centred programming for web services by Carbone, Honda and Yoshida [4]. To put it briefly, the work formalizes the core of WS-CDL as the global process calculus and defines a formal theory of end-point projections projecting the global process calculus to abstract descriptions of the behavior of each of the local "end-points" given as pi-calculus processes typed with session types.

A methodology for deriving process descriptions from a business contract formalized in a formal contract language was studied in [22], while [36] proposes an approach to extract a distributed process model from collaborative business process. In[9, 10], the authors have proposed a technique for the flexible decentralization of a process specification with necessary synchronization between the processing entities using dependency tables.

In [5, 12, 27] foundational work has been made on synthesizing distributed transition systems from global specification for the models of synchronous product and asynchronous automata[50]. In [27] Mukund categorized structural and behavioral characterizations of the synthesis problem for synchronous and loosely cooperating communication systems based on three different notions of equivalence: state space, language and bisimulation equivalence. Further Castellani et. al. [5] characterized when an an arbitrary transition system is isomorphic to its product transition systems with a specified distribution of actions and they have shown that for finite state specifications, a finite state distributed implementation can be synthesized. Complexity results for distributed synthesis problems for the three notions of equivalences were studied in [12].

Many commercial and research workflow management systems also support distributed workflow execution and some of them even support ad-hoc changes as well. ADEPT [34], Exotica [24], ORBWork [7], Rainman [31] and Newcastle-Nortel [39] are some of the distributed workflow management systems. A good overview and discussion about distributed workflow management systems can be found in [32, 33].

So far the formalisms discussed above are more or less confined to imperative modeling languages such as Petri nets, workflow/open nets and automata based languages. To the best of our knowledge, there exists very few works [8, 25] that have studied the synthesis problem in declarative modeling languages and none where both the global and local processes are given declaratively. In [8], Fahland has studied synthesizing declarative workflows expressed in DecSerFlow [45] by translating to Petri nets. Only a predefined set of DecSerFlow constraints are used in the mapping to the Petri nets patterns, so this approach has a limitation with regards to the extensibility of the DecSerFlow language. On the other hand, in [25] Montali has studied the composition of ConDec [44] models with respect to conformance with a given choreography, based on the compatibility of the local ConDec models. But his study was limited to only composition, whereas the problem of synthesizing local models from a global model has not been studied.

2 Dynamic Condition Response Graphs

Dynamic Condition Response (DCR) Graphs has recently been introduced [15] as a declarative process model generalizing labelled event structures [47] to allow finite representations of infinite behavior (i.e. a systems model [37, 38]) and representation of progress properties.

A DCR Graph consists of a set of *labelled events*, a *marking* defining the *executed* events, *pending response* events and *included* events, and four binary relations between the events, defining the temporal constraints between events and dynamic inclusion and exclusion of events.

We employ the following notations in the rest of the paper.

Notation: For a set A we write $\mathcal{P}(A)$ for the power set of A. For a binary relation $\rightarrow \subseteq A \times A$ and a subset $\xi \subseteq A$ of A we write $\rightarrow \xi$ and $\xi \rightarrow$ for the set $\{a \in A \mid (\exists a' \in \xi \mid a \rightarrow a')\}$ and the set $\{a \in A \mid (\exists a' \in \xi \mid a' \rightarrow a)\}$ respectively. Also, we write $\rightarrow^{-1}$ for the inverse relation. Finally, for a natural number k we write $[k]$ for the set $\{1, 2, \ldots, k\}$.

We then formally define a DCR Graph as follows.

Definition 1. *A Dynamic Condition Response Graph G is a tuple* $(\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l_\mathsf{G})$*, where*

(i) E_G *is the set of* events
(ii) $\mathsf{M}_\mathsf{G} = (\mathsf{Ex}_\mathsf{G}, \mathsf{Re}_\mathsf{G}, \mathsf{In}_\mathsf{G}) \in \mathcal{P}(\mathsf{E}_\mathsf{G}) \times \mathcal{P}(\mathsf{E}_\mathsf{G}) \times \mathcal{P}(\mathsf{E}_\mathsf{G})$ *is the* marking*,*
(iii) $\rightarrow\bullet \subseteq \mathsf{E}_\mathsf{G} \times \mathsf{E}_\mathsf{G}$ *is the* condition *relation*
(iv) $\bullet\rightarrow \subseteq \mathsf{E}_\mathsf{G} \times \mathsf{E}_\mathsf{G}$ *is the* response *relation*
(v) $\pm : \mathsf{E}_\mathsf{G} \times \mathsf{E}_\mathsf{G} \rightharpoonup \{+, \%\}$ *is a partial function defining the* dynamic inclusion and exclusion *relations by* $e \rightarrow + e'$ *if* $\pm(e, e') = +$ *and* $e \rightarrow \% e'$ *if* $\pm(e, e') = \%$
(vi) L_G *is the set of* labels
(vii) $l_\mathsf{G} : \mathsf{E}_\mathsf{G} \rightarrow \mathcal{P}(\mathsf{L}_\mathsf{G})$ *is a labeling function mapping events to sets of labels.*

We write $\mathcal{M}(G)$ *for the set* $\mathcal{P}(\mathsf{E}_\mathsf{G}) \times \mathcal{P}(\mathsf{E}_\mathsf{G}) \times \mathcal{P}(\mathsf{E}_\mathsf{G})$ *of markings of G.*

The marking $\mathsf{M}_\mathsf{G} = (\mathsf{Ex}_\mathsf{G}, \mathsf{Re}_\mathsf{G}, \mathsf{In}_\mathsf{G})$ is a tuple of three sets defining respectively the previously executed events (Ex_G), the set of required responses (Re_G), and the currently included events (In_G). The set of required responses are the events that must eventually be executed (or excluded) in order to accept the execution, also referred to as the pending responses. The set of included events are the events that currently are relevant for conditions and may be executed (if their conditions are met). The condition relation $\rightarrow\bullet$ defines which (of the currently included) events must have been executed before an event can be executed. That is, for an event e to be executed, it must be included, i.e. $e \in \mathsf{In}_\mathsf{G}$ and the included conditions must be executed: $(\rightarrow\bullet\ e) \cap \mathsf{In}_\mathsf{G} \subseteq \mathsf{Ex}_\mathsf{G}$. The response relation $\bullet\rightarrow$ defines which responses are required after executing an event. That is, if the event e is executed, the events $e \bullet\rightarrow$ are added to the set of required responses in the marking. The dynamic inclusion and exclusion relations define how the set of included events changes by executing an event: If the event e is executed, the events $e \rightarrow +$ are added to the set of included events in the marking and the events $e \rightarrow \%$ are removed. Finally, an event is labelled by zero or more labels. (This is slightly more

general than previous work, where labels of events were sets of triples consisting of an action, a role and a principal.)

Fig. 2 below shows an example DCR Graph identified during the development by Exformatics of a cross-organizational case management system for the umbrella organization of unions in Denmark, named LO.

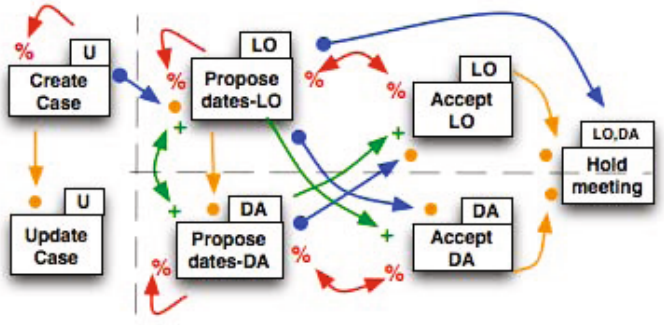


Fig. 2. Cross-organizational case management example

The graph has 7 events, drawn as boxes with "ears", and captures a process of creating a case, agreeing on meeting dates and holding meetings. The names of the events are written inside the box and the set of actions for each event, representing the roles that can execute the event, is written inside the "ear". That is, the event Create Case in the upper left has label U and represents the creation of a case by a case manager at a union (role U). The rightmost event, Hold meeting has two different labels, LO and DA, representing a meeting held by LO and DA (the umbrella organization of employers) respectively.

The semantics for DCR Graphs has been given in [14, 15] as a labelled transition system with acceptance condition for infinite computations. The set of accepted runs of DCR Graphs was characterized by a mapping to Büchi-automata in [26].

Definition 2. *For a DCR Graph* $G = (\mathsf{E_G}, \mathsf{M_G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L_G}, l_\mathsf{G})$*, we define the corresponding labelled transition system* $TS(G)$ *to be the tuple* $(\mathcal{M}(G), \mathsf{M_G}, \mathcal{EL}(G), \rightarrow)$ *where* $\mathcal{EL}(G) = \mathsf{E_G} \times \mathsf{L_G}$ *is the set of labels of the transition system,* $\mathsf{M_G} = (\mathsf{Ex_G}, \mathsf{In_G}, \mathsf{Re_G}) \in \mathcal{M}(G)$ *is the initial marking, and* $\rightarrow \subseteq \mathcal{M}(G) \times \mathcal{EL}(G) \times \mathcal{M}(G)$ *is the transition relation defined by* $\mathsf{M_G}' \xrightarrow{(e,a)} \mathsf{M_G}''$ *if*

(i) $\mathsf{M_G}' = (\mathsf{Ex_G}', \mathsf{In_G}', \mathsf{Re_G}')$ *is the marking before transition*
(ii) $\mathsf{M_G}'' = (\mathsf{Ex_G}'', \mathsf{In_G}'', \mathsf{Re_G}'')$ *is the marking after transition*
(iii) $e \in \mathsf{In_G}'$, $a \in l_\mathsf{G}(e)$
(iv) $\rightarrow\bullet e \cap \mathsf{In_G}' \subseteq \mathsf{Ex_G}'$,
(v) $\mathsf{Ex_G}'' = \mathsf{Ex_G}' \cup \{e\}$
(vi) $\mathsf{In_G}'' = (\mathsf{In_G}' \cup e\rightarrow+) \setminus e\rightarrow\%$,
(vii) $\mathsf{Re_G}'' = (\mathsf{Re_G}' \setminus \{e\}) \cup e\bullet\rightarrow$,

We define a run $a_0, a_1, \ldots$ *of the transition system to be a sequence of labels of a sequence of transitions* $\mathsf{M_G}_i \xrightarrow{(e_i,a_i)} \mathsf{M_G}_{i+1}$ *starting from the initial marking. We define*

a run to be accepting if for the underlying sequence of transitions it holds that $\forall i \geq 0, e \in \mathsf{Re}_{\mathsf{G}_i}.\exists j \geq i.(e = e_j \vee e \notin \mathsf{In}_{\mathsf{G}_{j+1}})$. *In words, a run is accepting if every response event either happen at some later state or become excluded.*

Condition (iii) in the above definition expresses that, only events that are currently included and mapped to the labels in L_G can be executed, Condition (iv) requires that all condition events to e which are currently included should have been executed previously. Condition (v), (vi) and (vii) are the updates to the sets of executed, included events and required responses respectively. Note that an event e' can not be both included and excluded by the same event e, but an event may trigger itself as a response.

To ease keeping track of transition systems of different DCR Graphs we extend the transition system to transitions between graphs in the obvious way, writing $G \xrightarrow{(e,a)} G'$ if $G = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l_\mathsf{G})$, $\mathsf{M}_\mathsf{G} \xrightarrow{(e,a)} \mathsf{M}_\mathsf{G}{}'$ in $TS(G)$ and $G' = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}{}', \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l_\mathsf{G})$.

3 Projection and Composition

In this section we define projections and compositions of dynamic condition response graphs.

3.1 Projection

First we define how to project a DCR Graph G with respect to a *projection parameter* $\delta = (\mathsf{E}_\delta, \mathsf{L}_\delta)$, where $\mathsf{E}_\delta \subseteq \mathsf{E}_\mathsf{G}$ is a subset of the events of G and $\mathsf{L}_\delta \subseteq \mathsf{L}_\mathsf{G}$ is a subset of the labels.

Intuitivly, the projection $G_{|\delta}$ contains only those events and relations that are relevant for the execution of events in E_δ and the labeling is restricted to the set L_δ. This includes both the events in E_δ and any other event that can affect the marking, or ability to execute of an event in E_δ through one or more relations.

Definition 3. *If* $G = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l)$ *then* $G_{|\delta} = (\mathsf{E}_{\mathsf{G}|\delta}, \mathsf{M}_{\mathsf{G}|\delta}, \rightarrow\bullet_{|\delta}, \bullet\rightarrow_{|\delta}, \pm_{|\delta}, \mathsf{L}_\delta, l_{|\delta})$ *is the projection of* G *with respect to* $\delta \subseteq \mathsf{E}_\mathsf{G}$ *where:*

(i) $\mathsf{E}_{\mathsf{G}|\delta} = \rightarrow\mathsf{E}_\delta$, *for* $\rightarrow = \bigcup_{c \in C} c$, *and* $C = \{\mathsf{id}, \rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow+\rightarrow\bullet, \rightarrow\%\rightarrow\bullet\}$

(ii) $l_{|\delta}(e) = \begin{cases} l_\mathsf{G}(e) \cap \mathsf{L}_\delta & \text{if } e \in \mathsf{E}_\delta \\ \emptyset & \text{if } e \in \mathsf{E}_{\mathsf{G}|\delta} \backslash \mathsf{E}_\delta \end{cases}$

(iii) $\mathsf{M}_{\mathsf{G}|\delta} = (\mathsf{Ex}_{\mathsf{G}|\delta}, \mathsf{Re}_{\mathsf{G}|\delta}, \mathsf{In}_{\mathsf{G}|\delta})$ *where:*

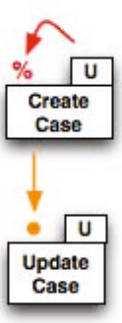
- *(a)* $\mathsf{Ex}_{\mathsf{G}|\delta} = \mathsf{Ex}_\mathsf{G} \cap \mathsf{E}_{\mathsf{G}|\delta}$
- *(b)* $\mathsf{Re}_{\mathsf{G}|\delta} = \mathsf{Re}_\mathsf{G} \cap \mathsf{E}_\delta$
- *(c)* $\mathsf{In}_{\mathsf{G}|\delta} = (\mathsf{In}_\mathsf{G} \cap ((\mathsf{id} \cup \rightarrow\bullet)\mathsf{E}_\delta)) \cup (\mathsf{E}_{\mathsf{G}|\delta} \setminus ((\mathsf{id} \cup \rightarrow\bullet)\mathsf{E}_\delta))$

(iv) $\rightarrow\bullet_{|\delta} = \rightarrow\bullet \cap ((\rightarrow\bullet\, \mathsf{E}_\delta) \times \mathsf{E}_\delta)$

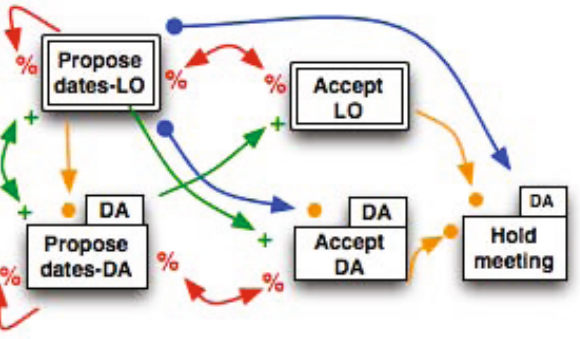
(v) $\bullet\rightarrow_{|\delta} = \bullet\rightarrow \cap ((\bullet\rightarrow\, \mathsf{E}_\delta) \times \mathsf{E}_\delta)$

(vi) $\rightarrow+_{|\delta} = \rightarrow+ \cap (((\rightarrow+\rightarrow\bullet\, \mathsf{E}_\delta) \times (\rightarrow\bullet\, \mathsf{E}_\delta)) \cup ((\rightarrow+\, \mathsf{E}_\delta) \times \mathsf{E}_\delta))$

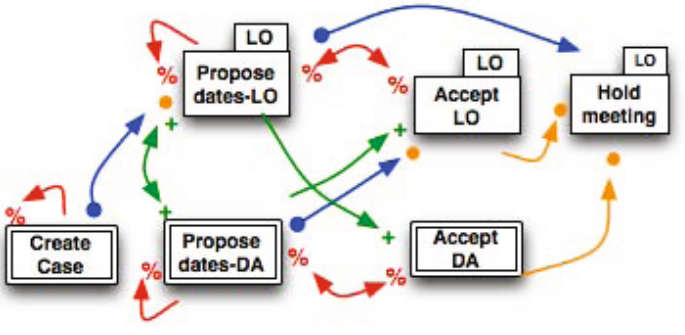
(vii) $\rightarrow\%_{|\delta} = \rightarrow\% \cap (((\rightarrow\%\rightarrow\bullet\, \mathsf{E}_\delta) \times (\rightarrow\bullet\, \mathsf{E}_\delta)) \cup ((\rightarrow\%\, \mathsf{E}_\delta) \times \mathsf{E}_\delta))$



(a) Projection Over Role U



(b) Projection Over Role DA



(c) Projection Over Role LO

Fig. 3. Projecting of Arrange Meeting Example Over Roles

(*i*) defines the set of events as the union of the set E_δ of events that we project over, any event that has a direct relation towards an event in E_δ and events that exclude or include an event which is a condition for an event in E_δ. The additional events will be included in the projection without labels, as can be seen from the definition of the labeling function in (*ii*). This means that the events can not be executed locally. However, when composed in a network containing other processes that can execute these events, their execution will be communicated to the process. For this reason we refer to these events as the (additional) external events of the projection. As proven in Prop. 1 the communication of the execution of this set of external events in addition to the local events shared by others ensure that the local state of the projection stay consistent with the global state. (*iii*) defines the projection of the marking: The executed events remain the same, but are limited to the events in $\mathsf{E}_{\mathsf{G}_{|\delta}}$. The responses are limited to events in E_δ because these are the only responses that will affect the local execution of the projected graph. The set of included events remains the same for events in E_δ or ${\mathsf{E}_\delta}^{-\bullet}$, because these can affect which events are enabled in the projected graph. All other external events of the projected graph are included regardless of their state in the marking of the

global graph. This is because in the local process is only notified of the execution of these events, not their in- or exclusion. Finally, *(iv)*, *(v)*, *(vi)* and *(vii)* state which relations should be included in the projection. For the events in E_δ all incoming relations should be included. Additionally inclusion and exclusion relations to events that are a condition for an event in E_δ are included as well.

To define networks of communicating DCR Graphs and their semantics we use the following extension of a DCR Graph allowing any event to be executed with a special input label. These transitions will only be used for the communication in a network and thus not be visible as user events.

Definition 4. *For a DCR Graph* $G = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l)$ *define* $G^\flat = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G} \cup \{\flat\}, l^\flat)$*, where* $l^\flat = l(e) \cup \{\flat\}$ *(assuming that* $\flat \notin \mathsf{L}_\mathsf{G}$*).*

We are now ready to state the key correspondence between global execution of events and the local execution of events in a projection.

Proposition 1. *Let* $G = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l)$ *be a DCR Graph and* $G_{|\delta}$ *its projection with respect to a projection parameter* $\delta = (\mathsf{E}_\delta, \mathsf{L}_\delta)$*. Then*

1. *for* $e \in \mathsf{E}_\delta$ *it holds that* $G \xrightarrow{(e,a)} G'$ *if and only if* $G_{|\delta} \xrightarrow{(e,a)} G'_{|\delta}$,
2. *for* $e \notin \mathsf{E}_{\mathsf{G}|\delta}$ *it holds that* $G \xrightarrow{(e,a)} G'$ *implies* $G_{|\delta} = G'_{|\delta}$,
3. *for* $e \in \mathsf{E}_{\mathsf{G}|\delta}$ *it holds that* $G \xrightarrow{(e,a)} G'$ *implies* $(G_{|\delta})^\flat \xrightarrow{(e,\flat)} (G'_{|\delta})^\flat$,

3.2 Composition

Now we define the binary composition of two DCR Graphs. Intuitively, the *composition* of G_1 and G_2 glues together the events that are both in G_1 and G_2.

Definition 5. *Formally, the composite* $G_1 \oplus G_2 = (\mathsf{E}_\mathsf{G}, \mathsf{M}_\mathsf{G}, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_\mathsf{G}, l)$*, where* $G_i = (\mathsf{E}_{\mathsf{G}i}, \mathsf{M}_{\mathsf{G}i}, \rightarrow\bullet_i, \bullet\rightarrow_i, \pm_i, \mathsf{L}_{\mathsf{G}i}, l_i)$*,* $\mathsf{M}_{\mathsf{G}i} = (\mathsf{Ex}_{\mathsf{G}i}, \mathsf{Re}_{\mathsf{G}i}, \mathsf{In}_{\mathsf{G}i})$ *and:*

(i) $\mathsf{E}_\mathsf{G} = (\mathsf{E}_{\mathsf{G}1} \cup \mathsf{E}_{\mathsf{G}2})$
(ii) $\mathsf{M}_\mathsf{G} = (\mathsf{Ex}_\mathsf{G}, \mathsf{Re}_\mathsf{G}, \mathsf{In}_\mathsf{G})$*, where:*
  (a) $\mathsf{Ex}_\mathsf{G} = \mathsf{Ex}_{\mathsf{G}1} \cup \mathsf{Ex}_{\mathsf{G}2}$
  (b) $\mathsf{In}_\mathsf{G} = (\mathsf{In}_{\mathsf{G}1} \cup \mathsf{In}_{\mathsf{G}2}) \setminus (((\mathsf{E}_{\mathsf{G}1}^i \cup \rightarrow\bullet\mathsf{E}_{\mathsf{G}1}^i) \setminus \mathsf{In}_{\mathsf{G}1}) \cup ((\mathsf{E}_{\mathsf{G}2}^i \cup \rightarrow\bullet\mathsf{E}_{\mathsf{G}2}^i) \setminus \mathsf{In}_{\mathsf{G}2}))$
  (c) $\mathsf{Re}_\mathsf{G} = \mathsf{Re}_{\mathsf{G}1} \cup \mathsf{Re}_{\mathsf{G}2}$
  for $\mathsf{E}_{\mathsf{G}j}^i = \{e \in \mathsf{E}_{\mathsf{G}j} \mid l_j(e) \neq \emptyset\}$
(iii) $\rightarrow = \rightarrow_1 \cup \rightarrow_2$ *for each* $\rightarrow \in \{\rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%\}$
(iv) $l(e) = l_1(e) \cup l_2(e)$
(v) $\mathsf{L}_\mathsf{G} = \mathsf{L}_{\mathsf{G}1} \cup \mathsf{L}_{\mathsf{G}2}$

(iib) states that events are included, if they're either included in G_1 or G_2, unless they are events that are either internal or have a condition towards an internal event and are excluded in G_1 or G_2. The intuition here is that if an event is internal or has a condition towards an internal event, then it affects the enabled events of the graph, so it's inclusion status should be the same in the composed graph. The inclusion/exclusion

status of other external events however may simply not have been updated because the graph is not aware of all relations towards these events. This is not unsafe because the inclusion of these events does not affect the execution of the graph. Therefor the definition states that if an event is internal or has a condition towards an internal event in G_1 or G_2, then it's inclusion status should be the same in the composed graph, and in any other case the event is included if it was included in G_1 or G_2. (iic) states that the events with pending responses are those events that have a pending response in G_1 or G_2.

Definition 6. *The composition $G_1 \oplus G_2$ is* well-defined *when:*

(i) $\forall(e \in \mathsf{E}_{G1} \cap \mathsf{E}_{G2} \mid (e \in \mathsf{Ex}_{G1} \Leftrightarrow e \in \mathsf{Ex}_{G2})$
(ii) $\forall(e \in (\mathsf{E}^i_{G1} \cup \rightarrow\bullet\mathsf{E}^i_{G1}) \cap (\mathsf{E}^i_{G2} \cup \rightarrow\bullet\mathsf{E}^i_{G2}) \mid (e \in \mathsf{In}_{G1} \Leftrightarrow e \in \mathsf{In}_{G2})$
(iii) $\forall(e \in \mathsf{E}^i_{G1} \cap \mathsf{E}^i_{G2} \mid (e \in \mathsf{Re}_{G1} \Leftrightarrow e \in \mathsf{Re}_{G2})$
(iv) $\forall(e, e' \in \mathsf{E}_{G1} \cap \mathsf{E}_{G2} \mid \neg((e \rightarrow_{+1} e' \wedge e \rightarrow\!\!\%_{2}\ e') \vee (e \rightarrow\!\!\%_{1}\ e' \wedge e \rightarrow_{+2} e')))$

(i) ensures that those events that will be glued together have the same execution marking. (ii) ensures that events that will be glued together and in both DCR Graphs belong to either the set of internal events or the set of events that have a conditional relation towards an internal event, have the same inclusion marking. (iii) ensures that events that will be glued together and in both DCR Graphs belong to the set of internal events have the same pending response marking. (iv) ensures that by composing the two DCR Graphs no event both includes and excludes the same event. If $G_1 \oplus G_2$ is well-defined, then we also say that G_1 and G_2 are *composable* with respect to eachother.

Lemma 1. *The composition operator $\oplus$ is commutative and associative.*

Definition 7. *We call a vector $\Delta = \delta_1 \ldots \delta_k$ of projection parameters* covering *for some DCR Graph $G = (\mathsf{E}_G, \mathsf{M}_G, \rightarrow\bullet, \bullet\rightarrow, \pm, \mathsf{L}_G, l_G)$ if:*

1. $\bigcup_{i \in [k]} \mathsf{E}_{\delta i} = \mathsf{E}_G$ *and*
2. $(\forall a \in \mathsf{L}_G . \forall e \in \mathsf{E}_G . a \in l_G(e) \Rightarrow (\exists i \in [k] . e \in \mathsf{E}_{\delta_i} \wedge a \in \mathsf{L}_{\delta_i})$

Proposition 2. *If some vector $\Delta = \delta_1 \ldots \delta_k$ of projection parameters is covering for some DCR Graph G then: $\bigoplus_{i \in [k]} G_{|\delta_i} = G$*

3.3 Safe Distributed Synchronous Execution

In this section we define networks of synchronously communicating DCR Graphs and prove the main technical theorem of the paper stating that a network of synchronously communicating DCR Graphs obtained by projecting a DCR Graph G with respect to a covering set of projection parameters has the same behavior as the original graph G.

Definition 8. *We define a* network of synchronously communicating DCR Graphs N *by the grammar*

$$N := G \mid N \| N$$

and let $\mathcal{N}_{\mathsf{E} \times \mathsf{L}}$ be the set of all networks with events in E and labels in L.

We write $\Pi_{i\in[n]}G_i$ for $G_1\|G_2\|\ldots\|G_n$. We define the set of events of a network of graphs inductively by $\mathcal{E}(G) = \mathsf{E}_\mathsf{G}$ and $\mathcal{E}(N_1\|N_2) = \mathcal{E}(N_1)\cup\mathcal{E}(N_2)$. Similarly, we define the set of labels of a network of graphs inductively by $\mathcal{L}(G) = \mathsf{L}_\mathsf{G}$ and $\mathcal{L}(N_1\|N_2) = \mathcal{L}(N_1)\cup\mathcal{L}(N_2)$.

Definition 9. *The semantics of networks of buffered DCR Graphs are given by the following inference rules:*

input
$$\frac{G_1^\flat \xrightarrow{(e,\flat)} G_2^\flat}{G_1 \overset{\triangleright e}{\rightsquigarrow} G_2}$$

sync input
$$\frac{N_1 \overset{\triangleright e}{\rightsquigarrow} N_1' \qquad N_2 \overset{\triangleright e}{\rightsquigarrow} N_2'}{N_1\|N_2 \overset{\triangleright e}{\rightsquigarrow} N_1'\|N_2'}$$

local input
$$\frac{N_i \overset{\triangleright e}{\rightsquigarrow} N_i' \qquad e\notin\mathcal{E}(N_{1-i})}{N_0\|N_1 \overset{\triangleright e}{\rightsquigarrow} N_0'\|N_1} \quad N_{1-i}=N_{1-i}', i\in\{0,1\}$$

sync step
$$\frac{N_i \xrightarrow{(e,a)} N_i' \qquad N_{1-i} \overset{\triangleright e}{\rightsquigarrow} N_{1-i}'}{N_0\|N_1 \xrightarrow{(e,a)} N_0'\|N_1'} \; i\in\{0,1\}$$

local step
$$\frac{N_i \xrightarrow{(e,a)} N_i' \qquad e\notin\mathcal{E}(N_{i-1})}{N_0\|N_1 \xrightarrow{(e,a)} N_0'\|N_1} \quad N_{1-i}=N_{1-i}', i\in\{0,1\}$$

For a network of synchronously communicating DCR Graphs N we define the corresponding transition system $TS(N)$ by $(\mathcal{N}_{\mathcal{EL}(N)}, N, \mathcal{EL}(N), \rightarrow\subseteq \mathcal{N}_{\mathcal{EL}(N)}\times\mathcal{EL}(N)\times\mathcal{N}_{\mathcal{EL}(N)})$ where $\mathcal{EL}(N) = \mathcal{E}(N)\times\mathcal{L}(N)$ and the transition relation $\rightarrow\subseteq \mathcal{N}_{\mathcal{EL}(N)}\times\mathcal{EL}(N)\times\mathcal{N}_{\mathcal{EL}(N)}$ is defined by the inference rules above.

We define a run $a_0, a_1, \ldots$ of the transition system to be a sequence of labels of a sequence of transitions $N_i \xrightarrow{(e_i,a_i)} N_{i+1}$ starting from the initial network. We define a run for a network $N = \Pi_{i\in[n]}G_i$ to be accepting if for the underlying sequence of transitions it holds that $\forall j\in[n], \forall i\geq 0, e\in\mathsf{Re}_{\mathsf{G}j,i}.\exists k\geq i.(e=e_k\vee e\notin\mathsf{In}_{\mathsf{G}j,k+1})$, where $\mathsf{Re}_{\mathsf{G}j,i}$ is the set of required responses in the jth DCR Graph in the network in the ith step of the run. In words, a run is accepting if every response event in a local DCR Graph in the network either happen at some later state or become excluded.

We are now ready to give the main theorem of the paper, stating the correspondence between a global DCR Graph and the network of synchronously communicating DCR Graph obtained from a covering projection.

Theorem 1. *For a Dynamic Condition Response Graph G and a covering vector of projection parameters $\Delta = \delta_1\ldots\delta_n$ it holds that $TS(G)$ is bisimilar to $TS(G_\Delta)$, where $G_\Delta = \Pi_{i\in[n]}G_{|\delta_i}$. Moreover, a run is accepting in $TS(G)$ if and only if the bisimilar run is accepting in $TS(G_\Delta)$.*

3.4 Example

In this section, we will use the arrange meeting example from Sec. 1 and show how events are executed in distributed setting. We assume the arrange meeting example is projected to a network $G^1_u \parallel G^1_{da} \parallel G^1_{lo}$ of three DCR Graphs as shown in the Fig. 3 and described in Sec. 3 and abbreviate the names for the events.

1. Using *sync step*, *local input*, and *input* we get the transition $G^1_u \parallel G^1_{da} \parallel G^1_{lo} \xrightarrow{(\texttt{Cc},\texttt{U})} G^2_u \parallel G^1_{da} \parallel G^2_{lo}$ capturing the local execution of the event `Cc` labelled with `U` in G^1_u which is communicated synchronously to G^1_{lo}. This updates the markings by adding the event `Cc` to the set of executed events in both G^1_u and G^1_{lo}. But since `Cc` has an exclude relation to itself in both G^1_u and G^1_{lo} (see Fig. 3(a) and 3(c)), the event is also excluded from the set of included events in both markings. Finally, because of the response relation to the event `PdLO` in G^1_{lo} (see Fig. 3(c)), the event `PdLO` is added to the set of required responses in the resulting marking G^2_{lo}.
2. We can now execute the event `PdLO` in the DCR graph G^2_{lo} concurrently with the event `Uc` in DCR graph G^2_u.

   As the event `Uc` is only local to G^2_u we get by using *local step* the transition $G^2_u \parallel G^1_{da} \parallel G^2_{lo} \xrightarrow{(\texttt{Uc},\texttt{U})} G^3_u \parallel G^1_{da} \parallel G^2_{lo}$ that only updates the marking of G^2_u.

   In addition to being local to G^2_{lo}, the event `PdLO` is also external event in graph G^1_{da}, so as in the first step by using *sync step local input*, and *input* we get the transition $G^3_u \parallel G^1_{da} \parallel G^2_{lo} \xrightarrow{(\texttt{PdLO},\texttt{LO})} G^3_u \parallel G^2_{da} \parallel G^3_{lo}$, where the event `PdLO` has been added to the executed event set of both the marking of G^1_{da} and G^2_{lo}. Again, because of the self-exclusion relations, the event `PdLO` is also excluded from the sets of included events in the two markings, and because of the response relations, the events `ADA` and `Hm` are added to the set of pending responses in G^1_{da} and the event `Hm` is added to the set of pending responses in G^2_{lo}.
3. In response to the dates proposed by `LO`, the `DA` may choose to propose new dates by executing the event `PdDA` in the graph graph G^2_{da}.

   $G^3_u \parallel G^2_{da} \parallel G^3_{lo} \xrightarrow{(\texttt{PdDA},\texttt{DA})} G^3_u \parallel G^3_{da} \parallel G^4_{lo}$ This triggers the exclusion of the events `PdDA` and `ADA` and the inclusion of the events `PdLO` and `ALO` in the markings of both G^2_{da} and G^3_{lo}. It will also include the event `ALO` in the required response set in the resulting marking G^4_{lo}.
4. Now `LO` may choose to accept the new dates proposed by `DA` by executing the event `ALO` in the graph graph G^4_{lo}, giving the transition

   $G^3_u \parallel G^3_{da} \parallel G^4_{lo} \xrightarrow{(\texttt{ALO},\texttt{LO}))} G^3_u \parallel G^4_{da} \parallel G^5_{lo}$. This records the event `ALO` as executed in markings of both G^4_{da} and G^5_{lo} and excludes `PdLO` in both markings (i.e. it is not possible to propose new dates after acceptance).
5. Since the event `ALO` is recorded as executed in markings of both G^4_{da} and G^5_{lo} and the event `ADA` is excluded, the hold meeting event `Hm` will be enabled in both graphs G^5_{lo} and G^4_{da}. The `LO` may choose to hold the meeting, giving the transition $G^3_u \mid \triangleright G^4_{da} \parallel G^5_{lo} \xrightarrow{(\texttt{Hm},\texttt{LO})} G^3_u \parallel G^5_{da} \parallel G^6_{lo}$

   Note that this event is also communicated to `DA`, added to the set of executed events and removed from the set of pending responses. Since there are no pending responses in any of the local graphs the finite run is in an accepting state.

4 Conclusion

We have given a general technique for distributing a declarative (global) process as a network of synchronously communicating (local) declarative processes and proven the global and distributed execution to be equivalent.

The global and local processes are given as Dynamic Condition Response (DCR) Graphs, a recently introduced declarative process model generalizing labelled prime event structures to a systems model able to finitely represent ω-regular languages. The DCR Graph model has the advantage that it is on the one hand declarative and compositional, and on the other hand it has a simple and intuitive operational semantics given as a transition semantics between markings of the graph. This allows the model to be used both as specification and execution model.

As briefly surveyed in Sec. 1.1 there have been a lot of related work on synthesis of distributed systems and proving consistency with respect to a global model or property. We believe this is the first treatment where both the local and global models are given declaratively in the same model. This maintains the flexibility of a declarative model for the local processes, and allows local processes to be further distributed if necessary.

We exemplified the safe distribution technique on a process identified in a case study of an inter-organizational case management system carried out jointly with Exformatics A/S.

We leave for future work to study the harder problem of asynchronously communicating distributed processes. This may benefit from researching the true concurrency semantics inherent in the model and extend the transition semantics to include concurrency, e.g. like in [18, 28]. We also plan to study behavioral types describing the interfaces between communicating DCR Graphs, extending the work on session types in [4] to a declarative setting. Moreover, we intend to address extension of the DCR Graph model with time, data and dynamic instantiation of sub processes (also referred to multiple instances) to be able to model more realistic workflow processes. A first step is taken in [17] extending DCR Graphs to allow nested sub graphs. This extension introduced an additional relation between events, the milestone relation, making it possible to express the acceptance of a sub graph succinctly. We believe the results in the present paper can be extended to nested DCR Graphs and the milestone relation, although it will complicate the definition of projections.

Finally, we plan to continue the ongoing implementation of tools for DCR Graphs, and in particular to implement the safe distribution technique in the current prototype design and simulation tools briefly described in [16].

References

1. van der Aalst, W.M.P., Weske, M.: The P2P Approach to Interorganizational Workflows. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 140–156. Springer, Heidelberg (2001)
2. Bravetti, M., Tennenholtz, M.: Contract Based Multi-party Service Composition. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 207–222. Springer, Heidelberg (2007)

3. Bravetti, M., Zavattaro, G.: A theory of contracts for strong service compliance. Mathematical. Structures in Comp. Sci. 19, 601–638 (2009)
4. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
5. Castellani, I., Mukund, M., Thiagarajan, P.: Synthesizing Distributed Transition Systems from Global Specifications. In: Pandu Rangan, C., Raman, V., Sarukkai, S. (eds.) FST TCS 1999. LNCS, vol. 1738, pp. 219–231. Springer, Heidelberg (1999)
6. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. 32(3), 3–9 (2009)
7. Das, S., Kochut, K., Miller, J., Sheth, A., Worah, D.: Orbwork: A reliable distributed corba-based workflow enactment system for meteor2. Technical report, The University of Georgia (1996)
8. Fahland, D.: Towards analyzing declarative workflows. In: Autonomous and Adaptive Web Services (2007)
9. Fdhila, W., Godart, C.: Toward synchronization between decentralized orchestrations of composite web services. In: CollaborateCom 2009, pp. 1–10 (2009)
10. Fdhila, W., Yildiz, U., Godart, C.: A flexible approach for automatic process decentralization using dependency tables. In: International Conference on Web Services (2009)
11. Fu, X., Bultan, T., Su, J.: Realizability of conversation protocols with message contents. In: Proceedings of the IEEE International Conference on Web Services, ICWS 2004, p. 96. IEEE Computer Society, Washington, DC, USA (2004)
12. Heljanko, K., Stefanescu, A.: Complexity results for checking distributed implementability. In: Proceedings of the Fifth International Conference on Application of Concurrency to System Design, pp. 78–87 (2005)
13. Hildebrandt, T.: Trustworthy pervasive healthcare processes (TrustCare) research project. Webpage (2008), http://www.trustcare.dk/
14. Hildebrandt, T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: Post-proceedings of PLACES 2010 (2010)
15. Hildebrandt, T., Mukkamala, R.R.: Distributed dynamic condition response structures. In: Pre-proceedings of International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2010) (March 2010)
16. Hildebrandt, T., Mukkamala, R.R., Slaats, T.: Designing a cross-organizational case management system using dynamic condition response graphs. In: Proceedings of IEEE International EDOC Conference (to appear, 2011), http://www.itu.dk/people/rao/pubs_accepted/dcrscasestudy-edoc11.pdf
17. Hildebrandt, T., Mukkamala, R.R., Slaats, T.: Nested dynamic condition response graphs. In: Proceedings of Fundamentals of Software Engineering (FSEN) (to appear, April 2011)
18. Hildebrandt, T., Sassone, V.: Comparing transition systems with independence and asynchronous transition systems. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 84–97. Springer, Heidelberg (1996)
19. Khalaf, R., Leymann, F.: Role-based decomposition of business processes using BPEL. In: International Conference on Web Services, ICWS 2006, pp. 770–780 (September 2006)
20. Kindler, E., Martens, A., Reisig, W.: Inter-operability of Workflow Applications: Local Criteria for Global Soundness. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) BPM. LNCS, vol. 1806, pp. 235–253. Springer, Heidelberg (2000)
21. Martens, A.: Analyzing Web Service Based Business Processes. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 19–33. Springer, Heidelberg (2005)

22. Milosevic, Z., Sadiq, S., Orlowska, M.: Towards a Methodology for Deriving Contract-Compliant Business Processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 395–400. Springer, Heidelberg (2006)
23. Mitra, S., Kumar, R., Basu, S.: Optimum decentralized choreography for web services composition. In: Proceedings of the 2008 IEEE International Conference on Services Computing, vol. 2 (2008)
24. Mohan, C., Agrawal, D., Alonso, G., El Abbadi, A., Guenthoer, R., Kamath, M.: Exotica: a project on advanced transaction management and workflow systems. SIGOIS Bull. 16, 45–50 (1995)
25. Montali, M.: Specification and Verification of Declarative Open Interaction Models. LNBIP, vol. 56. Springer, Heidelberg (2010)
26. Mukkamala, R.R., Hildebrandt, T.: From dynamic condition response structures to büchi automata. In: Proceedings of 4th IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2010) (August 2010)
27. Mukund, M.: From global specifications to distributed implementations. In: Synthesis and Control of Discrete Event Systems. Springer, Heidelberg (2002)
28. Mukund, M., Nielsen, M.: Ccs, locations and asynchronous transition systems. In: Shyamasundar, R. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 328–341. Springer, Heidelberg (1992)
29. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. SIGPLAN Not. 39, 170–187 (2004)
30. OASIS WSBPEL Technical Committee. Web Services Business Process Execution Language, version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
31. Paul, S., Park, E., Chaar, J.: Rainman: a workflow system for the internet. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems (1997)
32. Ranno, F., Shrivastava, S.K.: A review of distributed workflow management systems. In: Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (1999)
33. Reichert, M.U., Bauer, T., Dadam, P.: Flexibility for distributed workflows. In: Handbook of Research on Complex Dynamic Process Management: Techniques for Adaptability in Turbulent Environments, pp. 137–171. IGI Global, Hershey (2009)
34. Reichert, M., Bauer, T.: Supporting Ad-Hoc Changes in Distributed Workflow Management Systems. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 150–168. Springer, Heidelberg (2007)
35. Rinderle, S., Wombacher, A., Reichert, M.: Evolution of Process Choreographies in DYCHOR. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 273–290. Springer, Heidelberg (2006)
36. Sadiq, W., Sadiq, S., Schulz, K.: Model driven distribution of collaborative business processes. In: IEEE International Conference on Services Computing, 2006. SCC 2006, pp. 281–284 (September 2006)
37. Sassone, V., Nielsen, M., Winskel, G.: A classification of models for concurrency. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 82–96. Springer, Heidelberg (1993)
38. Sassone, V., Nielsen, M., Winskel, G.: Models for concurrency: Towards a classification. Theoretical Computer Science 170, 297–348 (1996)
39. Wheater, S.M., Shrivastava, S.K., Ranno, F.: A corba compliant transactional workflow system for internet applications. In: Proc. of IFIP Intl. Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware 1998, pp. 1–85233. Springer, Heidelberg (1998)

40. ter Hofstede, A., van Glabbeek, R., Stork, D.: Query Nets: Interacting Workflow Modules That Ensure Global Termination. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 184–199. Springer, Heidelberg (2003)
41. van der Aalst, W.M.P.: Interorganizational workflows: An approach based on message sequence charts and petri nets. Systems Analysis - Modelling - Simulation 34(3), 335–367 (1999)
42. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty Contracts: Agreeing and Implementing Interorganizational Processes. The Computer Journal 53(1), 90–106 (2010)
43. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D 23(2), 99–113 (2009)
44. van der Aalst, W.M.P., Pesic, M.: A declarative approach for flexible business processes management. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
45. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)
46. van der Aalst, W.M.P.: Inheritance of interorganizational workflows: How to agree to disagree without loosing control? Information Technology and Management 4, 345–389 (2003)
47. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)
48. Wodtke, D., Weikum, G.: A formal foundation for distributed workflow execution based on state charts. In: Afrati, F.N., Kolaitis, P.G. (eds.) ICDT 1997. LNCS, vol. 1186, pp. 230–246. Springer, Heidelberg (1996)
49. Yi, X., Kochut, K.J.: Process composition of web services with complex conversation protocols. In: Design, Analysis, and Simulation of Distributed Systems Symposium at Adavanced Simulation Technology (2004)
50. Zielonka, W.: Notes on finite asynchronous automata. Informatique Théorique et Applications 21(2), 99–135 (1987)

Verifying B Proof Rules Using Deep Embedding and Automated Theorem Proving

Mélanie Jacquel[1], Karim Berkani[1], David Delahaye[2], and Catherine Dubois[3]

[1] Siemens SAS I MO, Châtillon, France
{Melanie.Jacquel,Karim.Berkani}@siemens.com
[2] CEDRIC/CNAM, Paris, France
David.Delahaye@cnam.fr
[3] CEDRIC/ENSIIE, Évry, France
dubois@ensiie.fr

Abstract. We propose a formal and mechanized framework which consists in verifying proof rules of the B method, which cannot be automatically proved by the elementary prover of Atelier B and using an external automated theorem prover called Zenon. This framework contains in particular a set of tools, named BCARe and developed by Siemens SAS I MO, which relies on a deep embedding of the B theory within the logic of the Coq proof assistant and allows us to automatically generate the required properties to be checked for a given proof rule. Currently, this tool chain is able to automatically verify a part of the derived rules of the B-Book, as well as some added rules coming from Atelier B and the rule database maintained by Siemens SAS I MO.

Keywords: B Method, Proof Rules, Verification, Deep Embedding, Automated Theorem Proving, Coq, Zenon.

1 Introduction

The B method [1], or B for short, allows engineers to develop software with high guarantees of confidence; more precisely it allows them to build correct by design software. B is a formal method based on theorem proving and emphasizing a refinement-based development process. A typical scenario consists in first writing high-level formal specifications as abstract machines, and then refining them step by step into low-level sequential pseudo-code that can be automatically translated into C or Ada programs. Proof is required to verify the correctness of abstract machines (mainly ensuring the preservation of user-written invariant properties) and the correctness of the refinement steps (roughly speaking, the behavior is preserved by the refinement steps that introduce algorithmic decisions or data representation choices). In practice, it means that the user must discharge proof obligations. The Atelier B environment [8] is a platform that supports B and offers, among other tools, both automated and interactive provers.

A famous and significant use of B and Atelier B has concerned the control system of the driverless Meteor line 14 metro in Paris (opened 13 years ago).

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 253–268, 2011.

Since Meteor, Siemens SAS I MO has generalized its use of B for building other critical systems, e.g. the communication-based train control systems of the New York City Canarsie line. On both cases, a huge number of proof obligations (27,800 obligations for Meteor) had to be handled manually (using the interactive prover). In fact, most of them were proved by adding new proof rules (1,400 rules for Meteor) that the automated provers can exploit. Even today, many projects developed at Siemens SAS I MO still require to add new proof rules.

These new proof rules must of course be proved correct, otherwise the proof process is invalid. The proof of these proof rules is done by the elementary prover provided by Atelier B, which does not use any of them. Some of the added rules (900 rules in the Meteor case) can be proved by the Atelier B elementary prover, some of them cannot and are then proved manually by experts.

The main point in this approach is to prove the proof rules, basic or added ones. Currently, the proof rule database of Atelier B used at Siemens SAS I MO contains about 5,300 proof rules, 2,900 of which can be proved automatically by the elementary prover. The problem raised here is not to question the confidence into the Atelier B elementary prover, but to prove the proof rules that are not automatically proved and that must be proved manually. The latter process is tedious, long, and error-prone. We propose to replace it by a mechanized verification with the help of a more powerful Automated Theorem Prover (ATP), a first order one, e.g. Zenon [5]. In order to increase confidence in this external verification, it is important to be able to check the proofs done automatically, and furthermore to be able to connect them with the inference rules of the underlying B logic [1]. Thus, our approach is not only to use an external prover, but also to rely on a proof assistant which checks the generated proofs, i.e. Coq [15].

However, as Coq is not fully automated and may require human interaction, we propose to use Coq only to describe the B underlying logic and to serve as a proof verifier for the proofs delegated to the ATP. The expected results for a B proof rule will be, in case of success, a proof in the B logic (or, more technically, a Coq proof term encoding it). Some years ago, a first experiment using Coq has been conducted at Siemens SAS I MO to verify the Atelier B proof rules (see [3]), but it required human interaction and all the proofs done in Coq were done manually. However, this first manual attempt allowed us to handle 274 proof rules proved manually by experts and considered by the authors as representative ones. The methodology consisted in playing the manual proofs within Coq: 7 valid proof rules had incorrect proofs but the proofs could be given properly within Coq, and 13 rules were not valid because of lack of hypotheses about variable non-freeness. This discovery were important for the design of the verification platform presented in this paper, namely BCARe.

BCARe is a set of tools developed by Siemens SAS I MO to verify added proof rules. It contains tools to check if a proof rule is correctly protected by non-freeness assumptions, to typecheck a rule, and to prove a rule (by using Coq and Zenon). One of the main objectives of BCARe is to assist the experts to find proofs of proof rules, but the development standards used at Siemens SAS I MO expect these experts to give their final assessment. BCARe contains a deep embedding

of the B logic within Coq, that is an encoding of the B formulas and inference rules into the Coq logic, namely the calculus of inductive constructions. Other embeddings [4,6,7,12] of B have been implemented with different purposes in related work. For example, BiCoq [12] is a deep embedding of B in Coq. Like BCARe, BiCoq follows scrupulously the B-Book [1]; however, the former uses names whereas the latter uses De Bruijn indexes.

Some experiments like [9,14,10] concern automated verification of B proof obligations with ATPs or SMT solvers. We are interested in proof rules and not in proof obligations. Furthermore, as we do want the best degree of confidence in our mechanical proofs, it is essential to rely on an ATP able to provide proof traces checkable by a proof checker, e.g. Coq. Zenon is one of the ATPs able to provide several output proof formats, one of which is a Coq script that will be adapted to give us a proof using the B logic.

The paper is organized as follows: in Section 2, we first present the several steps required to verify B proof rules; we then introduce, in Section 3, the BCARe environment, which is a mechanized support for the verification of proof rules; finally, in Section 4, we describe our experiments for automating the verification proofs and provide some benchmarks concerning derived rules and added rules coming from Atelier B and the rule database maintained by Siemens SAS I MO.

2 Rule Verification in Atelier B

In this section, we present the notion of proof rules of Atelier B, together with the several steps required to ensure their verification, i.e. the steps which guarantee that the application of such rules does not introduce inconsistencies.

2.1 The B Set Theory

The B method [1] aims to assist experts to develop certified software. The initial step is defined with abstract properties of a model. Several steps of property refinement are then applied until the release of the complete software. A refinement step is characterized by adding details on the software behavior under construction. For each step, generated proof obligations must be demonstrated.

The B method is based on a typed set theory. There are two rule systems: one for demonstrating that a sentence is well-typed, and one for demonstrating that a sentence is a logical consequence of a set of axioms. The main aim of the type system is to avoid inconsistent sentences, such as Russell's paradox for example. The B proof system is based on a sequent calculus with equality. Six axiom schemes define the basic operators and the extensionality which, in turn, defines the equality of two sets. In addition, the other operators ($\cup$, $\cap$, etc) are defined using the previous basic ones.

2.2 The Atelier B Proof Assistant

Proofs. Atelier B [8] is a tool, developed by ClearSy, that implements the B method. Once a model is specified, its correctness is ensured by several mechanisms. The first one is typechecking, which is fully automated. If no error occurs

during typechecking, then the proof obligations can be generated. They represent the properties that must be proved to verify the mathematical correctness of the model compared with its properties. A proof system helps the developer make the corresponding demonstrations.

These proof obligations can be demonstrated automatically with a tactic of the Atelier B proof assistant. When this tactic fails, an interactive proof mode can be used. In this proof mode, the user can apply tactics on the goal and/or on the hypotheses to complete his/her proof. A tactic is an ordered list of theories (a theory is basically a container of rules), that determines the traversal of a rule base to determine if one or several rules can be applied.

Rules. We distinguish two kinds of rules in Atelier B: deduction and rewrite rules. The former is of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow B$, where the A_i are the antecedents (guards or predicates) and B the consequent (predicate). Guards are used to add some conditions to the use of the rule. A deduction rule can be used in both backward and forward ways. A rewrite rule is of the form $A_1 \wedge \ldots \wedge A_n \Rightarrow B == C$, where the A_i are the antecedents (guards or predicates) and where B and C are either expressions or predicates. The binary symbol "==" is the syntactic replacement: if B matches a subterm of the goal, then it is replaced by the corresponding instance of C. More precisely, the syntax of rules is defined as follows:

$$
\begin{array}{l}
V := I \mid V \mapsto V \\
E := V \mid [V := E]E \mid E \mapsto E \mid \mathit{choice}(S) \mid S \\
S := S \times S \mid \mathbb{P}(S) \mid \{V|P\} \mid \mathit{BIG} \mid I \\
P := P \wedge P \mid P \Rightarrow P \mid \neg P \mid \forall V.P \mid [V := E]P \mid E = E \mid E \in S \mid I \\
A := P \mid I \setminus P \mid I \setminus E \mid \mathsf{binhyp}(P) \mid \mathsf{blvar}(I) \mid A \wedge A \\
C := P \mid E == E \mid P == P \\
R := C \mid A \Rightarrow C
\end{array}
$$

where V represents the variables (in which I denotes the identifiers), E the expressions, S the sets, P the predicates, A the antecedents, C the consequents, and R the rules. Regarding guards, we only consider the non-freeness predicates ($I \setminus P$ and $I \setminus E$), as well as the binhyp and blvar guards, where $\mathsf{binhyp}(P)$ checks the presence of P in the proof context and $\mathsf{blvar}(I)$ instantiates I with the bound variables at the rewrite point. We are also able to deal with more complicated guards that we will not present in this paper. Furthermore, free variables may occur in E, S, and P; these variables are considered as metavariables (for pattern-matching). As for the other logical connectives (such as $\Leftrightarrow$, $\vee$, and $\exists$) and set operators (e.g. $\cup$, $\cap$, etc), they are defined from those given above.

Let us illustrate the two previous kinds of rules with some examples of added rules coming from Atelier B.

Example 1 (Deduction Rule). ForAllX.3: $(a \setminus A) \wedge A = \emptyset \Rightarrow \forall a.a \notin A$
If used in a backward way, this rule can be applied on a goal if a is non-free in A and if the current goal matches the consequent $\forall a.a \notin A$. It therefore generates the goal $A = \emptyset$ to be proved.

Example 2 (Rewrite Rule). SimplifyRelDorXY.2:
$\mathsf{binhyp}(f \in u \mathrel{+\!\!\!\rightarrow} v) \wedge \mathsf{binhyp}(a \in dom(f)) \wedge \mathsf{blvar}(Q) \wedge (Q \setminus (f \in u \mathrel{+\!\!\!\rightarrow} v)) \wedge (Q \setminus (a \in dom(f))) \Rightarrow \{a\} \lhd f == \{(a \mapsto f(a))\}$
This rule can be applied on a goal if there are some hypotheses of the context matching $f \in u \mathrel{+\!\!\!\rightarrow} v$ and $a \in dom(f)$, a term of the goal matching $\{a\} \lhd f$, and if the quantified variables at the rewrite point do not appear in u, v, a, and f.

Rule Verification. The verification of a rule is carried out in four steps:

1. The first step only deals with rewrite rules and consists in verifying that rewrite rules are correctly protected against variable capture. This is due to the fact that Atelier B does not verify the context of application when applying a rewrite rule and applies it in a purely syntactical way. As a consequence, when a rewrite rule is applied under binders and involves bound variables, variable capture may occur and lead to inconsistencies.
2. The second step aims to verify that the rule is well-formed, which amounts to typechecking the rule according to the B typing rules (see [1]). However, a rule may contain metavariables whose type may be left implicit. Therefore, a preliminary step is required to first infer the types of all metavariables such that the rule enriched with these type constraints can be typechecked.
3. The third step consists in verifying that the rule is well-defined. In [2], it is pointed out that conditional definitions may lead to some ill-defined expressions, such as division by zero or the application of a function to an argument lying outside its domain. A syntactical filter to be applied to the rule is proposed and contains all the well-definedness proof obligations.
4. The last step must verify that the rule can be derived using the B proof rules (see [1]). It is possible to do so over a rule, after applying another syntactical filter, defined in [2] in particular, in order to remove the proof obligations related to well-definedness.

3 The BCARe Environment

In this section, we present the BCARe environment, which is developed by Siemens SAS I MO, and which proposes a formal and mechanized framework for verifying B proof rules.

3.1 Rationale for Designing BCARe

Currently, an automated tool is used at Siemens SAS I MO for verifying the rules developed with Atelier B. However, when a proof fails, the rule is verified manually without the help of any proof assistant. The first aim of the BCARe environment, developed by Siemens SAS I MO, is to overcome this problem. For example, in the rule ForAllX.3, $a \setminus A$ must be verified before the application of the rule. It is possible to check that the previous condition is necessary with BCARe, while it is impossible to do so with the other available tools. Thus,

the BCARe environment has been essentially developed to deal with the rules whose correctness cannot be automatically established. The scope of this environment is currently a subset of the B set theory (propositional and first order logics, basic set theory operators, functions, generalized and quantified intersections and unions). Some other features, such as induction, are being integrated.

The different steps of a rule verification with BCARe follows the several steps defined in Section 2. If the rule is a rewrite rule then a tool checks that its guards correctly protect the free variables (see Subsection 3.3). Another tool then infers types for the rule using a type inference algorithm which has been defined regarding the B typing rules (see [1]), after which the typing lemma can be generated (see Subsection 3.4). Finally, this tool also generates the well-definedness lemma, as well as the lemma corresponding to the rule itself (See Subsections 3.5 and 3.6). Once these three lemmas are generated, their proofs must be completed. The generation of these lemmas and the corresponding proofs are realized using the Coq proof assistant [15]. In particular, this relies on an environment, called BCoq, which is an embedding of the B set theory in Coq (see Subsection 3.2). The proofs of these lemmas can be partially automated, and Section 4 describes our approaches regarding the automation of these proofs.

3.2 The BCoq Embedding

The generation of the previous lemmas is realized within the BCoq environment, which is a deep embedding of the B set theory in Coq, and where the B operators, as well as the deduction systems for types and proofs, are specified inductively (see [3]). Compared to a shallow embedding, the advantage of such an approach is that the correctness of a type or proof derivation is provided by construction.

The BCoq syntax is defined in Coq as follows (we do not provide the Coq concrete syntax, but only an abstraction of this syntax written with "."):

$$\dot{V} := \dot{I} \mid \dot{V} \mapsto \dot{V}$$
$$\dot{E} := \dot{V} \mid [\dot{V} := \dot{E}]\dot{E} \mid \dot{E} \mapsto \dot{E} \mid \dot{choice}\ (\dot{S}) \mid \dot{S}$$
$$\dot{S} := \dot{S} \times \dot{S} \mid \dot{\mathbb{P}}\ (\dot{S}) \mid \{\dot{V} \mid \dot{P}\} \mid \dot{BIG} \mid \dot{I}$$
$$\dot{P} := \dot{P} \wedge \dot{P} \mid \dot{P} \Rightarrow \dot{P} \mid \dot{\neg}\ \dot{P} \mid \dot{\forall}\ \dot{V}.\dot{P} \mid [\dot{V} := \dot{E}]\dot{P} \mid \dot{E} \doteq \dot{E} \mid \dot{E} \dot{\in} \dot{S} \mid \dot{I}$$

where $\dot{I}$, $\dot{V}$, $\dot{E}$, $\dot{S}$, and $\dot{P}$ respectively represent the reified versions of the several sets of terms I, V, E, S, and P, defined in Section 2.

In this grammar, there is no syntax for rules, as they are intended to be reified into predicates. However, there are still metavariables (free variables occurring in $\dot{E}$, $\dot{S}$, and $\dot{P}$). The reification process is performed by using the set of functions $[\![\cdot]\!]_X$, where $X \in \{V, E, S, P, A, C, R\}$, and which are defined in Figure 1. In this process, the sets of metavariables are computed, and these metavariables are then bound by means of shallow binders. The names of binders are managed using shallow binders as well, in order to deal with α-conversion and skolemization in particular (see Subsection 3.4). In the same way, the non-freeness guards are reified and kept in the term using shallow non-dependent products.

$[\![I_1]\!]_V = (I_2, \{(I_1, I_2)\})$, where $I_2 \notin \mathcal{V} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{I_2\}$

$[\![V_1 \mapsto V_2]\!]_V = (V_1' \dot{\mapsto} V_2', \{B_1 \cup B_2\})$, where $(V_1', B_1) = [\![V_1]\!]_V$ and $(V_2', B_2) = [\![V_2]\!]_V$

$$[\![I_1]\!]_V^b = \begin{cases} I_2, \text{ if } (I_1, I_2) \in b \\ I_1, \text{ otherwise and } \mathcal{M}_E \leftarrow \mathcal{M}_E \cup \{I_1\} \end{cases} \qquad [\![V_1 \mapsto V_2]\!]_V^b = [\![V_1]\!]_V^b \dot{\mapsto} [\![V_2]\!]_V^b$$

$[\![V_1]\!]_E^b = [\![V_1]\!]_V^b \qquad [\![[V_1 := E_1]E_2]\!]_E^b = [V_2 :\dot{=} [\![E_1]\!]_E^b][\![E_2]\!]_E^{b \cup B}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![E_1 \mapsto E_2]\!]_E^b = [\![E_1]\!]_E^b \dot{\mapsto} [\![E_2]\!]_E^b \qquad [\![S_1]\!]_E^b = [\![S_1]\!]_S^b$

$[\![S_1 \times S_2]\!]_S^b = [\![S_1]\!]_S^b \dot{\times} [\![S_2]\!]_S^b \qquad [\![\mathbb{P}(S_1)]\!]_S^b = \dot{\mathbb{P}}\,([\![S_1]\!]_S^b)$

$[\![\{V_1|P_1\}]\!]_S^b = \{V_2 \mid [\![P_1]\!]_P^{b \cup B}\}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![I_1]\!]_S^b = I_1$, and $\mathcal{M}_S \leftarrow \mathcal{M}_S \cup \{I_1\}$ if $I_1 \notin b$

$[\![P_1 \wedge P_2]\!]_P^b = [\![P_1]\!]_P^b \dot{\wedge} [\![P_2]\!]_P^b \qquad [\![P_1 \Rightarrow P_2]\!]_P^b = [\![P_1]\!]_P^b \dot{\Rightarrow} [\![P_2]\!]_P^b$

$[\![\neg P_1]\!]_P^b = \dot{\neg}\,[\![P_1]\!]_P^b \qquad [\![\forall V_1.P_1]\!]_P^b = \dot{\forall}\, V_2.[\![P_1]\!]_P^{b \cup B}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![[V_1 := E_1]P_1]\!]_P^b = [V_2 :\dot{=} [\![E_1]\!]_E^b][\![P_1]\!]_P^{b \cup B}$, where $(V_2, B) = [\![V_1]\!]_V$

$[\![E_1 = E_2]\!]_P^b = [\![E_1]\!]_P^b \doteq [\![E_2]\!]_P^b$

$[\![E_1 \in S_1]\!]_P^b = [\![E_1]\!]_E^b \,\dot{\in}\, [\![S_1]\!]_P^b \qquad [\![I_1]\!]_P^b = I_1$, and $\mathcal{M}_P \leftarrow \mathcal{M}_P \cup \{I_1\}$ if $I_1 \notin b$

$[\![P_1]\!]_A = [\![P_1]\!]_P^\emptyset \qquad [\![I_1 \setminus P_1]\!]_A = \top$, and $\mathcal{N} \leftarrow \mathcal{N} \cup [\![I_1]\!]_V^\emptyset \,\dot{\setminus}\, [\![P_1]\!]_P^\emptyset$ if $I_1 \notin \mathcal{R}$

$[\![I_1 \setminus E_1]\!]_A = \top$, and $\mathcal{N} \leftarrow \mathcal{N} \cup [\![I_1]\!]_V^\emptyset \,\dot{\setminus}\, [\![E_1]\!]_E^\emptyset$ if $I_1 \notin \mathcal{R}$

$[\![\mathsf{binhyp}(P_1)]\!]_A = [\![P_1]\!]_P^\emptyset \qquad [\![\mathsf{blvar}(I_1)]\!]_A = \top$ with $\mathcal{R} \leftarrow \mathcal{R} \cup \{I_1\}$

$$[\![A_1 \wedge A_2]\!]_A = \begin{cases} [\![A_i]\!]_A, \text{ if } [\![A_j]\!]_A = \top \text{ with } (i,j) = (1,2) \text{ or } (2,1) \\ \top, \text{ if } [\![A_i]\!]_A = \top \text{ with } i = 1,2 \\ [\![A_1]\!]_A \,\dot{\wedge}\, [\![A_2]\!]_A, \text{ otherwise} \end{cases}$$

$[\![P_1]\!]_C = [\![P_1]\!]_P^\emptyset \qquad [\![E_1 == E_2]\!]_C = [\![E_1 = E_2]\!]_P^\emptyset \qquad [\![P_1 == P_2]\!]_C = [\![P_1]\!]_P^\emptyset \,\dot{\Leftrightarrow}\, [\![P_2]\!]_P^\emptyset$

$$[\![A_1 \Rightarrow C_1]\!]_R = \begin{cases} \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.[\![C_1]\!]_C^\emptyset, \text{ if } [\![A_1]\!]_A = \top \\ \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.[\![A_1]\!]_A \,\dot{\Rightarrow}\, [\![C_1]\!]_C^\emptyset, \text{ otherwise} \end{cases}$$

$[\![C_1]\!]_R = \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.[\![C_1]\!]_C^\emptyset$

$\mathcal{V}$	is the set of variables of the initial rule.
$\mathcal{M}_{E/S/P}$	is the set of metavariables of expressions, sets, and predicates.
$\mathcal{M}$	is the set of metavariables $\mathcal{M}_E \cup \mathcal{M}_S \cup \mathcal{M}_P$.
$\mathcal{B}$	is the set of bound variables.
$\mathcal{N}$	is the set of non-freeness hypotheses of the initial rule.
$\mathcal{R}$	is the set of variables bounded by the guard blvar.

$\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.T \equiv$
$\forall_{x \in \mathcal{M}_E} x \in \dot{E}.\forall_{x \in \mathcal{M}_S} x \in \dot{S}.\forall_{x \in \mathcal{M}_P} x \in \dot{P}.\forall_{x \in \mathcal{B}} x \in \dot{I}.N_1 \rightarrow \ldots \rightarrow N_n \rightarrow T$,
where $\mathcal{N} = \{N_1, \ldots, N_n\}$ and T is a reified term.

Fig. 1. Reification of the Atelier B Rules

The BCoq environment also provides the reified relations "$\dot{\vdash}$" and "$\dot{\vdash}_\tau$", respectively for proof and typing judgments (see [1]).

3.3 Rewrite Rule Verification

As said in Section 2, Atelier B does not verify the context of application when applying a rewrite rule and applies it in a purely syntactical way. Therefore, when a rewrite rule is applied under binders and involves bound variables, variable capture may occur and lead to inconsistencies. For instance, let us consider the rewrite rule $\mathsf{binhyp}(x = a) \Rightarrow x == a$ and the goal $n = 0 \vdash \forall n.n \in \mathbb{N} \Rightarrow n = 0$. This goal is trivially false, but the rewrite rule can be applied, which leads to the goals $n = 0 \vdash \forall n.n \in \mathbb{N} \Rightarrow 0 = 0$ and $n = 0 \vdash n = 0$. These two goals can be completed and an inconsistency is then introduced (due to the capture of variable n in hypothesis by the binder of the conclusion).

To avoid variable capture, a first solution is to prevent us from performing rewriting under binders when bound variables are involved. Considering a rewrite rule of the form $G \wedge A \Rightarrow E == F$, where G is the conjunction of the guards, A the conjunction of the antecedents (other than guards), and E and F two expressions, this corresponds to the following criterion:

$$\mathsf{blvar}(Q) \wedge Q \setminus (E = F) \tag{1}$$

Using this criterion, the previous rewrite rule $\mathsf{binhyp}(x = a) \Rightarrow x == a$ is then rejected as the variables x and a are not protected. To be correct, this rule must be of the form $\mathsf{binhyp}(x = a) \wedge \mathsf{blvar}(Q) \wedge Q \setminus x = a \Rightarrow x == a$.

However, this criterion is a little too restrictive as it prevents us from defining some useful rewrite rules. For instance, the rewrite rule $s \cap t == t \cap s$ is rejected by this criterion whereas this rule cannot generate variable capture.

To accept this kind of rewrite rules, a second solution consists in allowing rewriting to be performed under binders only if the bound variables involved do not occur in the antecedents. More precisely, this criterion is defined as follows:

$$\mathsf{blvar}(Q) \wedge Q \setminus (G \wedge A) \tag{2}$$

With this criterion, the corrected rule and the other rule are both accepted. However, criteria (1) and (2) are complementary, and we use both of them as it allows us to accept more rules. Both criteria have been formally verified in [13].

3.4 Rule Typechecking

As seen in Section 2, it is required to verify that a rule is well-formed, which amounts to typechecking the rule according to the B typing rules (see [1]). However, a rule may contain metavariables whose type may be left implicit. For example, in the rule $a \cup b = b \cup a$, the types of a and b are unknown. The B type system does not allow us to infer types for metavariables occurring in rules. It only allows us to check that a predicate is well-typed when all the types are explicit. Therefore, we have to first infer a type for all metavariables.

To do so, we define a type inference system, which is described in Figure 2. The different rules correspond to the core language described in Subsection 3.2 and deal with reified rules. There are also some dedicated rules for other logical connectives and set operators, but they are not presented in this paper due to space restrictions. This type inference system is aimed to find types for variables of expressions (bound or not) and metavariables of sets, but not for metavariables of predicates which cannot be typechecked using the B typing rules. This is possible if metavariables of sets are not distinguished from variables of expressions, and in the following, a variable will denote either a variable of expression (bound or not), or a metavariable of set.

First, the algorithm assigns a unique type variable to each variable and all these variables with their types are gathered in a typing context Γ. The type inference tree is then built according to the rules of Figure 2. Some constraints may appear during this step due to some non-linearity constraints (for instance, see the rule "$\doteq$"). Once the tree is closed, the algorithm tries to solve the constraints. If it succeeds, the types of variables of Γ are updated with their instantiations.

Before generating the corresponding typing lemma, it is necessary to generate new hypotheses of non-freeness without which this lemma cannot be proved in general, as skolemization cannot be performed when eliminating binding terms. This is realized by means of the following operator:

$$\mathcal{S}_{U,V}(T) = N_1 \rightarrow \ldots \rightarrow N_n \rightarrow T$$

where T is a reified term, and for all $u \in U$, for all $v \in V$ such that $u \neq v$ and $u \setminus v \notin \mathcal{N}$, then there exists $i \in 1 \ldots n$ such that $N_i = u \setminus v$.

Finally, for a reified rule of the form $\forall x.\mathcal{M}, \mathcal{B}.\mathcal{N} \Rightarrow P$ and from the resulting typing context Γ, the typing lemma can be generated as follows:

$$\forall x.x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{S}_{\mathcal{B},\mathcal{B}\cup\mathcal{M}_\Gamma}(G, H \dot{\vdash}_\tau \dot{check}\ (P))$$

where: $\mathcal{M} \leftarrow \mathcal{M'}_E \cup \mathcal{M'}_S \cup \mathcal{M}_P$ with $\mathcal{M'}_E \leftarrow \mathcal{M}_E \cup \mathcal{M}_S$ and $\mathcal{M'}_S \leftarrow \mathcal{M}_\Gamma$, where $\mathcal{M}_\Gamma$ is the set of type variables of Γ; G is the reification of the types of Γ such that for all $(v,t) \in \Gamma$ and $v \notin \mathcal{B}$, $\dot{given}\ (t) \in G$ (in which $\dot{given}\ (t)$ means that t is the super-set of itself); H is the reification of the typing context Γ such that for all $(v,t) \in \Gamma$ and $v \notin \mathcal{B}$, $[\![v \in t]\!]_P \in H$.

3.5 Well-Definedness Verification

As said in Section 2, it is pointed out in [2] that conditional definitions may lead to some ill-defined expressions, such as the application of a function to an argument lying outside its domain. Thus, a syntactical filter to be applied to the rule to prove is proposed in [2], and contains all the proof obligations related to well-definedness. The filter is called $\mathcal{L}$ and is defined as a function over reified rules. The computation rules of this function are split into two sets of rules: decomposition and atomic rules. The decomposition rules are the following:

Rules for V

$$\frac{}{x : s, \Gamma \vdash x : s}\ var \qquad \frac{\Gamma \vdash x : s \quad \Gamma \vdash y : t}{\Gamma \vdash x \mapsto y : s \mathbin{\dot\times} t}\ \mapsto_V$$

Rules for E

$$\frac{\Gamma \vdash x : t \quad \Gamma \vdash E : t}{\Gamma \vdash x \doteq E : S_\tau}\ \doteq \qquad \frac{\Gamma \vdash x \doteq E : S_\tau \quad \Gamma \vdash F : t}{\Gamma \vdash [x \doteq E]F : t}\ subst_E$$

$$\frac{\Gamma \vdash x : s \quad \Gamma \vdash y : t}{\Gamma \vdash x \mapsto\ y : s \mathbin{\dot\times} t}\ \mapsto_E \qquad \frac{\Gamma \vdash s : \dot{\mathbb{P}}\,(t)}{\Gamma \vdash\ \dot{choice}\,(s) : t}\ \dot{choice}$$

where S_τ is the type of substitutions.

Rules for S

$$\frac{\Gamma \vdash S : \dot{\mathbb{P}}\,(s) \quad \Gamma \vdash T : \dot{\mathbb{P}}\,(t)}{\Gamma \vdash S \mathbin{\dot\times} T : \dot{\mathbb{P}}\,(s \mathbin{\dot\times} t)}\ \dot\times \qquad \frac{\Gamma \vdash E : \dot{\mathbb{P}}\,(s)}{\Gamma \vdash \dot{\mathbb{P}}\,(E) : \dot{\mathbb{P}}\,(\dot{\mathbb{P}}\,(s))}\ \dot{\mathbb{P}}$$

$$\frac{x : s, \Gamma \vdash P : P_\tau}{\Gamma \vdash \{x \mid P\} : \dot{\mathbb{P}}\,(s)}\ \{\dot|\} \qquad \frac{}{\Gamma \vdash B\dot{I}G : \dot{\mathbb{P}}\,(B\dot{I}G)}\ B\dot{I}G$$

Rules for P

$$\frac{\Gamma \vdash P : P_\tau \quad \Gamma \vdash Q : P_\tau}{\Gamma \vdash P \mathbin{\dot\wedge} Q : P_\tau}\ \dot\wedge \qquad \frac{\Gamma \vdash P : P_\tau \quad \Gamma \vdash Q : P_\tau}{\Gamma \vdash P \mathbin{\dot\Rightarrow} Q : P_\tau}\ \dot\Rightarrow$$

$$\frac{\Gamma \vdash P : P_\tau}{\Gamma \vdash\ \dot\neg P : P_\tau}\ \dot\neg \qquad \frac{\Gamma \vdash x : t \quad \Gamma \vdash P : P_\tau}{\Gamma \vdash\ \dot\forall x.P : P_\tau}\ \dot\forall$$

$$\frac{\Gamma \vdash x \doteq E : S_\tau \quad \Gamma \vdash P : P_\tau}{\Gamma \vdash [x \doteq E]P : P_\tau}\ subst_P \qquad \frac{\Gamma \vdash E : t \quad \Gamma \vdash F : t}{\Gamma \vdash E \doteq F : P_\tau}\ \doteq$$

$$\frac{\Gamma \vdash E : t \quad \Gamma \vdash S : \dot{\mathbb{P}}\,(t)}{\Gamma \vdash E \mathbin{\dot\in} S : P_\tau}\ \dot\in$$

where P_τ is the type of predicates.

Fig. 2. Type Inference Rules

$$\mathcal{L}(\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P) = \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{L}(P)$$
$$\mathcal{L}(P \mathbin{\dot\wedge} Q) = \mathcal{L}(P) \mathbin{\dot\wedge} (P \mathbin{\dot\Rightarrow} \mathcal{L}(Q)) \qquad \mathcal{L}(P \mathbin{\dot\Rightarrow} Q) = \mathcal{L}(P) \mathbin{\dot\wedge} (P \mathbin{\dot\Rightarrow} \mathcal{L}(Q))$$
$$\mathcal{L}(\dot\neg P) = \mathcal{L}(P) \qquad \mathcal{L}(\dot\forall x.P) = \dot\forall x.\mathcal{L}(P)$$

The atomic rules essentially aim to deal with applications of functions and handle atomic predicates, i.e. every predicate other than those considered above. The atomic rules are defined as follows:

$$\mathcal{L}(A) = \dot{true}$$
$$\mathcal{L}(A_{f(E)}) = \dot\exists(s \mathrel{\dot\mapsto} t).(f \mathbin{\dot\in} s \mathbin{\dot+\!\!\dot\mapsto} t \mathbin{\dot\wedge} E \mathbin{\dot\in} \dot{dom}(f)) \mathbin{\dot\wedge} \dot\forall y.(y \doteq f(E) \mathbin{\dot\Rightarrow} \mathcal{L}(A_y))$$
where $y \notin \mathcal{M} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{y\}$
$$\mathcal{L}(A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}) = \dot\forall x.(\mathcal{L}(x \mathbin{\dot\in} S) \mathbin{\dot\wedge} (x \mathbin{\dot\in} S \mathbin{\dot\Rightarrow} \mathcal{L}(P))) \mathbin{\dot\wedge}$$
$$\dot\forall y.(y \doteq \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \mathbin{\dot\Rightarrow} \mathcal{L}(A_y))$$
where $x, y \notin \mathcal{M} \cup \mathcal{B}$, and $\mathcal{B} \leftarrow \mathcal{B} \cup \{x, y\}$

in which an atomic predicate may be of the following form:

1. A, where $f(E), \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \notin A$;
2. $A_{f(E)}$, where $f(E) \in A_{f(E)}$, but $g(F), \{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \notin f, E$;
3. $A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}$, where $\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\} \in A_{\{x \mid x \mathbin{\dot\in} S \mathbin{\dot\wedge} P\}}$.

The binding rules, i.e. the rules for atomic predicates of the form (3) must be applied first, before the rule for atomic predicates of the form (2), in order to avoid to eliminate applications of functions under binders.

Compared to [2], our approach relaxes the restrictions over the super-set S in the atomic predicate for comprehension sets, which implies to add the recursive call $\mathcal{L}(x \mathbin{\dot\in} S)$ in the corresponding rule. In addition, we are also able to deal with substitutions (for expressions and predicates), as well as lambda-expressions (we do not provide the corresponding rules here in order to simplify our presentation).

Once this filter has been applied to a reified rule of the form $\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P$, the well-definedness lemma to be proved is generated as follows:

$$\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{S}_{\mathcal{B}, \mathcal{B} \cup \mathcal{M}_\Gamma}(H \mathrel{\dot\vdash} \mathcal{L}(P))$$

where $\mathcal{M}_S \leftarrow \mathcal{M}_S \cup \mathcal{M}_\Gamma$.

3.6 Rule Verification

Once the proof obligations related to well-definedness have been extracted from the rule by means of a first syntactical filter, it is possible to apply another filter to the rule, which eliminates the conditional definitions unconditionally and produces an equivalent rule simpler to prove. This new filter, which is introduced in [2], is called $\mathcal{E}$, and is defined as a function over reified rules, which unconditionally eliminates all the applications of functions. Considering the $\mathcal{L}$ filter

seen previously, it can be shown that $\mathcal{L}(P) \dot{\Rightarrow} (P \dot{\Leftrightarrow} \mathcal{E}(P))$. In the same way as for the $\mathcal{L}$ filter, the computation rules of $\mathcal{E}$ are split into two sets of rules: decomposition and atomic rules. The decomposition rules are the following:

$$\mathcal{E}(\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P) = \forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{E}(P) \qquad \mathcal{E}(P \mathbin{\dot{\wedge}} Q) = \mathcal{E}(P) \mathbin{\dot{\wedge}} \mathcal{E}(Q)$$
$$\mathcal{E}(P \dot{\Rightarrow} Q) = \mathcal{E}(P) \dot{\Rightarrow} \mathcal{E}(Q) \qquad \mathcal{E}(\dot{\neg}\, P) = \dot{\neg}\, \mathcal{E}(P) \qquad \mathcal{E}(\dot{\forall} x.P) = \dot{\forall} x.\mathcal{E}(P)$$

The atomic rules are defined as follows:

$$\mathcal{E}(A) = A$$
$$\mathcal{E}(A_{f(E)}) = \dot{\forall} y.((E, y) \mathbin{\dot{\in}} f \dot{\Rightarrow} \mathcal{E}(A_y)) \text{ where } y \notin \mathcal{M} \cup \mathcal{B}, \text{ and } \mathcal{B} \leftarrow \mathcal{B} \cup \{y\}$$
$$\mathcal{E}(A_{\{x \mid x \mathbin{\dot{\in}} S \mathbin{\dot{\wedge}} P\}}) = \dot{\forall} y.(y = \{x \mid x \mathbin{\dot{\in}} S \mathbin{\dot{\wedge}} \mathcal{E}(P)\} \dot{\Rightarrow} \mathcal{E}(A_y))$$
$$\text{where } y \notin \mathcal{M} \cup \mathcal{B}, \text{ and } \mathcal{B} \leftarrow \mathcal{B} \cup \{y\}$$

Once this filter has been applied to a reified rule of the form $\forall x \in \mathcal{M}, \mathcal{B}, \mathcal{N}.P$, the rule lemma to be proved is generated in the following way:

$$\forall x.\mathcal{M}, \mathcal{B}, \mathcal{N}.\mathcal{S}_{\mathcal{B}, \mathcal{B} \cup \mathcal{M}_\Gamma}(H \mathbin{\dot{\vdash}} \mathcal{E}(P))$$

where $\mathcal{M}_S \leftarrow \mathcal{M}_S \cup \mathcal{M}_\Gamma$ with $\mathcal{M}_\Gamma$ the set of the type variables of the typing context Γ, and where H is the reification of Γ.

3.7 Examples

In the following, we describe two examples of rule verification using BCARe.

Example 3 (Verification of ForAllX.3). This rule is a deduction rule, and there is no need to verify that there is no variable capture. As there is no application function, only the typing and rule lemmas are generated as follows:

Lemma *type_ForAllX3* : **forall** $t : \dot{S}$, **forall** $A\ a : \dot{V}$,
$a \mathbin{\dot{\setminus}} (A, t) \rightarrow \dot{given}\ (t),\ A \mathbin{\dot{\subseteq}} t \mathbin{\dot{\vdash}}_\tau \dot{check}\ (A \doteq \dot{\emptyset} \dot{\Rightarrow} \dot{\forall}\, a.a \mathbin{\dot{\notin}} A)$.

Lemma *rule_ForAllX3* : **forall** $A\ t : \dot{S}$, **forall** $a : \dot{V}$,
$a \mathbin{\dot{\setminus}} (A, t) \rightarrow A \mathbin{\dot{\subseteq}} t \mathbin{\dot{\vdash}} (A \doteq \dot{\emptyset} \dot{\Rightarrow} \dot{\forall}\, a.a \mathbin{\dot{\notin}} A)$.

Example 4 (Verification of SimplifyRelDorXY.2). This rule is a rewrite rule, and we must verify that the guards correctly protect the free variables. As the elements of Q do not belong to $\{f, u, v, a\}$, the criterion (2) is verified. The three lemmas are then generated in the following way:

Lemma *type_SimplifyRelDorXY_2* :
forall $t_1\ t_2 : \dot{S}$, **forall** $a\ f\ u\ v : \dot{V}$,
$\dot{given}\ (t_1),\ \dot{given}\ (t_2),\ u \mathbin{\dot{\in}} \dot{\mathbb{P}}\,(t_2),\ v \mathbin{\dot{\in}} \dot{\mathbb{P}}\,(t_1),\ f \mathbin{\dot{\in}} \dot{\mathbb{P}}\,(t_2 \mathbin{\dot{\times}} t_1),\ a \mathbin{\dot{\in}} t_2 \mathbin{\dot{\vdash}}_\tau$
$\dot{check}\ (f \mathbin{\dot{\in}} u \mathbin{\dot{\rightarrow}} v \mathbin{\dot{\wedge}} a \mathbin{\dot{\in}} \dot{dom}(f) \dot{\Rightarrow} \{a\} \mathbin{\dot{\lhd}} f \doteq \{a \mapsto f(a)\})$.

Lemma $wdef\_SimplifyRelDorXY\_2$:
$\mathbf{forall}\ f\ t_1\ t_2\ u\ v : \dot{S},\ \mathbf{forall}\ a : \dot{E},\ \mathbf{forall}\ s\ t : \dot{V},$
$s \mathbin{\dot{\setminus}} (t, f, u, v, a, t_1, t_2) \rightarrow t \mathbin{\dot{\setminus}} (s, f, u, v, a, t_1, t_2) \rightarrow$
$u \dot{\in} \dot{\mathbb{P}}(t_2),\ v \dot{\in} \dot{\mathbb{P}}(t_1),\ f \dot{\in} \dot{\mathbb{P}}(t_2 \dot{\times} t_1),\ a \dot{\in} t_2 \ \dot{\vdash}$
$f \dot{\in} u \dot{\nrightarrow} v \dot{\wedge} a \dot{\in} \dot{dom}(f) \dot{\Rightarrow} \dot{\exists}(s \dot{\mapsto} t).(f \dot{\in} s \dot{\nrightarrow} t \dot{\wedge} a \dot{\in} \dot{dom}(f))\,.$

Lemma $rule\_SimplifyRelDorXY\_2$:
$\mathbf{forall}\ f\ t_1\ t_2\ u\ v : \dot{S},\ \mathbf{forall}\ a : \dot{E},\ \mathbf{forall}\ y : \dot{V},$
$y \mathbin{\dot{\setminus}} (f, u, v, t_1, t_2, a) \rightarrow u \dot{\in} \dot{\mathbb{P}}(t_2),\ v \dot{\in} \dot{\mathbb{P}}(t_1),\ f \dot{\in} \dot{\mathbb{P}}(t_2 \dot{\times} t_1),\ a \dot{\in} t_2 \ \dot{\vdash}$
$f \dot{\in} u \dot{\nrightarrow} v \dot{\wedge} a \dot{\in} \dot{dom}(f) \dot{\Rightarrow} \dot{\forall} y.((a, y) \dot{\in} f \dot{\Rightarrow} \{a\} \dot{\lhd} f \doteq \{a \dot{\mapsto} y\})\,.$

4 Automated Verification of Proof Rules

In this section, we discuss some solutions that we have provided to automate the verification of proof rules in the framework of the BCARe environment. In particular, the several solutions aim to automatically prove the different lemmas generated in Coq from proof rules by BCARe. To do so, we have developed a set of tactics using the $\mathcal{L}_{\text{tac}}$ tactic language of Coq [11]. In the specific case of the rule lemma, we have also considered an alternative approach based on an external ATP called Zenon [5]. Both approaches are able to deal with rules involving all the set operators defined before the functional abstraction (introduction of anonymous functions) in the B-Book [1].

4.1 Verification Using $\mathcal{L}_{\text{tac}}$

To deal with the different lemmas generated in Coq from proof rules by the BCARe environment (see Section 3), a set of tactics has been developed using the $\mathcal{L}_{\text{tac}}$ tactic language of Coq [11]. Regarding the proof of the typing lemma, we have designed a correct and complete tactic (the B typechecking is decidable), which essentially performs pattern-matching over the goal in order to select the appropriate B typing rules. As for the proof of well-definedness lemma, we have written another tactic, which is able to manage only specific cases. This tactic mostly looks for instantiations which allow us to complete the proof using a direct propositional combination of the current hypotheses. Finally, the proof of the rule lemma is handled by means of a tactic which relies on a naive and incomplete heuristic, even though it succeeds in proving about 200 derived rules of the B-Book. This heuristic mainly consists in only considering Skolem symbols when instantiating (no unification is performed), while right contraction is never used. In addition, this tactic has also some efficiency issues in some cases that can be observed in Subsection 4.3. To palliate these several drawbacks, we have developed an alternative approach based on the use of an external ATP, able to provide a more powerful and efficient proof search procedure, and able to be easily interfaced with Coq (i.e. producing proof traces that can be exploited).

4.2 Verification Using Zenon

As an alternative approach to $\mathcal{L}_{\text{tac}}$ tactics, we have developed an interface with the Zenon ATP [5], in order to prove the rule lemmas in particular. One of the main difficulties when using an external ATP is to bring together the B set theory and the ATP logic. As seen previously, the B set theory [1] is actually based on a simplification of classical set theory. As for Zenon, it relies on the classical first order logic with equality (using the tableau method as proof search), and does not deal explicitly with the set theory. The idea consists in normalizing the formula to be proved (unfolding definitions), in order to obtain a first order logic formula containing only the "$\dot{\in}$" (reified) set operator. This formula is then syntactically interpreted within the ATP logic, in which the "$\dot{\in}$" operator is considered as a regular uninterpreted predicate symbol.

Another difficulty is to ensure the correctness of the external deduction. We have adopted a skeptical approach by building B proofs from the proofs produced by the ATP. In this way, it is possible to check the validity of these proofs that have been automatically found. However, this requires the ATP proof traces to be comprehensible enough so as to allow us to reconstruct proofs. This is the case of Zenon, which produces several proof traces at different levels. In particular, it produces Coq proofs, which can be used to build proofs within the BCoq embedding of BCARe (see Section 3). To do so, the Coq proofs generated by Zenon have to be (re)reified. This translation is syntactical and relies on an embedding of each tactic occurring in the Coq proofs produced by Zenon.

4.3 Benchmarks

In the following, we present the results of our implementation using Zenon on several examples of proof rules. This implementation actually consists of a Coq tactic written in OCaml. We also compare these results to those obtained using the corresponding $\mathcal{L}_{\text{tac}}$ tactic. To realize these benchmarks, we have tested both tactics on derived rules of the B-Book, as well as on several added rules coming from Atelier B and the database maintained by Siemens SAS I MO.

Regarding derived rules, we have considered about 200 rules and the results of the tests (run on an Intel Pentium D 3.40GHz/4GB computer) are summarized in the graph of Figure 3, where a point represents the test of a proof rule and where the x-axis and y-axis respectively correspond to the $\mathcal{L}_{\text{tac}}$ and Zenon proof times (expressed in seconds). In this graph, we only consider derived rules for which the proof times for $\mathcal{L}_{\text{tac}}$ and Zenon are less than 30s (this corresponds to about 66% of the tested rules). We can see that Zenon is faster than the $\mathcal{L}_{\text{tac}}$ tactic for the most part of the tested rules (for 71% of these rules, more precisely). Furthermore, over the 200 rules for which Zenon succeeds in finding a proof, 15 rules cannot be proved using the $\mathcal{L}_{\text{tac}}$ tactic. For the sake of scalability, we have also tested our tactics on added rules coming from Atelier B and the database maintained by Siemens SAS I MO. We have selected 1279 rules (over a total of 5039 rules) within the scope of both tactics. For these rules, Zenon can prove 813 rules (64%), whereas the $\mathcal{L}_{\text{tac}}$ tactic manages to prove 498 rules (39%).

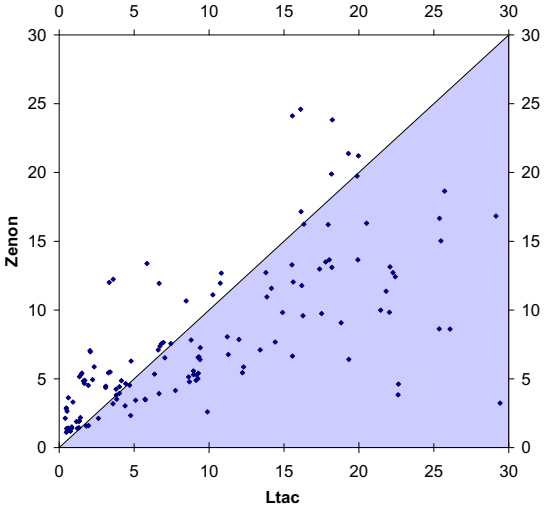


Fig. 3. Proof Times of Rule Lemmas using Zenon and $\mathcal{L}_{\text{tac}}$

In addition, the larger the $\mathcal{L}_{\text{tac}}$ proof time is, the larger is the number of rules for which Zenon is faster than $\mathcal{L}_{\text{tac}}$. These several experimental results tend to show that the use of Zenon is an approach which is not only more satisfactory than that of $\mathcal{L}_{\text{tac}}$, but also very promising in terms of scalability.

5 Conclusion

We have proposed a formal and mechanized framework which allows us to verify proof rules of the B method, and which is able to use an external automated theorem prover called Zenon. This framework relies on the BCARe set of tools, developed by Siemens SAS I MO, which provides a deep embedding of the B theory within the logic of the Coq proof assistant and allows us to automatically generate the required properties to be checked for a given proof rule. Currently, this tool chain is able to automatically verify about 200 derived rules of the B-Book, as well as 800 added rules coming from Atelier B and the rule database maintained by Siemens SAS I MO.

As future work, we first aim to completely verify the derived rules of the B-Book. The BCARe environment is already able to deal with all these derived rules, but the automated verification part (using Zenon) has to be adapted. In particular, this part has to be extended to manage proofs of properties involving applications of functions, substitutions, arithmetics, induction, and sequences. It seems clear that all the proofs will not be able to be automated, and our goal consists in automating at least a large part of them and characterizing the lack of automation for the other proofs. To palliate this potential lack of automation, we could consider alternative ATPs (other than Zenon) or SMT solvers, which might be more appropriate for some specific properties. In this case, we should

develop a verification platform able to use several provers and solvers. Once these derived rules have been verified, we plan to deal with the rest of the added rules of Atelier B (about 1,400 rules), and thereafter the rest of those of the database developed by Siemens SAS I MO (about 3,100 rules). If the latter focuses on the development of applications, the former consists in certifying Atelier B as a tool used in a safety-critical and high-integrity chain of production.

References

1. Abrial, J.-R.: The B-Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge, UK (1996) ISBN 0521496195
2. Abrial, J.-R., Mussat, L.: On Using Conditional Definitions in Formal Theories. In: Bert, D., et al. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 242–269. Springer, Heidelberg (2002)
3. Berkani, K., Dubois, C., Faivre, A., Falampin, J.: Validation des règles de base de l'Atelier B. Technique et Science Informatiques (TSI) 23(7), 855–878 (2004)
4. Bodeveix, J.-P., Filali, M., Muñoz, C.: A Formalization of the B-Method in Coq and PVS. B Users Group Meeting, Toulouse, France (September 1999)
5. 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)
6. Chartier, P.: Formalisation of B in Isabelle/HOL. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 66–82. Springer, Heidelberg (1998)
7. Cirstea, H., Kirchner, C.: Using Rewriting and Strategies for Describing the B Predicate Prover. In: Strategies in Automated Deduction, Lindau, Germany, pp. 25–36 (July 1998)
8. ClearSy. Atelier B 4.0 (February 2009), http://www.atelierb.eu/
9. Couchot, J.-F., Dadeau, F., Déharbe, D., Giorgetti, A., Ranise, S.: Proving and Debugging Set-Based Specifications. In: Workshop on Formal Methods, Campina Grande, Brazil. ENTCS, vol. 95, pp. 189–208. Elsevier (October 2003)
10. Déharbe, D.: Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 217–230. Springer, Heidelberg (2010)
11. Delahaye, D.: A Tactic Language for the System Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)
12. Jaeger, É., Dubois, C.: Why would you trust B? In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 288–302. Springer, Heidelberg (2007)
13. Le Lay, É.: Automatiser la validation des règles. Master's thesis, INSA (Rennes), Siemens SAS I MO (September 2008)
14. Mikhailov, L., Butler, M.: An Approach to Combining B and Alloy. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 140–161. Springer, Heidelberg (2002)
15. The Coq Development Team. Coq, version 8.3. INRIA (October 2010), http://coq.inria.fr/

Hybrid Specification of Reactive Systems: An Institutional Approach*

Alexandre Madeira[1,2,3], José M. Faria[1], Manuel A. Martins[3], and Luís S. Barbosa[2]

[1] Critical Software S.A., Portugal
[2] Department of Informatics, Minho University
[3] Department of Mathematics, University of Aveiro

Abstract. This paper introduces a rigorous methodology for requirements specification of systems that react to external stimulus by evolving through different operational modes. In each mode different functionalities are provided. Starting from a classical state-machine specification, the envisaged methodology interprets each state as a different mode of operation endowed with an algebraic specification of the corresponding functionality. Specifications are given in an expressive variant of hybrid logic which is, at a later stage, translated into first-order logic to bring into scene suitable tool support. The paper's main contribution is to provide rigorous foundations for the method, framing specification logics as institutions and the translation process as a comorphism between them.

1 Introduction

Motivation. The successful development and deployment of safety-critical, reactive systems, from the early concept and system definition phases, down to implementation and validation, poses a number of challenges that engineers must overcome. From the outset, there are two basic approaches to formally capture requirements for this sort of software: one emphasizes *behaviour* and its evolution; the other focus on *data* and their transformations.

Within the first paradigm, reactive systems are typically specified through (some variant of) *state-machines*. Such models capture system's evolution in terms of event occurrence and its impact in the system internal state configuration. Automata theory, and its more recent, abstract rendering in coalgebraic terms, provide a suitable formalism for both specification and analysis. Crucial notions of bisimulation, minimization and invariant, among others, play a fundamental, long established role in this framework. In the dual, data-oriented

* This research was partially supported by FCT (the Portuguese Foundation for Science and Technology) under contract PTDC/EIA-CCO/ 108302/2008 (the MONDRIAN project) and CIDMA at University of Aveiro. A. Madeira and J. M. Faria worked under contracts SFRH/BDE/ 33650/2009 and SFRH / BDE / 51049 / 2010, two PhD grants jointly supported by FCT and *Critical Software S.A., Portugal*. M. Martins was further supported by the project *Nociones de Completud*, reference FFI2009-09345 (Spain).

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 269–285, 2011.

approach the system's functionality is given in terms of input-output relations modeling operations on *data*. A specification is a theory in a suitable logic, expressed over a signature, which captures its syntactic interface. Its semantics is a class of concrete algebras acting as models for the specified theory [5,18].

In practice, however, both approaches are interconnected: the functionality offered by the system, at each moment, may depend on the stage of its evolution. Such is typically the case of complex, reactive, reconfigurable software.

This paper explores such a interconnection. Starting from a classical state-machine specification, the methodology illustrated in the sequel goes a step further: different states are interpreted as different *modes* of operation and each of them is equipped with an algebraic specification (over the system's interface) of the corresponding functionality. Technically, specifications become *structured* state-machines, states denoting *algebras*, rather than *sets*.

The following paragraph sums up the envisaged approach. It should be remarked this has been developed in a concrete, industrial context — that of a leading, portuguese IT company, whose mission includes the production of formally certified software for critical systems. Such a context makes effective, but sound tool support a must. As discussed in the sequel, rigorous foundations also (may) lead to fulfill this objective.

Approach and paper outline. The approach proposed in the paper is sketched in Figure 1. The upper plane sums up the envisaged methodology. The block on the left hand side represents the specification framework, structured in two stages, as explained below. The annotation on top — *Hybrid logic* — states the underlying logic. The block on the right concerns verification and analysis of hybrid specifications suitably translated to first order logic (FOL). The translation itself is depicted as a *comorphism* between the two logic systems in presence: hybrid logic, chosen for its expressive power, first order, to benefit from existent verification support. Hybrid logic [2] plays a fundamental role here given its ability to make explicit references, through special symbols called *nominals*, to specific states within a model.

The lower plane of Figure 1 refers to the methodology foundations. Actually, a basic property to require from a specification formalism is its ability to be framed as an *institution* [7,4]. This is not a formal idiosyncrasy: institutions, as abstract, general representations of logical systems, provide modular structuring and parameterization mechanisms which are defined 'once and for all' abstracting from the concrete particularities of the each specification logic [24]. Moreover, several current specification formalisms, notably, CafeOBJ [5], CASL [18] and HETS [20] were designed to take advantage of such a general framework.

Moreover institutions provide a systematic way to relate logics and transport results from one to another [17], which means that a theorem prover for the latter can be used to reason about specifications written in the former. This is achieved through a special class of maps between institutions, referred to as *comorphisms*, as depicted in Figure 1.

The rest of the paper is organized around two main sections: one on the *methodology* (sections 2) and another on *foundations* (section 3). Section 4

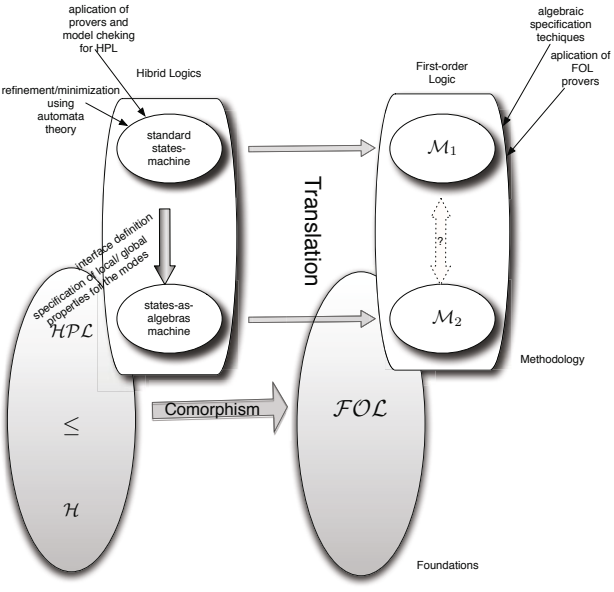


Fig. 1. The approach

discusses current work on suitable tool support based on HETS [20]. Section 5 concludes and provides a few pointers for future work. The reader may found the detailed proofs of all formal claims of the paper on the technical report [13].

2 A Specification Methodology

As stated above, the paper proposes a methodology to the specification and analysis of reactive systems which is intended to be effectively used in an industrial context. The methodology has the following stages, which will be detailed later in the paper:

I (I.1) Express the requirements in *hybrid propositinal logic* (HPL), identifying states and transitions to build a first state-machine; (I.2) Specify local properties as propositions; At this stage, traditional technics of state machine analysis/refinement may be applied, and available reasoning tools for HPL used (see Section 2.1).

II (II.1)Define the actual system's interface through the set of (external) services offered. Technically, this is supported by the definition of a (multi-sorted) first-order signature. (II.2) Express, whenever possible, the attributes of the first machine as functional properties over this signature.

III Translate both specifications into FOL, providing a common ground for testing and verification.

In the sequel the methodology is illustrated in a number of specification fragments of an *automatic cruise control* (ACC) system. The example, small but

self-contained, is taken from [9], where the overall requirements are summarized as follows:

> *"The mode class CruiseControl contains four modes, Off, Inactive, Cruise, and Override. At any given time, the system must be in one of these modes. Turning the ignition on causes the system to leave Off mode and enter Inactive mode, while turning the cruise control level to const when the brake is off and the engine running causes the system to enter Cruise mode. (...) Once cruise control has been invoked, the system uses the automobile's actual speed to determine whether to set the throttle to accelerate or decelerate the automobile, or to maintain the current speed (...)To override cruise control (i.e., enter Override), the driver turns the lever to off or applies the brake".*

2.1 Hybrid Specifications (Stage I)

The requirements for the cruise control system example can be captured by the state machine depicted in Figure 2. This section introduces its specification in propositional hybrid logic (HPL). Such a presentation has the advantage of being compact, unambiguous and closer to the input format of typical verification engines.

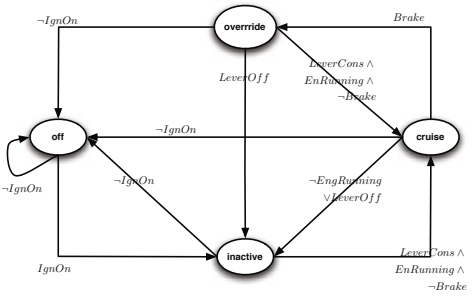


Fig. 2. State-machine of the system

The set of HPL formulas is defined by the following grammar:

$$\varphi, \psi ::= p \mid i \mid \neg\varphi \mid [\lambda]\varphi \mid @_i\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \tag{1}$$

where λ ranges over a set Λ of modal operators. Models of this logic are state-machines with an additional function state : Nom $\rightarrow S$ which assigns to each nominal a state. This allows explicit reference to particular states in a specification. Thus, models are tuples $\mathcal{P} = \langle S, \text{state}, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}\rangle$ where S is a set of states, $R_\lambda \subseteq S \times S$ is the accessibility relation associated to the modality λ and P_s : Prop $\rightarrow \{\top, \bot\}$ is the function that assigns to the propositions on the state $s \in S$. The satisfaction relation is defined as in standard modal logic (e.g. $\mathcal{P} \models^s p$ iff $P_s(p) = \top$; $\mathcal{P} \models^s [\lambda]\varphi$ iff $\mathcal{P} \models^{s'} \varphi$ for any s' such that $(s, s') \in R_\lambda$) adding the following cases related to nominals:

- $\mathcal{P} \models^s @_i\varphi$ iff $\mathcal{P} \models^{\text{state}(i)} \varphi$;
- $\mathcal{P} \models^s i$ iff state$(i) = s$.

Moreover, we abbreviate formulas $\neg[\lambda]\neg\varphi$ and $\langle\lambda\rangle\varphi \wedge [\lambda]\varphi$ to $\langle\lambda\rangle$ and $\langle\lambda\rangle^\circ\varphi$, respectively.

For the running example, a modality $\{next\}$ is introduced to denote the state-machine accessibility relation. Nominals in $\{off, inactive, override, cruise\}$ correspond to the operation modes mentioned in the requirements. Finally, a set of propositions is considered — one for each label in Figure 2. With such signature, transitions are specified as follows:

- $(T_1)@_{off}(\ IgnOn \Rightarrow \langle next\rangle^\circ inactive\ )$
- $(T_2)\neg\ IgnOn \Rightarrow \langle next\rangle^\circ off$
- $(T_3)@_{inactive}(LeverCons \wedge IgnOn \wedge \neg\ Brake \Rightarrow \langle next\rangle^\circ cruise)$
- $(T_4)@_{cruise}(\neg\ EngRunning \vee LeverOff \Rightarrow \langle next\rangle^\circ inactive)$
- $(T_5)@_{cruise}(\ Brake \Rightarrow \langle next\rangle^\circ override)$
- $(T_6)@_{override}(LeverCons \wedge IgnOn) \wedge EngRunning \wedge \neg\ Brake \Rightarrow \langle next\rangle^\circ cruise)$

Local properties can also be expressed resorting to the satisfaction operator $@_i$, for each nominal i, to reference the corresponding state. For instance, the requirement that the engine controls speed decelerating the car if the *speed is high* and maintaining it when it is considered *adequate* is modelled by

- $(L^1_{cruise})@_{cruise}(\ IgnOn \wedge EngRunning \wedge HighSpeed \Rightarrow decel)$
- $(L^2_{cruise})@_{cruise}(IgnOn \wedge EngRunning \wedge AdmissibleSpeed \Rightarrow mantain)$

Finally, admissibility properties, concerning propositions, are also captured. For instance, the fact that *the lever cannot be switched in more than one position at each time*, and similarly for the acceleration and speed modes, is expressed as

- $(A_1)\ LeverOff \Leftrightarrow \neg\ LeverCons$
- ...
- $(A_4)\ HighSpeed \Rightarrow \neg\ CruiseSpeed \wedge \neg\ LowSpeed$

2.2 States-as-Algebras Models (Stage II)

The logic. The second stage in the methodology equips each state of the underlying state-machine with an *algebra*, more precisely a first-order structure, to model its local functionality. Therefore, hybrid structures are enriched with a family of first-order structures indexed by the set of states, i.e., they become structures

$$\mathcal{M} = \langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (A_s)_{s\in S}\rangle$$

where first-order structures in the family $(A_s)_{s\in S}$ are defined over the same signature and universe, say A. Each A_s models the system's behaviour at state $s \in S$.

Definition 1. *Let Σ a first-order signature and X a set of variables for it,* Nom, Prop *and Λ three disjoint sets of nominals, propositions and modalities respectively. The set of* hybrid equational formulas *is defined by the following grammar:*

$$\varphi, \psi ::= p \mid i \mid t \approx t' \mid P(\bar{t}) \mid \neg\varphi \mid \varphi \star \psi \mid [\lambda]\varphi \mid @_i\varphi \mid \forall x\varphi \qquad (2)$$

where $\star \in \{\vee, \wedge, \Rightarrow\}$, p is a proposition, i is a nominal, $t \approx t'$ is a Σ-equation over X, $x \in X$, P is a Σ-predicate of type $s_1, \ldots, s_n$ where $\bar{t} := t_1, \ldots, t_n$ and $t_i \in (T_\Sigma(X))_{s_i}$.

An assignment for $M = \langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (A_s)_{s\in S}\rangle$ consists of a (sorted-set) function $g : X \to A$, where A is the carrier set of the first-order structures of $\mathcal{M}$ and X is a set of variables. We write $g \sim^x g'$ if for any variable $y \neq x$, $g(y) = g'(y)$. Note that the assignment $g : X \to A$ induces an S-family of assignments $g^s : T_\Sigma(X) \to A$ defined, for any $x \in X$, by $g^s(x) = g(x)$ and, for each term $f(t_1, \ldots, t_n)$, by $g^s(f(t_1, \ldots, t_n)) = f^{A_s}(g^s(t_1), \ldots, g^s(t_n))$.

Definition 2. *Let $\mathcal{M} = \langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (A_s)_{s\in S}\rangle$ be an hybrid structure. For any assignment $g : X \to A$, the* satisfaction relation *is recursively defined as follows:*

- $\mathcal{M}, g \models^s i$ *if* $state(i) = s$;
- $\mathcal{M}, g \models^s p$ *if* $P_s(p) = \top$;
- $\mathcal{M}, g \models^s t \approx t'$ *if* $A_s \models t \approx t'[g]$ *i.e., if* $g^s(t) = g^s(t')$;
- $\mathcal{M}, g \models^s Q(t_1, \ldots, t_n)$ *if* $A_s \models Q(t_1, \ldots, t_n)[g]$, *i.e., if* $Q^{A_s}(g^s(t_1), \ldots, g^s(t_n))$;
- $\mathcal{M}, g \models^s \rho \vee \rho'$ *if* $\mathcal{M} \models^s \rho$ *or* $\mathcal{M} \models^s \rho'$; *and similarly for the remaining boolean connectives;*
- $\mathcal{M}, g \models^s \forall x\rho$ *if, for any assignment* $g' : X \to A$, *if* $g \sim^x g'$, *one has* $\mathcal{M}, g' \models^s \rho$;
- $\mathcal{M}, g \models^s [\lambda]\rho$ *if, for any* $s' \in S$ *such that* $(s, s') \in R_\lambda$, *one has* $\mathcal{M} \models^{s'} \rho$.

We write $\mathcal{M} \models^s \rho$ when for any assignment $g : X \to A$, $\mathcal{M}, g \models^s \rho$ and $\mathcal{M}, g \models \rho$ when for any $s \in S$, $\mathcal{M}, g \models^s \rho$.

In order to model the system's functionality, as provided by the car artifact, we resort to a classical algebraic specification. This entails the need for introducing data types able to support the envisaged notions of *time*, *speed* and *acceleration*. In the running example integer numbers, with the usual operations and predicates $\{+, \leq, \geq, <, >\}$, can do the job.

spec TimeSort =Int **with sort** $Int \mapsto time$, **ops** $0 \mapsto init$, $suc \mapsto after$ **end**
spec SpeedSort =Int **with sort** $Int \mapsto speed$ **end**
spec AcellSort =Int **with sort** $Int \mapsto accel$ **end**

Thus, the operation *Pedal* models the accelerations applied by the driver at each moment. On the other hand, *Automatic* captures accelerations applied on the engine by the ACC, and *CurrentSpeed* records the current speed. Finally, constant *MaxCruiseSpeed* represents the maximum speed allowed on the ACC mode:

spec ACCSign =
TimeSort **and** SpeedSort **and** AcellSort
then ops $Pedal : time \to accel$;
$Automatic : time \to accel$;
$Speed : speed \times accel \to speed$;
$CurrentSpeed : time \to speed$;
$MaxCruiseSpeed : speed$

There are properties that globally hold, in all the configurations of the system. For instance,

$\forall\ s : speed;\ a : accel;\ t : time$
- $(G_1)\ Speed(s,\ a) \geq 0$
- $(G_2)\ CurrentSpeed(t) = 0 \wedge Pedal(t) \geq 0 \Rightarrow CurrentSpeed(after(t)) \geq 0$
- $(G_3)\ Pedal(t) \geq 0 \Leftrightarrow CurrentSpeed(t) < CurrentSpeed(after(t))$
- $(G_4)\ Speed(s,\ a) = s \Leftrightarrow a = 0$
- $(G_5)\ CurrentSpeed(after(t)) = Speed(CurrentSpeed(t), Pedal(t))$

Local properties. Differently from the properties above, local requirements hold only at particular configurations. Let us explore some of them. First, in state off, it is required that speed and acceleration are null and no other operations in the interface react:

$\forall\ t : time;\ s : speed;\ a : accel$
- $(L^1_{off})\ @_{off}\ CurrentSpeed(t) = 0$
- $(L^2_{off})\ @_{off}\ Speed(s,\ a) = 0$

In state *inactive*, the speed and acceleration depend on the accelerations automatically introduced in the system, i.e,

$\forall\ s : speed;\ a : accel$
- $(L^1_{inactive})\ @_{inactive}\ Speed(s,\ a) = s + a$

$\forall\ t{:}\ time;\ s : speed;\ a : accel$
- $(L^{1'}_{cruise})\ @_{cruise}[CurrentSpeed(t) > MaxCruiseSpeed \Rightarrow Automatic(after(t)) < 0]$
- $(L^{2'}_{cruise})\ @_{cruise}[CurrentSpeed(t) \leq MaxCruiseSpeed \Leftrightarrow Automatic(after(t)) = 0]$
- $(L^3_{cruise})@_{cruise}\ Speed(s,\ a) = s + a$
- $(L^4_{cruise})@_{cruise}\ Pedal(t) \geq 0 \Rightarrow Pedal(t) = Automatic(t)$

An interesting feature in this example is that properties local to states *override* and off do coincide. The system's behaviour on both states only differs in what concerns the definition of the allowed transitions. The latter are dealt as follows.

Transitions specification. To specify state transitions we simply resort to the state-machine built in Stage I, through axioms $(T_1), \ldots, (T_n)$ from Section 2.1. However, some propositions may now be expressed by means of algebraic properties of local states. For instance, we may replace (T_4) by

$\forall\ t{:}\ time;$
- $(T_{4'})@_{cruise}[CurrentSpeed(t) = 0 \Rightarrow \langle next\rangle^{\circ}(inactive \wedge CurrentSpeed(after(t)) = 0)]$
- $(T_{4''})@_{cruise}[LeverOff \Rightarrow \langle next\rangle^{\circ} inactive].$

Furthermore, the fact that when ACC is activated by transition T_6, the speed should to be maintained, is captured by

$\forall\ t{:}\ time;\ \forall\ s{:}\ speed$
- $(T_{6'})@_{override}[(LeverCons \wedge\ CurrentSpeed(t) = s\ \wedge\ s \geq 0) \Rightarrow \langle next\rangle^{\circ}(cruise \wedge CurrentSpeed(after(t)) = s)].$

3 Foundations

3.1 Going "institutional"

Dealing with the sort of specifications produced in Stages I and II above, entails the need for a *uniform* specification framework in which both equational properties of data types, modal properties of transitions and local properties of states

can be expressed and verified. The canonical way to do it is through the notion of an *institution* [7,4], as an abstract representation of a logical system, encompassing syntax, semantics and satisfaction. Let us recall the formal definition:

Definition 3 (Institution). *An* institution $\left(\mathrm{Sign}^{\mathcal{I}}, \mathrm{Sen}^{\mathcal{I}}, \mathrm{Mod}^{\mathcal{I}}, (\models^{\mathcal{I}}_{\Sigma})_{\Sigma \in |\mathrm{Sign}^{\mathcal{I}}|}\right)$ *consists of*

- *a category* $\mathrm{Sign}^{\mathcal{I}}$ *whose objects are called* signatures.
- *a functor* $\mathrm{Sen}^{\mathcal{I}} : \mathrm{Sign}^{\mathcal{I}} \to \mathsf{Set}$ *giving for each signature a set whose elements are called* sentences *over that signature.*
- *a functor* $\mathrm{Mod}^{\mathcal{I}} : (\mathrm{Sign}^{\mathcal{I}})^{op} \to \mathsf{CAT}$*, giving for each signature* Σ *a category whose objects are* Σ-models*, and whose arrows the corresponding* Σ-morphisms*, and*
- *a* satisfaction relation $\models^{\mathcal{I}}_{\Sigma} \subseteq |\mathrm{Mod}^{\mathcal{I}}(\Sigma)| \times \mathrm{Sen}^{\mathcal{I}}(\Sigma)$

such that for each morphism $\varphi : \Sigma \to \Sigma' \in \mathrm{Sign}^{\mathcal{I}}$ *, the satisfaction condition*

$$M' \models^{\mathcal{I}}_{\Sigma'} \mathrm{Sen}^{\mathcal{I}}(\varphi)(\rho) \; iff \; \mathrm{Mod}^{\mathcal{I}}(\varphi)(M') \models^{\mathcal{I}}_{\Sigma} \rho \tag{3}$$

holds for each $M' \in |Mod^{\mathcal{I}}(\Sigma')|$ *and* $\rho \in \mathrm{Sen}^{\mathcal{I}}(\Sigma)$*.*

A well known example of institution is the institution of first order logic, denoted in the sequel by $\mathcal{FOL}$ (see [4] for a detailed account). Institutions provide a suitable setting to do *abstract specification theory* [24], structuring any kind of specifications through combinators which are institution-independent, i.e. not tied to a specific logic system. In CASL [18], for example, such combinators allow the construction of basic specifications, by defining a signature and a set of sentences, the union of specifications, and the derivation and translation of specifications along signature morphisms. The use of this set of (abstract) combinators, makes possible to approach, in a uniform way and trough the same theory, systems expressed in completely different logics.

Therefore, our first aim concerning *foundations* is to prove that the proposed specification formalism may be framed on this big picture of institution theory. Let start by collecting the necessary ingredients to define a suitable institution $\mathcal{H}$.

Category $\mathrm{SIGN}^{\mathcal{H}}$: Signatures are tuples $\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle$ where Σ is a first-order logic signature, X is a set of first-order variables and Nom, Prop and Λ are (disjoint) sets of symbols of nominals, propositions and modalities. Signature morphisms

$$\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle \xrightarrow{\varphi} \langle(\Sigma', X'), \mathrm{Nom}', \mathrm{Prop}', \Lambda'\rangle$$

are tuples $\varphi = (\varphi_{\mathrm{Sig}}, \varphi_{\mathrm{Nom}}, \varphi_{\mathrm{Prop}}, \varphi_{\mathrm{MS}})$ where $\varphi_{\mathrm{Nom}} : \mathrm{Nom} \to \mathrm{Nom}'$, $\varphi_{\mathrm{Prop}} : \mathrm{Prop} \to \mathrm{Prop}'$ and $\varphi_{\mathrm{MS}} : \Lambda \to \Lambda'$ are functions and $\varphi_{\mathrm{Sig}} : (\Sigma, X) \to (\Sigma', X')$ is a morphism in $\mathcal{FOL}$, i.e., a tuple $\varphi_{\mathrm{Sig}} = (\varphi^{sort}_{\mathrm{Sig}}, \varphi^{op}_{\mathrm{Sig}}, \varphi^{pred}_{\mathrm{Sig}}, \varphi^{var}_{\mathrm{Sig}})$

- for any operation $f \in \Sigma_{s_1 \ldots s_n, s}$, $\varphi^{op}_{\mathrm{Sig}}(f) \in \Sigma'_{\varphi^{sort}_{\mathrm{Sig}}(s_1)\ldots\varphi^{sort}_{\mathrm{Sig}}(s_n), \varphi^{sort}_{\mathrm{Sig}}(s)}$;

- for any predicate $Q \in \Sigma_{s_1 \ldots s_n}$, $\varphi^{pred}_{\mathrm{Sig}}(Q) \in \Sigma'_{\varphi^{sort}_{\mathrm{Sig}}(s_1)\ldots\varphi^{sort}_{\mathrm{Sig}}(s_n)}$;
- for any variable $x \in X_s$, $\varphi^{var}_{\mathrm{Sig}}(x) \in X'_{\varphi^{sort}_{\mathrm{Sig}}(s)}$.

Functor $\mathrm{SEN}^{\mathcal{H}}$: This functor maps a signature $\Delta = \langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle$ into the set of hybrid sentences, i.e., on the subset of bonded-variables formulas of Definition 1, and a morphism

$$\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle \xrightarrow{\varphi} \langle(\Sigma', X'), \mathrm{Nom}', \mathrm{Prop}', \Lambda'\rangle$$

into the sentence translation

$$\mathrm{Sen}^{\mathcal{H}}(\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle) \xrightarrow{\mathrm{Sen}^{\mathcal{H}}(\varphi)} \mathrm{Sen}^{\mathcal{H}}(\langle(\Sigma', X'), \mathrm{Nom}', \mathrm{Prop}', \Lambda'\rangle)$$

recursively defined as follows

- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho) = \mathrm{Sen}^{\mathcal{FOL}}(\varphi_{\mathrm{Sig}})(\rho)$ for any $\rho \in \mathrm{Sen}^{\mathcal{FOL}}(\Sigma)$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(i) = \varphi_{\mathrm{Nom}}(i)$, $i \in \mathrm{Nom}$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(p) = \varphi_{\mathrm{Prop}}(p)$, $p \in \mathrm{Prop}$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(t \approx t') = \varphi^{term}(t) \approx \varphi^{term}(t)$, where $\varphi^{term} : T_{\Sigma}(X) \to T_{\Sigma'}(X')$ is a function recursively defined as follows
  - $\varphi^{term}(x) = \varphi^{var}_{\mathrm{Sig}}(x)$ for $x \in X$;
  - $\varphi^{term}(f(t_1, \ldots, t_n)) = \varphi^{op}_{\mathrm{Sig}}(f)(\varphi^{term}(t_1), \ldots, \varphi^{term}(t_n))$, for any $f \in \Sigma_{s_1 \ldots s_n, s}$, $t_i \in (T_{\Sigma}(X))_{s_i}$.
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(Q(t_1, \ldots, t_n)) = \varphi^{pred}_{\mathrm{Sig}}(Q)(\varphi^{term}(t_1), \ldots, \varphi^{term}(t_n))$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(\neg\rho) = \neg\mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho)$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho \odot \rho') = \mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho) \odot \mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho')$, $\odot \in \{\vee, \wedge, \rightarrow\}$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(@_i\rho) = @_{\varphi_{\mathrm{Nom}}(i)}\mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho)$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)([\lambda]\rho) = [\varphi_{\mathrm{MS}}(\lambda)]\mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho)$;
- $\mathrm{Sen}^{\mathcal{H}}(\varphi)(\forall x\rho) = \forall\varphi^{var}_{\mathrm{Sig}}(x)\mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho)$.

Functor $\mathrm{Mod}^{\mathcal{H}}$: This functor maps each signature $\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle$ to a category whose models are the hybrid structures $\mathcal{M} = \langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (A_s)_{s\in S}\rangle$ defined above. Morphisms between models $\langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (A_s)_{s\in S}\rangle$ and $\langle S', state', (R'_\lambda)_{\lambda\in\Lambda}, (P'_s)_{s\in S'}, (A'_s)_{s\in S'}\rangle$ consists of pairs (h_{st}, h_{mod}) such that

- h_{mod} is an S-family $\left(h_{mod_s} : A_s \to A'_{h_{st}(s)}\right)_{s\in S}$ of first-order structures morphisms;
- $P_s(p) = P'_{h_{st}(s)}(\varphi_{\mathrm{Prop}}(p))$;
- $h_{st} : S \to S'$ is a function such that
  - $(s, s') \in R_\lambda$ implies that $(h_{st}(s), h_{st}(s')) \in R'_\lambda$,
  - $state'(n) = h_{st}(state(n))$,

Functor $\mathrm{Mod}^{\mathcal{H}}$ maps each morphism

$$\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle \xrightarrow{\varphi} \langle(\Sigma', X'), \mathrm{Nom}', \mathrm{Prop}', \Lambda'\rangle$$

into the reduct functor

$$\mathrm{Mod}^{\mathcal{H}}(\langle(\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle) \xleftarrow{\mathrm{Mod}^{\mathcal{H}}(\varphi)} \mathrm{Mod}^{\mathcal{H}}(\langle(\Sigma', X'), \mathrm{Nom}', \mathrm{Prop}', \Lambda'\rangle)$$

that maps each $\langle(\Sigma', X'), \mathrm{Nom}', \mathrm{Prop}', \Lambda'\rangle$-model $\langle S', state', (R'_\lambda)_{\lambda\in\Lambda'}, (P'_s)_{s\in S'}, (A'_s)_{s\in S'}\rangle$ into the $\langle\Sigma, \mathrm{Nom}, \mathrm{Prop}, \Lambda\rangle$-model $\langle S, state, (R_\lambda)_{\lambda\in\Lambda}, (P_s)_{s\in S}, (A_s)_{s\in S}\rangle$ such that

- $S = S'$;
- $state(n) = state'(\varphi_{\mathrm{Nom}}(n))$ for any $n \in \mathrm{Nom}$;
- $R_\lambda = R'_{\varphi_{\mathrm{MS}}(\lambda)}$ for any $\lambda \in \Lambda$;
- $A_s = \mathrm{Mod}^{\mathcal{FOL}}(\varphi_{\mathrm{Sig}})(A'_s)$ for any $s \in S$, where $\mathrm{Mod}^{\mathcal{FOL}}(\varphi_{\mathrm{Sig}})$, the reduct notion on the institution of first-order logic, consists of the classical reduct notion on first-order structures;
- $P_s(p) = P'_s(\varphi_{\mathrm{Prop}}(p))$ for any $p \in \mathrm{Prop}$

Satisfaction $\models^{\mathcal{H}}$: Satisfaction is the restriction of Definition 2 to sentences.

Theorem 1. *Let $\Delta = ((\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda)$ and Δ' two $\mathcal{H}$-signatures and $\varphi : \Delta \to \Delta'$ a morphism of signatures. For any $\rho \in \mathrm{Sen}^{\mathcal{H}}(\Delta)$, $\mathcal{M}' = \langle S', state', R_{\Lambda'}, (P'_s)_{s\in S'}, (A'_s)_{s\in S'}\rangle \in |\mathrm{Mod}^{\mathcal{H}}(\Delta')|$, and $s \in S$,*

$$\mathrm{Mod}^{\mathcal{H}}(\varphi)(\mathcal{M}'), g \models^s \rho \text{ iff } \mathcal{M}', g' \models^s \mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho).$$

where, for any $x \in X$, $g(x) = g'(\varphi^{var}_{\mathrm{Sig}}(x))$.

Proof. The proof is done by induction on the structures of sentences.

The satisfaction condition for $\mathcal{H}$ follows from a well known fact, which states that satisfaction of a formula only depends on assignment of free variables. Actually,

Corollary 1 (Satisfaction condition). *Let $\Delta = ((\Sigma, X), \mathrm{Nom}, \mathrm{Prop}, \Lambda)$ and Δ' be two $\mathcal{H}$-signatures and $\varphi : \Delta \to \Delta'$ a morphism of signatures. For any $\rho \in \mathrm{Sen}^{\mathcal{H}}(\Delta)$, $\mathcal{M}' = \langle S', state', R_{\Lambda'}, (P'_s)_{s\in S'}, (A'_s)_{s\in S'}\rangle \in |\mathrm{Mod}^{\mathcal{H}}(\Delta')|$,*

$$\mathrm{Mod}^{\mathcal{H}}(\varphi)(\mathcal{M}') \models \rho \text{ iff } \mathcal{M}' \models \mathrm{Sen}^{\mathcal{H}}(\varphi)(\rho).$$

Therefore,

Corollary 2. $(\mathrm{Sign}^{\mathcal{H}}, \mathrm{Sen}^{\mathcal{H}}, \mathrm{Mod}^{\mathcal{H}}, \models^{\mathcal{H}})$ *is an institution.*

Finally, observe that models, language and satisfaction presented on Section 2.1 also constitute an institution. This institution is similarly defined, by forgetting the first-order signature from hybrid signatures, the state-family of first-order structures from models and the equations and quantifications from sentences. By obvious reasons, we call this the *institution of propositional hybrid logic* and write $\mathcal{HPL}$.

3.2 Translating to $\mathcal{FOL}$ (Stage III)

Stage III in the envisaged methodology was not discussed in section 2. Actually, from a methodological point of view it is rather straightforward: a translation of specifications to a well-known first order setting. Technically, however, this can be stated in a very precise way as a *comorphism*. Comorphims play, at the institutional level, the role of logical translations, lifting specifications expressed within different institutions to a common level [17]. Therefore, any tools, namely proof assistants, available at the target institution, can be borrowed by the source one. Formaly,

Definition 4 (Comorphism). *Given institutions* $\mathcal{I} = (\text{Sign}, \text{Sen}, \text{Mod}, \models)$ *and* $\mathcal{I}' = (\text{Sign}', \text{Sen}', \text{Mod}', \models')$ *a* comorphism $(\Phi, \alpha, \beta) : \mathcal{I} \to \mathcal{I}'$ *consists of*

1. *a functor* $\Phi : \text{Sign} \to \text{Sign}'$,
2. *a natural transformation* $\alpha : \text{Sen} \Rightarrow \Phi; \text{Sen}'$, *and*
3. *a natural transformation* $\beta : \Phi^{op}; \text{Mod}' \Rightarrow \text{Mod}$

such that the following satisfaction condition *holds*

$$M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(\rho) \quad \textit{iff} \quad \beta_\Sigma(M') \models_\Sigma \rho$$

for each signature $\Sigma \in |\text{Sign}|$, $\Phi(\Sigma)$*-model* M', *and* Σ*-sentence* ρ.

The comorphism is conservative *whenever, for each* Σ*-model* M *in* $\mathcal{I}$, *there exists a* $\Phi(\Sigma)$*-model* M' *in* $\mathcal{I}'$ *such that* $M = \beta_\Sigma(M')$.

Note that the comorfism conservativeness is necessary to borrow institutions proof support since it entails that $\Gamma \models_\Sigma \rho$ iff $\alpha_\Sigma(\Gamma) \models'_{\Phi(\Sigma)} \alpha(\rho)$, when we just have the left-right implication on its absence.

In this sub-section, we establish a comorphism from $\mathcal{H}$ into $\mathcal{FOL}$. The translation procedure is based on the addition of a special sort to represent states. Hence, in order to 'collapse' every local state algebra in a unique structure, the signature of all operations and predicates is enriched with an argument of this sort. Moreover, nominals are regarded as constants over ST, modalities as usual first-order relations and propositions as unary predicates over ST. For that we have a functor

$$\begin{array}{rcl} \Phi : \text{Sign}^{\mathcal{H}} & \longrightarrow & \text{Sign}^{\mathcal{FOL}} \\ \langle(\Sigma, X), \text{Nom}, \text{Prop}, \Lambda\rangle & \longmapsto & \big(\langle S^\Sigma + \{\text{ST}\}, \overline{F^\Sigma} + \overline{\text{Nom}}, \overline{P^\Sigma} + \overline{\text{Prop}} + \overline{\Lambda}\rangle, \bar{X}\big), \end{array}$$

where $\Sigma = (S^\Sigma, F^\Sigma, P^\Sigma)$ and

- $\overline{F_\Sigma} = \begin{cases} (\overline{F_\Sigma})_{\text{ST}w \to s} = (F_\Sigma)_{w \to s}, \text{ for any } s \in S_\Sigma, w \in S_\Sigma^* \\ \emptyset, \text{ for the other cases} \end{cases}$;
- $\overline{P_\Sigma} = \begin{cases} (\overline{P_\Sigma})_{\text{ST}w} = (P_\Sigma)_w, \text{ for any } w \in S_\Sigma^*; \\ \emptyset, \text{ for the other cases} \end{cases}$
- $\overline{\text{Nom}} = \{c_i : \ \to \text{ST} \mid i \in \text{Nom}\}$;
- $\overline{\text{Prop}} = \{\bar{p} : \text{ ST} \mid p \in \text{Prop}\}$;
- $\overline{\Lambda} = \{\lambda : \text{ ST}^n \mid \lambda \in \Lambda_n\}$.
- $\overline{X} = \begin{cases} \overline{X}_{sort} = X_{sort}, \text{ for any } sort \in S^\Sigma; \\ \overline{X}_{\text{ST}} = \{w, v\} \end{cases}$

Natural transformation $\beta : \Phi^{op}; \text{Mod}^{\mathcal{FOL}} \Rightarrow \text{Mod}^{\mathcal{H}}$ maps each first-order structure $(M; M_{\bar{F}} + M_{\overline{\text{Nom}}}; M_{\bar{P}} + M_{\overline{\text{Prop}}} + M_{\bar{\Lambda}}) \in \text{Mod}(\langle S_\Sigma + \{\text{ST}\}, \overline{F_\Sigma} + \overline{\text{Nom}}, \overline{P_\Sigma} + \overline{\text{Prop}} + \overline{\Lambda}\rangle)$ into

$$\langle S, state, R_\Lambda, (P_s)_{s \in S}, (A_s)_{s \in S}\rangle \xleftarrow{\beta_{b\langle F, \text{Nom}, \text{Prop}, \Lambda\rangle}} \langle M; M_{\bar{F}} + M_{\overline{\text{Nom}}}; M_{\bar{P}} + M_{\overline{\text{Prop}}} + M_{\bar{\Lambda}}\rangle \ ,$$

where for any $i \in \mathrm{Nom}$, $state(i) = c_i^M$, for any $\lambda \in \Lambda$, $R_\lambda = R_\lambda^M$. Moreover, A_s, $s \in S$ is a first-order structure whose carrier set is A^{S_Σ}; functions $f \in F^\Sigma_{s_1 \ldots s_n, s}$ and predicates $Q \in P^\Sigma_{s_1,\ldots,s_n}$ are defined for each $u_i \in U$, $i \leq n$, by $f^{A_s}(u_1, \ldots, u_n) = \bar{f}^M(s, u_1, \ldots, u_n)$ and $Q^{A_s}(u_1, \ldots, u_n) = \bar{P}^M(s, u_1, \ldots, u_n)$ respectively. The family $(P_s)_{s \in S}$, is defined, for each s as $P_s(p) = \top$ iff $\bar{p}^M(s)$.

Natural transformation $\alpha : \mathrm{Sen}^{\mathcal{H}} \Rightarrow \Phi; \mathrm{Sen}^{\mathcal{FOL}}$ is defined for each $(F, \mathrm{Nom}, \Lambda)$-sentence by $\alpha(\rho) = (\forall w)\alpha_w(\rho)$, where w is a variable of ST and α_w is recursively defined as follows:

$$
\begin{array}{rll}
\alpha_w(t \approx t') = \mathcal{T}_w(t) \approx \mathcal{T}_w(t') & & t, t' \in (T_\Sigma(x))_s, s \in S^\Sigma \\
\alpha_w(Q(t_1, \ldots, t_n)) = \bar{Q}(w, \mathcal{T}_w(t_1), \ldots, \mathcal{T}_w(t_n)) & & Q \in P^\Sigma_{s_1,\ldots,s_n}, t_i \in (T_\Sigma(X))_{s_i} \\
\alpha_w(i) = c_i \approx w, & & i \in \mathrm{Nom} \\
\alpha_w(p) = \bar{p}(w), & & p \in \mathrm{Prop} \\
\alpha_w(@_i\rho) = \alpha_{c_i}(\rho), & & \\
\alpha_w([\lambda]\rho) = (\forall v)[(w, v) \in R_\lambda \rightarrow \alpha_v(\rho)], & & \lambda \in \Lambda \\
\alpha_w(\neg\rho) = \neg\alpha_w(\rho) & & \\
\alpha_w(\rho \odot \rho') = \alpha_w(\rho) \odot \alpha_w(\rho'), & & \odot \in \{\vee, \wedge, \rightarrow\} \\
\alpha_w(\forall x \rho) = \forall x\ \alpha_w(\rho) & & x \in X
\end{array}
$$

where $\mathcal{T}_w : T_\Sigma(X) \rightarrow T_{\bar{\Sigma}}(\overline{X})$, for $\overline{\Sigma} = (\overline{S^\Sigma}, \overline{F^\Sigma}, \overline{P^\Sigma})$, defined for each variable $x \in X$, $\mathcal{T}_w(x) = x$ and for each $f(t_1, \ldots, t_n) \in T_\Sigma(X)$ by $\mathcal{T}_w(f(t_1, \ldots, t_n)) = \bar{f}(w, \mathcal{T}_w(t_1), \ldots, \mathcal{T}_w(t_n))$.

Theorem 2. *Let $\Delta \in |\mathrm{SIGN}^{\mathcal{H}}|$, $\rho \in \mathrm{SEN}^{\mathcal{H}}$ and $M' \in \mathrm{Mod}^{\mathcal{FOL}}(\Phi(\Delta))$. Then, for α and β defined as above, for any $s \in S$ and any assignment $g : \bar{X} \rightarrow A$ such that whenever $g(w) = s$, we have that*

$$\beta_\Delta(M'), g\restriction_X \models^s_{\mathcal{H}} \rho \text{ iff } M', g \models^{\mathcal{FOL}}_{\Phi(\Delta)} \alpha_w(\rho). \tag{4}$$

Proof. The proof is done by induction on the structures of sentences.

As direct consequence we have the general satisfaction condition for comorphisms:

Corollary 3 (Comorphism satisfaction condition). *Let $\Delta \in |\mathrm{SIGN}^{\mathcal{H}}|$, $\rho \in \mathrm{SEN}^{\mathcal{H}}$ and $M' \in \mathrm{Mod}^{\mathcal{FOL}}(\Phi(\Delta))$. Then, for α and β defined as above we have that,*

$$\beta_\Delta(M') \models^{\mathcal{H}}_\Delta \rho \text{ iff } M' \models^{\mathcal{FOL}}_{\Phi(\Delta)} \alpha_\Delta(\rho). \tag{5}$$

Moreover it is conservative: this is directly entailed by the assumption that states have constant domains. It is straitforward to see that, we may define a comorphism from $\mathcal{HPL}$ into $\mathcal{FOL}$ from the presented one. This is achieved by forgetting the first-order components of the signatures and models and by restricting α to the hybrid propositional formulas.

Recalling our running example, we end up with the signature

ops

$Speed^* : st^* \times speed \times accel \rightarrow speed$; $Pedal^* : st^* \times time \rightarrow accel$;...

pred

$next : st^* \times st^*$; $IgnOn^* : st^*$; ...

Note that, now, global properties are universally quantified, and local properties take as state argument the respective nominal. For instance, global properties (G_1) and (G_2) are translated into

$\forall\ s : speed;\ w : st^*;\ a : accel; t : time$
- $(G_{1^*})\ \geq^*(w\ , Speed^*(w,\ s,\ a),\ 0^*(w))$
- $(G_{2^*})\ CurrentSpeed^*(w,t) = 0^*(w) \wedge \geq^*(w,\ Pedal^*(w,t),\ 0^*(w))$.

and local properties (L^1_{off}) and (L^4_{cruise}), into

$\forall\ t : time$
- $(L^{1^*}_{off})\ CurrentSpeed^*(off,t) = 0^*(off)$
- $(L^{4^*}_{cruise}) \geq^*(cruise, Pedal^*(cruise,t), 0^*(cruise)) \Rightarrow Pedal(cruise,t) = Automatic^*(cruise,t)$.

For instance, transition (T_1) is expressed by

$\bullet(T_{1*})\ IgnOn(off) \Rightarrow$
$[(\forall w : st^*)\ (off, w) \in next \Rightarrow inactive = w \wedge (\exists w' : st^*)\ (off, w') \in next \Rightarrow inactive = w']$,
i.e.,
$\bullet IgnOn(off) \Rightarrow (off, inactive) \in next.$

4 Tool Support

A central ingredient for the successful integration of a formal method in the industrial practice is the existence of effective tool support.

Certainly hybrid specifications produces in Stage I of our methodology can be anchored on recent implementations of logical calculus for HPL (see e.g. HTab [11], HyLoTab [25] and Spartacus [8]). Moreover, model checking for HPL models is also an active research issue (e.g. [12,10]).

Our focus is, however, a different, somehow more standard, one: hybrid specifications are translated to $\mathcal{FOL}$ through a suitable comorphism. This solution provides a uniform first order logical framework for analysis and verification supporting the whole methodology. Moreover, to the best of our knowledge, richer versions of hybrid logic, as required at Stage II, lack effective tool support, which makes our approach by translation the only option available. Beyond the conceptual support of institutions theory and the structured specification methodology offered by CASL, we have effective computational tools, to support our sort of specification. On this perspective HETS-*heterogeneous tools set* [20] deserves a special attention.

Using a metaphor of [19], HETS may be seen as a "motherboard" where different "expansion cards" can be plugged. These pieces are individual logics (with their particular analyzers and proof tools) as well as logic translations. To make them *compatible*, logics have to be formalized as institutions and, the corresponding translations, as comorphisms. Therefore, the integration of the hybrid specifications on the HETS framework is legitimate, since all formal requirements (e.g., that institutions exist, that a comorphism can be defined, etc.) are provided in the present work. HETS already integrates parsers, static analyzers and provers for a wide set of individual logics, and manages heterogeneous proofs resorting to the so-called graphs of logics, i.e., graphs whose nodes are logics and, whose edges, are comorphisms between them.

Furthermore, and directly relevant to our methodology, HETS provides a rich support for $\mathcal{FOL}$, and consequently, for $\mathcal{H}$ and $\mathcal{HPL}$. For instance, provers SoftFOL, Spass, MathServe Broker, among others, are already "pluged" into HETS [19], and therefore, all of them provide effective to our specification methodology (see Figure 3). Moreover, we are also able to take advantage of a number of "borrowed" provers from other institutions through comorphisms with source in $\mathcal{FOL}$.

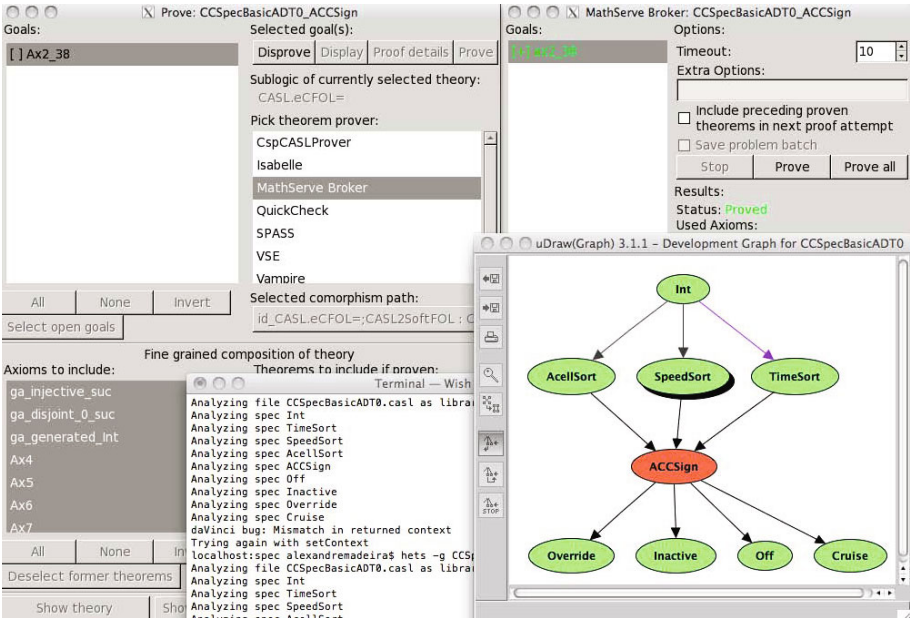


Fig. 3. HETS session

An open issue at this level concerns *verification*. So far model checking of hybrid structures is restricted to propositional hybrid logic [6,12]. The combination of traditional algebraic specification tools, like first-order provers and rewriting engines (e.g. CafeObj [5]), together with provers and model checkers for hybrid logics (e.g. [1,6]) may broaden the scope of application.

5 Conclusions

The paper introduced a rigorous methodology for requirements specification of reactive systems, flexible enough to capture the existence of different operational modes at each stage of evolution. Variants of hybrid logic provided the right conceptual framework to develop such specifications. At a later stage, such specifications are translated into first-order logic to bring into scene suitable tool support. The paper's main contribution was to provide rigorous foundations for the method, framing specification logics as institutions and the translation process as a comorphism between them.

A lot of work remains to be done. From an experimental point of view, we are conducting case studies with different size and complexity to assess the methodology.

Another line of research is concerned with establishing a precise comparison with approaches to specification with a similar purpose. For instance, many (variations) of state machines may be represented as hybrid models. Moreover, some structured state-machines, such as ASM (Abstract State Machines) [3] can also be represented as our states-as-algebras models. An interesting aspect to explore, is whether the institutions constructed here may provide an uniform platform to reason, in a property-oriented perspective, about these model-oriented specifications. Moreover, recent theoretical developments from the authors justifies to look to the presented methodology in a more broad sense: it proofs in [16] that the hybridization idea presented above can be extended to arbitrary institutions. Trough this result it would be worth to consider, on place of the first-order structures, other kind of semantical models such algebras, temporal frames or even Haskel modules, since all of these structures are objects of some particular institution.

Last but not least, *refinement*. At stage III FOL is used as a common language to support reasoning and verification on models built on stages I and II. It is, therefore, expectable to find a way to use this common platform to formally relate these models. In particular, it would be important to formally assure that requirements specified on the first stage are not violated on the second one. This entails the need for a rigorous formalization of the intuitive arrow "?" of figure 1. A natural candidate to do this job, is the classical concept of refinement from algebraic specifications (e.g. [23]). Throughout this notion, a specification SP refines a specification SP_0 over the same signature, if all the properties satisfied by SP_0 are also satisfied by SP. More generally, when specification signatures are related by a morphism, a translation of properties is in order wrt to the signature morphism.

In general, however, this refinement relation is not adequate. For instance, as suggested on stage II, it is expectable to map a proposition of the state-machine into an equation on the respective states-as-algebras model. These formulas are represented in $\mathcal{FOL}$ by a predicate and an equation, respectively, which cannot be related through signature morphisms (which only relate predicates with predicates and equations with equations). Less conventional approaches to refinement may help to overcome this sort of situations. A possibility we are currently investigating is to resort to logical interpretations, instead of signature morphisms, to direct refinement as studied by the authors in [15,14,22]. Interpretations are multi-functions between the specifications formulas which preserve and reflect logical consequence.

There are others specification frameworks also based on modal versions of first-order logic. For instance, in [21] it is defined a logic (for hybrid systems) based on a dynamical version of first-order logic (over $\mathbb{R}$) with nominals. It is important to note that the semantical paradigm of those approaches is quite different for the proposed here; namely, as usual, they deal with states as values

of system variables on of given moment of execution, evaluated in an unique first-order structure. In our work, it corresponds not to a set of values, but to state-families of first-order structures, modeling the behaviour of all the system functionalities.

References

1. Areces, C., Heguiabehere, J.: Hylores: A hybrid logic prover based on direct resolution. In: Proceedings of Advances in Modal Logic, AiML 2002 (2002)
2. Blackburn, P.: Representation, reasoning, and relational structures: a hybrid logic manifesto. Logic Journal of IGPL 8(3), 339–365 (2000)
3. Börger, E., Stärk, R.: Abstract state machines: A method for high-level system design and analysis. Springer, Heidelberg (2003)
4. Diaconescu, R.: Institution-independent Model Theory. Birkhäuser, Basel (2008)
5. Diaconescu, R., Futatsugi, K.: Logical foundations of CafeOBJ. Theor. Comput. Sci. 285(2), 289–318 (2002)
6. Franceschet, M., de Rijke, M.: Model checking for hybrid logics (with an application to semistructured data). Journal of Applied Logic 4(3), 279–304 (2006)
7. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. J. ACM 39, 95–146 (1992)
8. Götzmann, D., Kaminski, M., Smolka, G.: Spartacus: A tableau prover for hybrid logic. Electr. Notes Theor. Comput. Sci. 262, 127–139 (2010)
9. Heitmeyer, C.L., Kirby, J., Labaw, B.G.: The SCR Method for Formally Specifying, Verifying, and Validating Requirements: Tool Support. In: ICSE, pp. 610–611 (1997)
10. Hoareau, C., Satoh, I.: Hybrid logics and model checking: A recipe for query processing in location-aware environments. In: AINA, pp. 130–137. IEEE Computer Society, Los Alamitos (2008)
11. Hoffmann, G., Areces, C.: Htab: a terminating tableaux system for hybrid logic. Electr. Notes Theor. Comput. Sci. 231, 3–19 (2009)
12. Lange, M.: Model checking for hybrid logic. J. of Logic, Lang. and Inf. 18(4), 465–491 (2009)
13. Madeira, A., Faria, J.M., Martins, M.A., Barbosa, L.S.: Hybrid specification of reactive systems: An institutional approach (extended version). Technical Report CCTC-11-03, University of Minho (July 2011)
14. Martins, M.A., Madeira, A., Barbosa, L.S.: Refinement by interpretation in a general setting. Electron. Notes Theor. Comput. Sci. 259, 105–121 (2009)
15. Martins, M.A., Madeira, A., Barbosa, L.S.: Refinement via interpretation. In: Hung, D.V., Krishnan, P. (eds.) SEFM, pp. 250–259. IEEE Computer Society (2009)
16. Martins, M.A., Madeira, A., Diaconescu, R., Barbosa, L.S.: Hybridization of institutions. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 283–297. Springer, Heidelberg (2011)
17. Mossakowski, T.: Foundations of heterogeneous specification. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 359–375. Springer, Heidelberg (2003)
18. Mossakowski, T., Haxthausen, A., Sannella, D., Tarlecki, A.: CASL: The common algebraic specification language: Semantics and proof theory. Computing and Informatics 22, 285–321 (2003)

19. Mossakowski, T., Maeder, C., Codescu, M., Lucke, D.: Hets user guide - version 0.97. Technical report, DFKI Lab Bremen (March 2011), http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/index_e.htm
20. Mossakowski, T., Maeder, C., Lüttich, K.: The heterogeneous tool set, HETS. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
21. Platzer, A.: Towards a hybrid dynamic logic for hybrid dynamic systems. Electron. Notes Theor. Comput. Sci. 174, 63–77 (2007)
22. Rodrigues, C.J., Martins, M.A., Madeira, A., Barbosa, L.S.: Refinement by interpretation in π-institutions. EPTCS 55, 53–64 (2011)
23. Sannella, D.: Algebraic specification and program development by stepwise refinement (Extended abstract). In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 1–9. Springer, Heidelberg (2000)
24. Tarlecki, A.: Abstract specification theory: An overview. In: Broy, M., Pizka, M. (eds.) Models, Algebras, and Logics of Engineering Software. NATO Science Series, Computer and Systems Sciences, vol. 191, pp. 43–79. IOS Press, Amsterdam (2003)
25. van Eijck, J.: Hylotab-tableau-based theorem proving for hybrid logics. Technical report, CWI (2002), http://homepages.cwi.nl/~jve/#Publications

Leveraging State-Based User Preferences in Context-Aware Reconfigurations for Self-Adaptive Systems

Marco Mori[1], Fei Li[2], Christoph Dorn[3],
Paola Inverardi[4], and Schahram Dustdar[2]

[1] IMT Institute for Advanced Studies Lucca
marco.mori@imtlucca.it
[2] Distributed System Group, Vienna University of Technology
{li,dustdar}@infosys.tuwien.ac
[3] Institute for Software Research, University of California, Irvine
cdorn@uci.edu
[4] Dip. di Informatica, Università dell'Aquila
paola.inverardi@di.univaq.it

Abstract. Applications in ubiquitous environments need to adapt to a range of fluid factors, like user preferences, context, and various system configurations. In this paper, we address the problem of system adaptation in order to continuously achieve high user benefit while keeping reconfiguration costs low. To this end, the presented approach leverages not only the immediate context but also future transitions. In contrast to existing approaches that either maximize benefit or minimize reconfiguration costs, our proposed decision support mechanism achieves a trade-off between those factors. Considering user preferences, deployment constraints, and probabilistic context state transitions, we propose a multi-objective utility function to determine the best reconfiguration choices. Experimental results show that the proposed approach achieves high user benefit while keeping reconfigurations costs low.

1 Introduction

As ubiquitous computing is becoming more and more widespread, software engineers have to deal with different variability dimensions including the system context, users, and the system configuration itself. Changes are not always predictable since they are beyond the control of the system and they may require human intervention. To reduce maintenance cost it is desirable to achieve automatic self-adaptations in response to various kinds of changes. Adaptations should meet the desired quality requirements according to user preferences and they should be performed at reasonable cost and in a timely manner.

Self-adaptive systems are able to adjust their run-time behavior in face of changing external circumstances [17,4,1]. Software engineers define a set of alternative behaviors at design time while the actual adaptation decisions are postponed to run-time. Context plays a key role for adaptation since it determines

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 286–301, 2011.

the variation of user preferences and the space for the admissible adaptation alternatives. Context, thus, needs to be explicitly modeled in order to account for the run-time adaptation as proposed in research of context-aware systems [3,10].

This paper addresses the problem of how to achieve simultaneous adaptation to system execution context and user preferences. The execution environment determines the space of admissible system configurations whereas the user determines the benefit of each available configuration. Switching between different configurations comes at some cost. Consequently, predictive information promises a significant cost saving potential by making adaptation decisions aware of probable future context changes and thereby anticipates upcoming reconfiguration needs. When determining system configurations, the challenge lies in finding a suitable trade-off between two opposite objective functions: maximize user benefit while minimize reconfiguration costs. Pure user benefit-driven selection comes with high costs due to frequent reconfigurations. In contrast, pure cost-driven adaption neglects user preferences and invariably prefers the current configuration, thus it changes the system configuration only when absolutely necessary. For balancing the two factors, we define our solution as a multi-criteria selection problem among different system alternatives and we evaluate each of them through an aggregating objective function that combines cost and user benefit. Experiments based on a case study and simulation demonstrate that our approach successfully determines Pareto-optimal configurations of high user benefit and low reconfiguration costs. Our contributions in this paper are:

- a concrete methodology to characterize system resources and configuration eligibility while
- defining user preferences on non-functional properties specific to distinct user contexts (situations).
- applying probabilistic information on future context states to predict upcoming context changes
- defining the configuration selection as a multi-criteria optimization problem.

The remainder of this paper is structured as follows: Section 2 presents a motivating scenario followed by a description of our approach in Section 3. Section 4 introduces the optimization framework in terms of their main components, while Section 5 formalizes the optimization problem. Section 6 provides evaluation and validation results based on our case study and a simulation. Section 7 discusses related work before conclusions and future work round up our paper in Section 8.

2 Motivating Scenario

An e-Health application supports doctors' activities by providing the most relevant services to visualize per-patient case history. Patient information is available at three levels of granularity: (i) a complete case history that includes textual reports and medical images, (ii) a compact version with only the recent history of reports and images, and (iii) only a textual case history. In addition images are displayed either as black and white images, in low color (256 colors), or as fully colored images (4096 colors).

Doctors need to receive aggregated per-patient information to support their activities at different locations. These activities include patient consulting, check-up, and medical procedures such as operations. Moreover they may be involved in emergency situations. These activities are performed at different locations such as common visiting rooms, surgery rooms, patient home or outside the hospital when an emergency arises. The doctor is able to visualize per-patient information through an accessible device inside or outside the hospital. Devices differ in their hardware resources such as bandwidth availability ($netB$), number of screen colors (SC), CPU speed ($CPUClockRate$) and available memory (Mem). Hardware has an impact on the available services: e.g., low bandwidth and 8 bit colors restrict the responsiveness to retrieve the patient's medical history and available image quality.

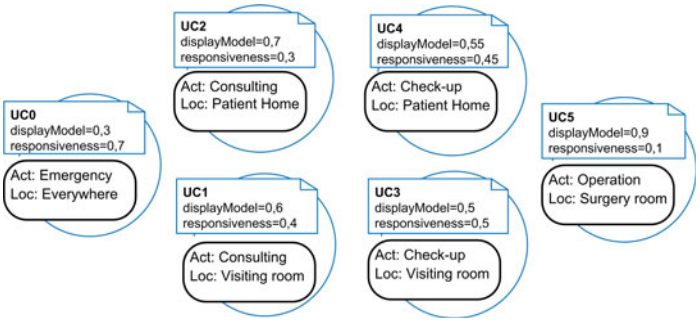


Fig. 1. User Preferences Example

Activity and location influence the doctor's preference for displaying the case history and image quality, see Fig. 1. The doctor might prefer a responsive system in case of an emergency activity. In another case, immediate retrieval of per-patient information is not as important as a detailed history for consulting activities. Upon context changes, the e-Health application needs reconfiguration based on the underlying hardware resources and the doctor's (context-dependent) preferences.

3 Approach

From a feature engineering perspective, features are the basic unit of behavior [6]. Breaking the system logic into feature components enables us to reduce the impact that any change might have on the system. Thus, we represent each alternative system variant as a distinct configuration of features. In [9] we defined a methodology to create the space of admissible configurations for a self-adaptive application starting from the set of basic features. In this paper we are going to define a decision making mechanism that is suitable for feature-based systems having the cost of deploying a feature independent from the running configuration. Figure 2 visualizes the conceptual aspects of our work. We consider for each

configuration a set of deployment constraints to assess the configuration admissibility. These constraints are evaluated against the current underlying context (resources) to establish whether the environment can support the execution of that particular configuration. On the other hand, we map a feature configuration to non-functional properties to represent the configuration's quality. This quality becomes a configuration utility (i.e., user benefit) when matched with user preferences. For each user, we assume the availability of historical transitions between the various context states. We also assume the time required for system adaptation upon a state transition to be negligible compared to the frequency of user context changes.

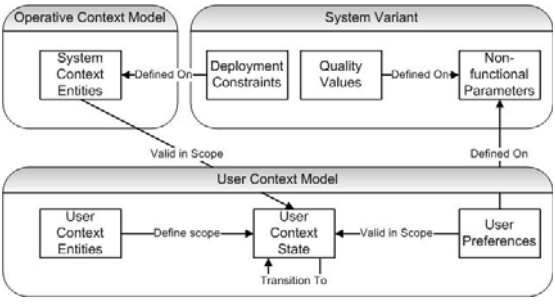


Fig. 2. Conceptual Model

4 Basic Models

In this section we introduce the basic definitions and models we use to formulate the context-aware reconfiguration problem. System reconfiguration aims at satisfying two objectives: user benefit and cost which arise due to system reconfiguration. We represent each *system variant* in terms of a *deployment constraint* and the provided *non functional properties*. We propose a definition of *transition cost* to penalize reconfigurations. In our framework, there are two types of context models. On one hand, features express their deployment constraint as conditions on the *operative context model*. In contrast, user preferences are not static but depend on the underlying *user context entities* instances. Thus, the *user context model* provides mean to map particular user preferences (over the *non-functional properties*) to specific context instances. We exploit the operative context model to evaluate which alternatives are admissible at a given point in time, and the user context model to deal with the current and probable future user preferences.

4.1 Operative Context Model

The system variants describe which resources they require for execution and thus need to be represented in the operative context model. Each system context entity is identified through a tag within the set $TagId = \{TagId_1, ..., TagId_n\}$ and it can assume one among its admissible values contained in the corresponding

finite domain $\mathbb{D}_1, ..., \mathbb{D}_n$. We adopt the modeling approach proposed in [13] to represent our definition of *operative context space* and *operative context scope*.

The *operative context space* for the system context entities is defined as the Cartesian product of their admissible values: $O = \bigotimes \mathbb{D}_i \;\; s.t. \;\; i = 1, ..., n$. Each element in O is a vector r expressing a different assignment of values, e.g. $r = (netB(100Kbps), Mem(10MB), SC(256colors), CPUClockRate(100Mhz))$.

An *operative context scope os* is a subset of the operative context space O, $os \in 2^O$, e.g. $os = (netB(100-200Kbps), Mem(10-50MB), SC(10-20colors), CPUClockRate(100-150Mhz))$.

4.2 System and System Variants

We have adopted the feature engineering perspective to express the system variants. Each basic unit of behavior is expressed as a feature, that is the smallest unit of behavior that can be perceived by the user [11]. System variants are expressed as configurations; each one obtained by combining subsets of features. We define each configuration in terms of deployment constraints and the fitness values for the non-functional properties.

Deployment Constraints. It is a predicate which expresses the demand to the operative context entities for a configuration c, e.g. $netB(5kbps) \wedge Mem(0, 1MB)$ is true only with a sufficient level of bandwidth and memory. An operative context scope os_c entails the elements in O that make the predicate true. Then, we evaluate if a configuration c is eligible in a context scope os with the function f_c which is equal to 1 only if $os \subseteq os_c$ and 0 otherwise. In our problem we also exploit the function $Eligible(r)$ to evaluate which configurations are eligible with the context values in r.

Non-functional Properties. Each configuration for an adaptive application provides different qualities to the user. These non-functional properties $NFP = \{nfp_1, nfp_2, ..., nfp_s\}$ can be quantitatively measured to drive the adaptation and to guarantee the user benefit. We map the properties values (defined over finite domains) to normalized fitness values in the real range [0,1]. For each configuration c, the vector fv_c contains the fitness values.

Transition Cost. An important factor to consider during the reconfiguration process is the penalty of switching from the source configuration to the target configuration. Since in our approach system configurations are made by features, we characterize this penalty based on the distance between the two configurations as $Dist_{y,z} = [NToDeploy \; NToUnDeploy]$ expressing the number of features to deploy and un-deploy switching from y to z. The vector $FCost = [CDeploying_f \; CUnDeploying_f]$ contains the same cost of deploying and un-deploying a feature. Based on the two vectors we define the transition cost of switching from y to z as:

$$TC(y, z) = (Dist_{y,z} \cdot FCost^T)/MaxCost \tag{1}$$

This cost is normalized to the maximum theoretical cost, which depends by the maximum number of features to deploy and un-deploy:

$$MaxCost = [MaxToDeploy\ MaxToUnDeploy] \cdot FCost^T \tag{2}$$

This simplified cost model is sufficient for our purpose since we do not address the problem of executing the actual system reconfiguration at the implementation level.

4.3 User Context Model

User context entities characterize the user's situation. As they are beyond the control of the application, they play a key role in the adaptation process. As mentioned in Section 2, the user's preferences change when switching from one user context state to another. Note that our approach is independent from the actual user context entities and how they change as long as there is a mapping of the various observable user context states to user preferences.

We define a mapping between the user context state UC — as defined by a set of user context entities — and the associated user preferences. User preferences express the importance (i.e., weight) of the various non-functional properties in a given context state. Higher weights express higher importance applied in the mapping functions $w : NFP \rightarrow [0, 1]$. Furthermore, we introduce a probabilistic automaton to represent the changing user preferences as induced by the underlying transitions between context states.

This automaton is defined as $A = (UC, P, E)$ where:

- $UC = \{UC_0, ..., UC_t\}$ is the set of states expressing the space of the user preferences. Each state is represented as a different combination of weights upon the non-functional parameters: $UC_j = [w_1(nfp_1)...w_s(nfp_s)]\ j = 1, .., t$; at each state the weights are defined as: $\sum_{i=1}^{s} w_i(nfp_i) = 1$
- P is the set of transition probabilities
- $E : UC \times P \rightarrow UC$ is the probabilistic transition function

This probabilistic state-based model shows how the preferences reflect the changes of user context entities. Historical data collected during system execution allows us to determine the actual transition probabilities between user context states. We continuously sample user context data at fixed intervals of time so that the probability to have two or more preference changes (i.e., context changes) within one interval is negligibly low. This process, however, is beyond the scope of this paper. Nevertheless techniques like [12] show the possibility to get preferences from user context, whereas methodologies like [14] define how to build a probabilistic model and maintain it updated with current system execution.

We expect that the various user context states come with changes in the operative context space. For example, bandwidth will not be the same in every location. Consequently, we consider also if a particular system variant is admissible in the observed user context state, independent from user preferences. We define

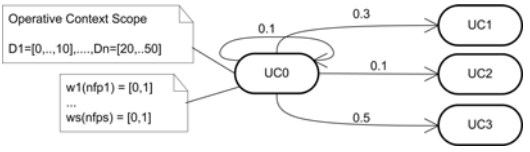


Fig. 3. Probabilistic automaton excerpt

a mapping function to associate each state in UC with an operative context scope within the set OS ($UCR : UC \rightarrow OS$). This models the correspondence between the user preferences and the observed system context entities. Fig. 3 provides an excerpt of a probabilistic automaton, detailing the mapping of user preferences and operative context scopes to a user context state.

5 Problem Formalization

Two events trigger the optimization problem and subsequent reconfiguration. Either the user moves into a new user context state characterized by a changing preference or the operative context cannot support the execution of the current system variant anymore. The best configuration to deploy depends on the achievable user benefit and the associated costs for reconfiguring the system. A strategy that maximizes the user benefit after each transition possibly requires many system reconfigurations. On the other hand choosing a fixed configuration which is always eligible throughout all states may result in possibly sub-optimal user benefit or may not exist at all. As a consequence we have to consider a trade-off analysis between two conflicting criteria, i.e. user benefit and reconfiguration costs. In the following we formalize the user benefit, the reconfiguration cost, and describe their combination in a single utility function.

As shown in Eq.3 the component B_{curr} evaluates how well a certain configuration c fits the current user context state. The user benefit at each state is the product of the corresponding user preferences vector with the quality attribute fv_c offered by the configuration.

$$B_{curr} = UC_{curr} \cdot fv_c^T \tag{3}$$

A configuration that gives optimal user benefit for a certain state may be sub-optimal if we consider the probable future states. Therefore we introduce an equation component that evaluates the expected user benefit in the future as given by the probabilistic context transitions. The cost component B_F shown in Eq.4 computes the future benefit of a configuration. We limit the calculation of future benefit to a single hop in the transition graph. Considering additional states (i.e., multiple hops) is expected to yield little additional benefit as each of the reachable states will have very small probability and thus hardly any impact.

$$B_F = \sum_{j=1}^{\#OutLink(UC_{curr})} p(UC_{curr}, UC_j) \cdot \left[UC_j \cdot fv_c^T\right] \cdot f_c(os_j) \tag{4}$$

B_F aggregates the user benefit for each subsequent user context state weighted according to the respective transition probability. A configuration yields user benefit only if it is eligible in the corresponding operative context scope ($f_c(os_j) = 1$). Ultimately, the overall user benefit equation is obtained by combining the current and future user benefits as follows:

$$B_{Agg} = h \cdot B_{curr} + (1 - h) \cdot B_F \tag{5}$$

The horizon h regulates the importance of the current user benefit compared to the future user benefit. The horizon close to 1 expresses a preference for the current state, whereas for h close to 0 we deem the future more relevant. Thus for environments where the user is expected to rapidly switch between states, the horizon configuration parameter should be closer to 0 as he/she will leave the current state soon.

The reconfiguration cost TC represents the cost of switching from the current deployed configuration to the configuration c (Eq. 1). The problem of selecting the best configuration in a operative context model state r, given a predefined user context model, is formalized as a max optimization problem combining the expressions defined in Eq. 3, 4, 1:

$$\max_{c \in Eligible(r)} \alpha \cdot [h \cdot B_{curr} + (1 - h) \cdot B_F] - (1 - \alpha) \cdot TC(c_{curr}, c) \tag{6}$$

The parameter α regulates the trade-off between user benefit and reconfiguration cost. Setting α closer to 1 makes the optimization more likely to meet the user benefit in spite of a high cost of reconfiguration. When setting this parameter closer to 0, we reduce the reconfiguration cost by selecting general purpose system variants that may be sub-optimal on the user benefit. The parameter h enables to tune the interest between the current user preferences and the probable future user preferences as explained above.

By introducing the variables α and h we make our optimization process customizable to various environments. The horizon h enables tuning to self-transitions in the context automata. If the resulting self-transitions are very high but still we are interested in optimizing future preferences we need to decrease the value of h. On the other hand if we end up with low self-transitions but we want to better match current preferences we have to increase the value of h. In addition, by setting $h = 1$ we enable comparison to existing approaches that are future-unaware.

6 Evaluation

This section presents two ways to evaluate our contribution. Firstly, we model the motivating scenario in Section 2, and apply our approach to find an optimal solution to the scenario. Secondly, we simulate the reconfiguration process at large scale in order to provide general guidelines of parameter settings to extensive application scenarios.

6.1 Case Study

Applying the feature engineering perspective the e-Health scenario yields following alternative features to view the per-patient case history: $S = \{f_{viewAllIm}, f_{viewLastIm}, f_{viewAllRep}, f_{viewLastRep}, f_{viewSum}, f_{paintBW}, f_{paintCol}, f_{paintFCol}\}$.

Table 1. System Configurations

Configuration	Deployment Constraint	Non-Functional Properties
$c_1 = \{f_{viewSum}\}$	$netB(5kbps) \wedge Mem(0,1MB)$	$displayModel = summary$ $responsiveness = high$
$c_2 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintBW}\}$	$netB(20kbps) \wedge Mem(2,5MB) \wedge$ $CPUClockRate(40Mhz)$	$displayModel = lastHistory$ $responsiveness = mediumHigh$
$c_3 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintCol}\}$	$netB(20kbps) \wedge Mem(2,5MB) \wedge$ $CPUClockRate(50Mhz)$	$displayModel = lastHistory$ $responsiveness = mediumHigh$
$c_4 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintFCol}\}$	$netB(200kbps) \wedge Mem(40MB) \wedge$ $CPUClockRate(10Mhz) \wedge SC(4096colors)$	$displayModel = lastHistory$ $responsiveness = mediumLow$
$c_5 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintBW}\}$	$netB(40kbps) \wedge Mem(10MB) \wedge$ $CPUClockRate(40Mhz)$	$displayModel = completeHistory$ $responsiveness = medium$
$c_6 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintCol}\}$	$netB(40kbps) \wedge Mem(10MB) \wedge$ $CPUClockRate(50Mhz) \wedge SC(256colors)$	$displayModel = completeHistory$ $responsiveness = medium$
$c_7 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintFCol}\}$	$netB(800kbps) \wedge Mem(160MB) \wedge$ $CPUClockRate(100Mhz) \wedge SC(4096colors)$	$displayModel = completeHistory$ $responsiveness = low$

Table 1 lists the 7 configurations built from these proposed features. Configurations c_1 provides only a textual representation of the patient's case history ($summary$). The next three configurations display only the very recent case entries ($lastHistory$) by means of textual reports and medical images which may be colored following the three different modes (BW, $Fullycolored$, $Colored$). The last three configurations display the complete case history ($completeHistory$) with a different coloring modality. The feature combination in each configuration determines the responsiveness level (ranging from Low to $High$). In the following we describe how the reconfiguration process takes place whenever the user switches context. We potentially observe a change of the user preferences when the doctor moves to a different location or engages in a different task. If this is the case, we then have to evaluate which configuration maximizes Eq. 6. We select the best configuration starting from the following inputs: the set of eligible configurations in the current operative context, the user context automata and the reconfiguration costs. In this case study, we obtain a user context automaton with the transitions probabilities shown in Fig.4 by analyzing historical user data detailing the movements and the doctor's working timetable. Each state is characterized by different weights for each quality attribute (UC_0 = [0.3 0.7], UC_1 = [0.6 0.4], UC_2 = [0.7 0.3], UC_3 = [0.5 0.5], UC_4 = [0.45 0.55], UC_5 = [0.9 0.1]). The first component of each vector indicates how important the *displayModel* property is, while the second expresses the weight for *responsiveness*. In addition each user context state is associated to a different operative context scope $OS_0, .., OS_5$.

Suppose the doctor changes from an emergency activity to a check-up activity within the hospital visiting room. As a consequence the user context switches from UC_0 to UC_3 and the reconfiguration process commences. Note that the user is free to switch between context states which exhibit no corresponding transition in the automaton. Let us suppose that the running configuration is c_2 and the

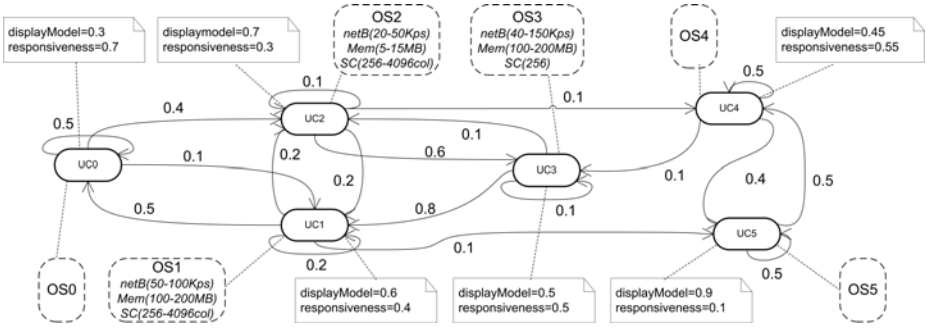


Fig. 4. User Context Automata

operative context state is $r_{curr} = (netB(50Kbps), CPUClockRate(100Mhz), Mem(20MB), SC(256colors))$. We check the deployment constraints for the configurations in Table 1 against the state r_{curr}. Thus we compute the set of eligible configuration as $Eligible(r_{curr}) = \{c_1, c_2, c_3, c_5, c_6\}$. Each configuration provides two non-functional properties $NFP = \{displayModel, responsiveness\}$. The first assumes one value among *summary*, *lastHistory* and *completeHistory* whereas the second assumes one value among *low*, *mediumLow*, *mediumHigh*, *medium* and *high*. Starting from the qualities offered by each configuration we evaluate the parallel fitness vectors exploiting a possible normalization:

$$\begin{aligned}
c1: &\ [summary\ high] &&\Rightarrow fv_{c1} = [0.1\ 0.8]\\
c2: &\ [lastHistory\ mediumHigh] &&\Rightarrow fv_{c2} = [0.5\ 0.65]\\
c3: &\ [lastHistory\ mediumHigh] &&\Rightarrow fv_{c3} = [0.5\ 0.65]\\
c5: &\ [completeHistory\ medium] &&\Rightarrow fv_{c5} = [0.9\ 0.5]\\
c6: &\ [completeHistory\ medium] &&\Rightarrow fv_{c6} = [0.9\ 0.5]
\end{aligned}$$

For purpose of demonstrating, we assume the cost of deploying and un-deploying any feature is $FCost = [2\ 1]$. The distance between each admissible configuration and the current one ($c_2 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintBW}\}$) in terms of features to deploy and un-deploy is given in Table 2. The normalized costs of switching from the current configuration to each possible target one are: $TC(c_2, c_1) = 0.555$, $TC(c_2, c_2) = 0$, $TC(c_2, c_3) = 0.333$, $TC(c_2, c_5) = 0.667$, $TC(c_2, c_6) = 1$. The maximum theoretical cost we exploit for the normalization is evaluated as $MaxCost = [3\ 3] \cdot [1\ 1]^T = 6$ (Eq. 2). We solve the optimization problem at Eq.6 considering the new user context state UC_3, the set of eligible configurations at the operative state r_{curr}, and the costs. For demonstrating our approach we set α to 0.7 to express that the user benefit is more important than

Table 2. Distance evaluation

Dist./Conf.	c_1	c_2	c_3	c_5	c_6
ToDeploy	1	0	1	2	3
ToUnDeploy	3	0	1	2	3

costs. We also set the variable h to 0.5 to consider equally current and future user preferences.

Our proposed methodology enables selecting the configuration which fits better the current preferences while considering the future user preferences. Future preferences are determined by the probable future task and location in which the doctor will be involved. In addition also the costs of switching configuration are taken into account.

At the current user context state ($UC3$, the one where the user just arrived), the doctor is performing a check-up activity at the visiting room where the $responsiveness$ and $displayModel$ properties are equally ranked (Figure 1). By looking at the automata in Fig.4 we reason that with very high probability the doctor will thereafter switch to another state (UC_1). This probable subsequent state comes with different weights for $responsiveness$ and $displayModel$ (UC_1). As a consequence we anticipate this future transition by selecting a system configuration which provides already better display modality now, even if it does not strictly meet the current user preferences. Nevertheless, in this example the top ranked configuration maximizes also the current preferences.

Table 3 presents the overall utility value for the eligible configurations obtained by combining the user benefit component (Eq.5) and the cost (Eq. 1). User benefit components do not need normalization since they are evaluated exploiting normalized user preferences and normalized quality vector. In this illustrative example the best configuration is c_6 since it corresponds to the best trade-off between user benefit and costs with given h and α.

Table 3. Configurations evaluation

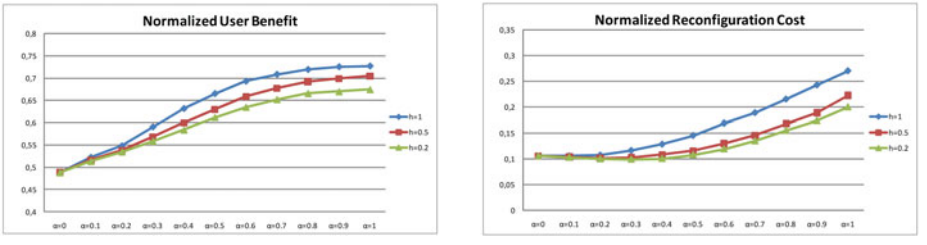
Configuration	B_{curr}	B_F	Cost	**Overall utility**
$c_1 = \{f_{viewSum}\}$	0.45	0.38	0.555	0,457
$c_2 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintBW}\}$	0.575	0.56	0	0,397
$c_3 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintCol}\}$	0.575	0.56	0.333	0,497
$c_5 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintBW}\}$	0.7	0,662	0.667	0,6767
$c_6 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintCol}\}$	0.7	0,662	1	**0,7767**

6.2 Experiment

Besides a case study, we validate our approach by simulating context changes and the resulting reconfigurations for various parameter settings. The results demonstrate that a predictive approach considerably improves the reconfiguration process. The simulation process takes the user context automata, costs, and a set of system variants as input. During the simulation we measure two metrics: the achieved user benefit and the incurred reconfiguration costs.

We run the same experiment with different values for the parameters α and h to analyze the effect on the two metrics. For each experiment we construct a set of 200 paths of 100 hops generated according to the probabilities of a fixed user context automata (Sec. 6). We then generate a fixed number of alternatives system variants. For each variant we define randomly the eligible context states and the values of non functional properties. Each experiment consists of iterating through the context automaton according to the 200 predefined paths. At

each state, we select the variant that maximizes Eq. 6. For each chosen variant, we log the current user benefit and reconfiguration cost. Finally, we evaluate the averages of the two metrics over all paths within a single experiment configuration. Then we repeat the experiment with the same paths sequences but varying α and h values. We then compare the results for different combinations of α and h. Setting the horizon h to 1 we simulate a future unaware reconfiguration strategy. There was no difference in the resulting cost and benefit trends for cost vectors $FCost = [2\ 1]$ and $FCost = [10\ 1]$; thus we report only the results for the former.



(a) (b)

Fig. 5. Normalized average user benefit (a) and normalized average reconfiguration cost (b) with $h = 1.0$, $h = 0.5$ and $h = 0.2$ depending on utility objectives weights α.

Figure 5(a) and 5(b) show the normalized user benefit and reconfiguration across 33 experiment configurations. Figure 5(b) compares reconfiguration cost with three different values of h. Here we observe higher reconfiguration cost if we consider only the current user context state ($h = 1$). On the other hand configurations are likely to change less frequently whenever we consider future user preferences. This holds if we consider current and future preferences equally ($h = 0.5$) as well as if we give more relevance to the future state ($h = 0.2$). We can conclude that looking into the future lowers the cost. As shown in the Figure we can reduce the reconfiguration cost by regulating h independent from α. Since α represents the weight for the aggregated user benefit (Eq. 5), it increases the significance of user benefit over the cost when it is close to 1.

Although we can reduce the reconfiguration cost by exploiting future user preferences, we potentially lower user benefit at the same time. A configuration that optimizes both current and future preferences does not necessarily maximize current user benefit. Figure 5(a) presents the difference of user benefit considering the static and predictive approach with value of h. We get the best average user benefit if we consider only the current user context state ($h = 1$) while we get lower values if we consider the future user preferences ($h = 0.2$ and $h = 0.5$).

Figure 5(a) and 5(b) suggest that if we consider the current and future user preferences the relative decrease in user benefit is smaller than the reduction of reconfiguration cost. As shown in the figures there is potential user benefit without raising the cost. In addition, Figure 5(a) and 5(b) also suggest that

within the user benefit component the parameter h regulates the benefit and cost objectives. In fact setting h closer to 1 increases the cost of reconfiguration in order to increment the benefit, whereas by setting h closer to 0 we partially alleviate the cost of reconfiguration by accepting lower user benefit configurations. The difference between α and h is that the horizon has a lower impact on the objectives compared to α.

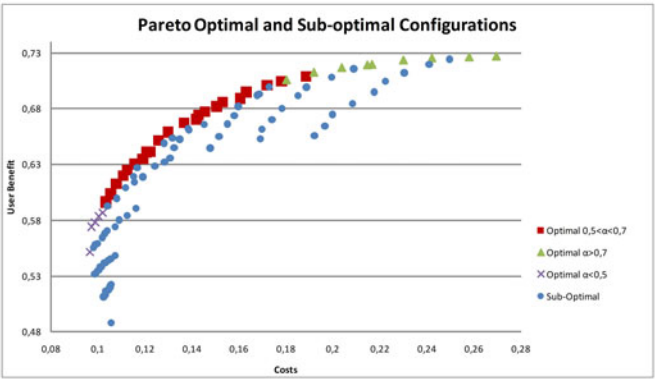


Fig. 6. Pareto-optimal and sub-optimal Configurations

Finally, we analyze the set of Pareto optimal configurations for $h = [0; 0.1; 0.2; \ldots 1]$ and $\alpha = [0; 0.1; 0.2; \ldots 1]$ for a total of 121 compared configurations. Pareto optimal points are roughly evenly distributed across h thus making it possible to select desirable values according to the specific application. We have discovered which range of α could be exploited to get most of the optimal points. In Figure 6, the Pareto optimal configurations are displayed following three different series; red squares stand for points in the range of $\alpha = [0.5; 0.7]$, crosses for optimal points for $\alpha = [0; 0.5]$ and triangles for $\alpha = [0.7; 1]$. Sub-optimal configurations are given in blue circles. As the configuration values are averaged over multiple transitions (as outlined above) also sub-optimal configurations close to the Pareto-optimal ones might me candidates. As shown in the Figure we have noted that around 50% of optimal configurations lie in the range of $\alpha = [0.5; 0.7]$. We can thus conclude that too low α values put too much weight on costs and therefore waste a lot of potential to improve user benefit. Hence, our approach is able to realize considerable user benefit even in very cost-constrained environments. Pareto optimal configurations as shown in the Figure help to decide how to set α while leaving to the designers the choice of h for specific ubiquitous applications.

The results demonstrate that predictive approaches ($h < 1$) allow the reduction of reconfiguration cost while providing an acceptable level of benefit to the user. We can conclude that our predictive approach is as good as non predictive approaches ($h = 1$) whenever we want to maximize the user benefit without focusing too much on cost. For cost-sensitive environments, a non-predictive approach fails to produce Pareto optimal points. Indeed, Pareto optimal points with $h = 1$ have high α values ($\alpha = [0.7; 1]$).

7 Related Work

Self-adaptive systems need automatic reconfiguration at run-time considering the characteristics of execution environments and the user preferences. Since it is important to make adaptation resilient to changes, it is necessary to support the decision-making process with predictive information.

In the literature there are a number of decision making mechanisms exploiting user preference to support the adaptation. Sykes et al. [18] evaluate the utility of each system component primarily by the user preferences upon each non functional property. Then the overall utility degree for each system variant is obtained as the average of each component utility. The authors define the space of adaptation strategies without considering the environment condition explicitly. The PLASTIC approach [2] considers how to exploit user preferences in performing service based adaptation. The approach performs a non-functional selection among the system variants that can be deployed in the current execution environment based on the required resources. The approach proposes a resource model that is similar to our operative context scope since it supports the definition of eligible configurations. However no predictive information is included to drive their adaptation process. In the field of service discovery, Li et al. [13] exploit a user preference model to support the service recommendation to the user. At run-time, services are checked with respect to their precondition and then they are ranked based on the user preferences upon their possible outcomes. The approach considers only a simple context model without considering future changes. Dorn and Dustdar [7] observe the behavior of multiple users to adapt the available software capabilities (i.e. features) to the preferences of the whole group. Their approach, however, does not take into account operative context constraints, neither do they apply predictive knowledge on potential future context changes.

All the mentioned approaches neither consider predictive information about the context resources nor about the user preferences. Adaptations are performed only by exploiting information on the current context.

Cheng et al. [5] extend the Rainbow evolution framework [8] in order to exploit the predictive availability of context resources to enable the adaptations. However they lack the notion of user preferences. Poladian et al. [15,16] face the problem of selecting a sequence of system variants for a predefined sequence of fixed time slots, each of which is characterized by a prediction of resource availabilities. The sequence which better fits the fixed user preferences at each time slot is selected. Also a factor of cost is introduced in order to give an increased utility to components which are already running. In addition to this work, which is heavily focused on resource prediction, we also consider the predictive context states and corresponding user preferences because of our intention to address ubiquitous environments. To the best of our knowledge there are no approaches that support system adaptation by considering run-time user preference changes, operative context changes and cost factors coherently in one formal framework. We claim that considering all these factors together promote better performance of the adaptation process.

8 Conclusions

In this paper we proposed a reconfiguration scheme for ubiquitous applications. In our approach we considered several factors including user preferences, non-functional properties, and reconfiguration cost, which may affect adaptation decisions in response to changing context. By applying feature engineering and context-awareness techniques, we quantified these factors and their aggregated effects in order to provide decision support in the face of multiple adaptation options. We conducted a series of experiments to evaluate the effectiveness of our approach with different configuration parameters. As the analysis of Pareto optimal solutions showed, our mechanism is able to maintain high user benefit while significantly reducing reconfiguration costs. Results further demonstrated that even a trade-off favoring reconfiguration costs over user benefits proves more effective than simply focusing on reducing costs alone. Based on these results, we provided guidelines to users on how to apply different parameters in order to weigh between user benefits and reconfiguration costs. The complete methodology was illustrated by means of a case study in the e-Health domain.

In the future, we will measure actual reconfiguration costs coming from the implemented case study. We also plan to integrate support for directly determining Pareto optimal solutions, thus relieving users of having to select suitable values for α and h. Furthermore, we will conduct theoretical analysis of the context state topology such as number of states, average connectivity between the states, and transition frequency, by which we expect to open up more intelligent and effective reconfiguration strategies.

Acknowledgments. This work has been partially supported by the EU IST CONNECT (http://connect-forever.eu/) No 231167 of the FET - FP7 program, the EU IST CHOReOS (http://www.choreos.eu/) No 257178 of the FP7 program, and the Austrian Science Fund (FWF) J3068-N23.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling dimensions of self-adaptive software systems. In: SEAMS, pp. 27–47 (2009)
2. Autili, M., Di Benedetto, P., Inverardi, P.: Context-aware adaptive services: The PLASTIC approach. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 124–139. Springer, Heidelberg (2009)
3. Baldauf, M., Dustdar, S., Rosenberg, F.: A survey on context-aware systems. IJAHUC 2(4), 263–277 (2007)
4. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)
5. Cheng, S.-W., Poladian, V., Garlan, D., Schmerl, B.R.: Improving architecture-based self-adaptation through resource prediction. In: SEAMS, pp. 71–88 (2009)
6. Classen, A., Heymans, P., Schobbens, P.-Y.: What's in a feature: A requirements engineering perspective. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 16–30. Springer, Heidelberg (2008)

7. Dorn, C., Dustdar, S.: Interaction-driven self-adaptation of service ensembles. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 393–408. Springer, Heidelberg (2010)
8. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37(10), 46–54 (2004)
9. Inverardi, P., Mori, M.: Feature oriented evolutions for context-aware adaptive systems. In: EVOL/IWPSE, pp. 93–97 (2010)
10. Kapitsaki, G.M., Prezerakos, G.N., Tselikas, N.D., Venieris, I.S.: Context-aware service engineering: A survey. JSS 82(8) (2009)
11. Keck, D., Kuehn, P.: The feature and service interaction problem in telecommunications systems: a survey. In: IEEE TSE (1998)
12. Krause, A., Smailagic, A., Siewiorek, D.P.: Context-aware mobile computing: Learning context-dependent personal preferences from a wearable sensor array. IEEE Trans. Mob. Comput. 5(2), 113–127 (2006)
13. Li, F., Rasch, K., Truong, H., Ayani, R., Dustdar, S.: Proactive service discovery in pervasive environments. In: ICPS, pp. 126–133 (2010)
14. Maia, P.H.M., Kramer, J., Uchitel, S., Mendonça, N.C.: Towards accurate probabilistic models using state refinement. In: ESEC/FSE, pp. 281–284 (2009)
15. Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B.R., Sousa, J.P.: Leveraging resource prediction for anticipatory dynamic configuration. In: SASO, pp. 214–223 (2007)
16. Poladian, V., Sousa, J.P., Garlan, D., Shaw, M.: Dynamic configuration of resource-aware services. In: ICSE, pp. 604–613 (2004)
17. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. TAAS 4(2) (2009)
18. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Exploiting non-functional preferences in architectural adaptation for self-managed systems. In: SAC, pp. 431–438 (2010)

Context-Bounded Model Checking of LTL Properties for ANSI-C Software

Jeremy Morse[1], Lucas Cordeiro[2], Denis Nicole[1], and Bernd Fischer[1]

[1] Electronics and Computer Science, University of Southampton, UK
{jcmm106,dan,bf}@ecs.soton.ac.uk
[2] Electronic and Information Research Center,
Federal University of Amazonas, Brazil
lucascordeiro@ufam.edu.br

Abstract. Context-bounded model checking has successfully been used to verify safety properties in multi-threaded systems automatically, even if they are implemented in low-level programming languages like ANSI-C. In this paper, we describe and experiment with an approach to extend context-bounded model checking to liveness properties expressed in linear-time temporal logic (LTL). Our approach converts the LTL formulae into Büchi-automata and then further into C monitor threads, which are interleaved with the execution of the program under test. This combined system is then checked using the ESBMC model checker. Since this approach explores a larger number of interleavings than normal context-bounded model checking, we use a state hashing technique which substantially reduces the number of redundant interleavings that are explored and so mitigates state space explosion. Our experimental results show that we can verify non-trivial properties in the firmware of a medical device.

1 Introduction

Model checking has been used successfully to verify actual software (as opposed to abstract system designs) [26, 2, 5, 1, 6], including multi-threaded applications written in low-level languages such as ANSI-C [9, 24, 18]. In *context-bounded model checking*, the state spaces of such applications are bounded by limiting the size of the program's data structures (e.g., arrays) as well as the number of loop iterations and context switches between the different threads that are explored by the model checker. This approach is typically used for the verification of safety properties expressed as assertions in the code, but it can also be used to verify properties such as the absence of global or local deadlock.

Many important requirements on the software behaviour can, however, be expressed more naturally as liveness properties in a temporal logic, for example "whenever the start button is pressed the charge eventually exceeds a minimum level". Such requirements are difficult to check directly as safety properties; it is typically necessary to add additional executable code to the program under test to retain the past state information. This amounts to the *ad hoc* introduction of a hand-coded state machine capturing (past-time) temporal formulae.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 302–317, 2011.

Here, we instead use context-bounded model checking to validate multithreaded C programs directly against (future-time) temporal formulas over the variables and expressions of the C program under test. Thus, if the C variables `pressed`, `charge`, and `min` represent the state of the button, and the current and minimum charge levels, respectively, then we can capture the requirement above with the linear-time temporal logic (LTL) formula G({pressed} -> F {charge > min}). We check these formulas following the usual approach [7, 14], albeit with a twist: we convert the negated LTL formula (the so-called *never claim* [13]) into a Büchi automaton (BA), which is composed with the program under test; if the composed system admits an accepting run, the program violates the specified requirement. We check the actual C program, however, rather than its corresponding BA. We thus convert the LTL's BA further into a separate C *monitor thread* and check all interleavings between this monitor and the program using ESBMC [9], an off-the-shelf, efficient bounded model checker for ANSI-C. We bound the execution of the monitor thread in such a way that it still searches for loops through accepting states after the program has reached its own bound. We thus consider the bounded program as the finite prefix of an infinite trace where state changes are limited to this finite prefix; this gives us a method to check both safety and liveness uniformly within the framework of bounded model checking.

Our approach avoids any imprecision from translating the C program into a BA, but the monitor has to capture transient behaviour internal to the program under test. The monitor and the program communicate via auxiliary variables reporting the truth values of the LTL formula's embedded expressions. Our tool automatically inserts and maintains these and also uses them to guide ESBMC's thread exploration. Nevertheless, our approach requires that the underlying bounded model checker must be able to accommodate deep interleavings of the monitor thread with the program threads. We have thus implemented a state hashing strategy which eliminates multiple examinations of identical parts of the state space and improves ESBMC's performance.

Our paper makes three main contributions. First, it describes the first mechanism, to the best of our knowledge, to verify LTL properties against an unmodified C code base. Second, since ESBMC is a symbolic model checker based on the satisfiability modulo theory approach, it also describes the first symbolic LTL model checker that does not use binary decision diagrams (BDDs). Third, it is the first application of the concept of state hashing to symbolic model checking.

2 From LTL to Monitor Threads

2.1 Linear-Time Temporal Logic

Linear-time temporal logic (LTL) is a commonly used specification logic in model checking [3, 15, 16], which extends propositional logic by including temporal operators. The primitive propositions of our LTL are side-effect-free boolean C expressions over the global variables of the C program.

Definition 1. *Our LTL syntax is defined over primitive propositions, logical operators and temporal operators as follows:*

$$\begin{aligned}\phi ::= &\texttt{true} \mid \texttt{false} \mid \{p\} \mid \texttt{!}\phi \mid \phi_1 \texttt{ \&\& } \phi_2 \\ &\mid \phi_1 \texttt{ || } \phi_2 \mid \phi_1 \texttt{ -> } \phi_2 \mid \texttt{(}\phi\texttt{)} \\ &\mid \texttt{F}\phi \mid \texttt{G}\phi \mid \phi_1\texttt{U}\phi_2 \mid \phi_1\texttt{R}\phi_2\end{aligned}$$

The logical operators include *negation* (`!`), *conjunction* (`&&`), *disjunction* (`||`) and *implication* (`->`). The temporal operators are "in some future state (eventually)" (`F`), "in all future states (globally)" (`G`), "until" (`U`) and "release" (`R`). Here, p is a side-effect-free boolean C expression, and $\phi_1\texttt{U}\phi_2$ means that ϕ_1 must hold continuously until ϕ_2 holds; ϕ_2 must become true before termination. The other temporal operators can be defined in terms of `U`, as shown below.

We are only interested in temporal formulae which are closed under stuttering; following Lamport [20], we thus do not provide an explicit "next state" operator `X`. Our LTL expressions are thus insensitive to refinements of the timestep to intervals less than those required to capture the ordering of changes in the global state. The timesteps only need to be sufficiently fine to resolve any potentially dangerous interleavings of the program. For efficiency reasons we assume interleavings only at statement boundaries and assume sequential consistency [19], but options to ESBMC allow us also to use a finer-grained analysis to detect data races arising from interleavings within statements.

We use a *linear-time* rather than a *branching-time* approach and thus there are no explicit path quantifications (i.e., CTL*-style operators `A` and `E`). There is, however, an implicit universal quantification over all possible interleavings and program executions. In this formulation we have the following identities:[1]

$$\begin{aligned}\phi &= \texttt{false U } \phi \\ \texttt{F } \phi &= \texttt{true U } \phi \\ \texttt{G } \phi &= \texttt{!F !}\phi \\ \phi_1 \texttt{ R } \phi_2 &= \texttt{! (! } \phi_1 \texttt{ U ! } \phi_2 \texttt{)}\end{aligned}$$

We interpret a possibly multi-threaded C program as a Kripke structure whose state transitions are derived from the possibly interleaved execution sequence of C statements and whose valuations are the possible values of the program's global variables. Since we have implemented a *bounded* model checker, all (bounded) programs will either deadlock or terminate in finite time. We use a separate run of ESBMC to assure deadlock freedom and formally extend the behaviour of deadlock-free programs with an infinite sequence of timesteps which leave all global variables unchanged. Thus every program that is scheduled generates an infinite sequence of states. We finally describe the desired liveness property ϕ as an LTL expression in the above syntax and then check that there are no possible infinite sequences of program states for which `!` ϕ holds.

2.2 Büchi Automata

Büchi automata (BA) are finite-state automata over infinite words first described by Büchi [4]. We follow Holzmann's presentation [14] and define a BA as a tuple

[1] This differs from the notation of [21], which has `X` ϕ = `false U` ϕ

$B = (S, s_0, L, T, F)$ where S is a finite set of states, $s_0 \in S$ the initial state of the BA, L a finite set of labels, $T \subseteq (S \times L \times S)$ a set of state transitions and $F \subseteq S$ a set of final states. B may be deterministic or non-deterministic. A *run* is a sequence of state transitions taken by B as it operates over some input. A run is accepted if B passes through an accepting state $s \in F$ infinitely often along the run.

A number of algorithms exist for converting an LTL formula to a BA accepting a program trace [11, 25, 12]. We use the `ltl2ba` [11] algorithm and tool, which produces smaller automata than some other algorithms. Figure 1 illustrates the BA produced from the LTL formula in the introduction. Input symbols are propositions composed from the primitive C-expressions.

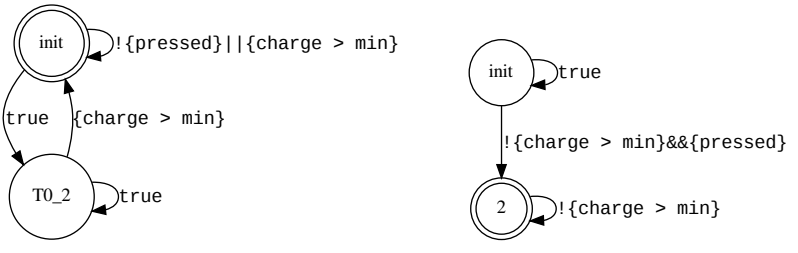


Fig. 1. The left BA accepts the example from the introduction, `G({pressed} -> F {charge > min})`. The right BA is its negation, used for the never claim in our monitor.

2.3 Monitor Threads

In our context, a monitor is some portion of code that inspects a program state and verifies that it satisfies a given property, failing an assertion if this is not the case. A monitor thread is a monitor that is interleaved with the execution of the program under test. This allows it to verify that the property holds at each particular interleaving of the program, detecting any transient violations between program interleavings.

Monitor threads have been employed in SPIN to verify LTL properties against the execution of a program [14]. A non-deterministic BA representing the negation of the LTL property, the so-called never claim, is implemented in a Promela process which will accept a program trace that violates the original LTL property. SPIN then generates execution traces of interleavings of the program being verified, and for each step in each trace runs the Promela BA. This is called a *synchronous interleaving*. In this work we employ a similar mechanism to verify LTL properties by interleaving the program under verification with a monitor thread, detailed in Section 3.2.

3 Model-Checking LTL Properties with ESBMC

3.1 ESBMC

ESBMC is a context-bounded model checker for embedded ANSI-C software based on SMT solvers, which allows the verification of single- and multi-threaded

software with shared variables and locks [10, 9]. ESBMC supports full ANSI-C, and can verify programs that make use of bit-level, arrays, pointers, structs, unions, memory allocation and fixed-point arithmetic. It can reason about arithmetic under- and overflows, pointer safety, memory leaks, array bounds violations, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions.

In ESBMC, the program to be analyzed is modelled as a state transition system $M = (S, R, S_0)$, which is extracted from the control-flow graph (CFG). S represents the set of states, $R \subseteq S \times S$ represents the set of transitions (i.e., pairs of states specifying how the system can move from state to state) and $S_0 \subseteq S$ represents the set of initial states. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location of the CFG to *pc*. We identify each transition $\gamma = (s_i, s_{i+1}) \in R$ between two states s_i and s_{i+1} with a logical formula $\gamma(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter and the program variables.

Given the transition system M, a safety property ϕ, a context bound C and a bound k, ESBMC builds a reachability tree (RT) that represents the program unfolding for C, k and ϕ. We then derive a verification condition(VC) ψ_k^π for each given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ_k^π is satisfiable if and only if ϕ has a counterexample of depth k that is exhibited by π. ψ_k^π is given by the following logical formula:

$$\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i) \tag{1}$$

Here, I characterizes the set of initial states (i.e. $I(s_0) \leftrightarrow s_0 \in S_0$) of M and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents executions of M of length i and ψ_k^π can be satisfied if and only if for some $i \leq k$ there exists a reachable state along π at time step i in which ϕ is violated. ψ_k^π is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. If ψ_k^π is satisfiable, then ϕ is violated along π and the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counter-example. A counter-example for a property ϕ is a sequence of states $s_0, s_1, \ldots, s_i$ with $s_0 \in S_0$, $s_i \in S$, and $\gamma\,(s_j, s_{j+1})$ for $0 \leq j < i$. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we can define $\psi_k = \bigwedge_\pi \psi_k^\pi$ and use this to check all paths. ESBMC combines symbolic model checking with explicit state space exploration; in particular, it explicitly explores the possible interleavings (up to the given context bound) while it treats each interleaving itself symbolically.

3.2 Checking LTL Properties against a C Program

As discussed in Section 2.3, an LTL property can be verified against a program by interpreting the corresponding BA over the program states along the execution

path. We apply this approach to a C code base by implementing the BA in C which is then executed as a monitor thread, interleaved with the execution of the program. This involves three technical aspects: the conversion of the BA to C, the interaction of the monitor thread with the program under test, and the control of the interleavings.

The monitor thread itself is not interleaved with the program in a special manner as in SPIN, but instead is treated as any other program thread. We use a counting mechanism to ensure that the BA thread operates on the program states in the right sequential order. This approach can be slower than a synchronous composition, but it requires no fundamental changes to the way that ESBMC operates as it uses only existing features.

Implementing a Büchi Automata in C. We follow the SPIN approach of inverting the LTL formula being verified so that the BA accepts execution traces which violate the original formula. We then modified the `ltl2ba` tool to convert its usual Promela output to C, which uses some ESBMC built-ins.

The C implementation of the BA (see Figure 2 for the code corresponding to the BA in Figure 1) consists of an infinite loop (unrolled an appropriate number of times, see below) around a switch statement on the state variable that branches to code which atomically (lines 18, 46) evaluates the target state of the transition. Non-deterministic behaviour is simulated by attempting all transitions from a state non-deterministically (lines 24, 27, 36), after which guards on each transition evaluate whether the transition can be taken (lines 25, 28, 37). These guards use ESBMC's *assume* statements, which ensure that transitions not permitted by the current state of the program under test are not explored.

To determine when the BA has accepted a program trace, we first await a time where the program has terminated — given that we operate in the context of bounded model checking this is guaranteed as any infinite loop is unrolled only to the length of the bound. Detection of thread deadlock is already performed by ESBMC. The BA loop is run a second time with the final program state as input, recording the number of times it passes through each state (lines 44-45). If a loop through an accepting state exists it will be visited more than once, triggering an assertion showing that the BA accepted the trace. This technique places a constraint on the unwinding bound of the BA loop, that it is sufficient for any such loop to be detected. Setting this bound to twice the number of states in the BA permits it to pass through every state twice on the largest possible loop.

This acceptance criteria operates on the principle that, should some program state need to be reached for the LTL formula to hold, then it needs to have happened by the time that the program bound has been reached. This can be an overapproximation of the program being verified, as there can be circumstances where that program state could be reached if the program bound were higher.

We strictly control where interleavings may occur in the BA to ensure its soundness. The evaluation of the next state to transition to is executed atomically, ensuring that the BA always perceives a consistent view of program state. We also yield execution (line 17) before the BA inputs a program state to force new interleavings to be explored. Certain utility functions are provided to allow

a program test harness to start the BA and check for acceptance at the end of execution (not shown).

Interacting with the Existing Code Base. LTL formulas allow verification engineers to describe program behaviour with propositions about program state. To describe the state of a C program, we support the use of C expressions as propositions within LTL formulas. Any characters in the formula enclosed in braces are interpreted as a C expression and as a single proposition within LTL. The expression itself may use any global variables that exist within the program under test as well as constants and side-effect free operators. The expression must also evaluate to a value that can be interpreted as a boolean under normal C semantics. For example, the following liveness property verifies that a certain input condition results in a timer increasing:

```
!G(({pressed_key == 4} && {mstate == 1}) -> F{stime > ref_stime})
```

and the following safety property checks a buffer bound condition:

```
!G({buffer_size != 0} -> {next < buffer_size})
```

Within the BA (Figure 2) these expressions are required for use in the guards preventing invalid transitions from being explored. We avoid using the expressions directly in the BA; instead ESBMC searches the program under verification for assignments to global variables used in a C expression, then inserts code to update a Boolean variable corresponding to the truth of the expression (lines 2, 4) immediately after the symbol is assigned to. If multiple propositions update on the same variable, re-evaluations are executed atomically. All modifications are performed on ESBMC's internal representation of the program and do not alter the code base.

Synchronous Interleaving. An impediment of operating the monitor thread containing the BA as a normal program thread is that it is not always guaranteed to receive a consistent series of input states—that is, it is entirely possible for the BA not to be scheduled to run after an event of interest, and thus not perform a state transition it should have. This is clearly an invalid action when one considers it in terms of a BA skipping an input symbol. The full exploration of state space also guarantees we will explore the interleavings where this is not the case.

To handle this the BA discards interleavings where the propositions have changed more than once but the BA has not had opportunity to run and interpret them (lines 19–21 in Figure 2). We maintain a global variable (line 10) counting the number of times that the C expressions forming propositions in the LTL formula have been re-evaluated, keep a corresponding counter (line 9, 21) within the BA, and use an assume statement to only consider traces where the global counter has changed at most once since the last time the BA ran.

```
1  char __ESBMC_property___cexpr_0[] = "pressed";
2  bool __cexpr_0_status;
3  char __ESBMC_property___cexpr_1[] = "charge > min";
4  bool __cexpr_1_status;
5
6  typedef enum {T0_init, accept_S2 } ltl2ba_state;
7  ltl2ba_state state = T0_init;
8  unsigned int __visited_states[2];
9  unsigned int __transitions_seen;
10 extern unsigned int __transitions_count;
11
12 void ltl2ba_fsm(bool state_stats) {
13   unsigned int choice;
14   while (1) {
15     choice = nondet_uint();
16     /* Force a context switch */
17     __ESBMC_yield();
18     __ESBMC_atomic_begin();
19     __ESBMC_assume(__transition_count <=
20                    __transitions_seen + 1);
21     __transitions_seen = __transition_count;
22     switch(state) {
23     case T0_init:
24       if (choice == 0) {
25         __ESBMC_assume((1));
26         state = T0_init;
27       } else if (choice == 1) {
28         __ESBMC_assume((!__cexpr_1_status &&
29                         __cexpr_0_status));
30         state = accept_S2;
31       } else {
32         __ESBMC_assume(0);
33       }
34       break;
35     case accept_S2:
36       if (choice == 0) {
37         __ESBMC_assume((!__cexpr_1_status));
38         state = accept_S2;
39       } else {
40         __ESBMC_assume(0);
41       }
42       break;
43     }
44     if (state_stats)
45       __visited_states[state]++;
46     __ESBMC_atomic_end();
47   }
48 }
49 return;
```

Fig. 2. C implementation of the Büchi automaton for the formula `!G({pressed} -> F {charge > min})`

4 Optimizing State Space Exploration

The context-bounded approach has proven to be effective for model checking multi-threaded software, with a small number of context switches allowing us to explore much of the system behaviour. Our approach to verifying programs against LTL properties requires frequent context-switching between monitor and program threads, which makes a greater context bound necessary. We thus implemented *state hashing* in ESBMC to reduce the number of redundant interleavings and thus the state space to be explored. This is also the first work to our knowledge where state hashing has been used in conjunction with symbolic model checking.

The driving force behind our approach to state hashing is that during the exploration of the RT of multi-threaded software many interleavings pass through identical RT nodes, i.e., nodes that represent the same global and thread-local program states, respectively, and differ only in the currently active thread. Only one of these nodes need be explored, as the reachability subtrees of all other nodes will be identical. As an example, consider a simple multithreaded C program shown in Figure 3 and its corresponding RT shown in Figure 4. The RT consists of the nodes ν_0 to ν_{16}, where each node is defined as a tuple $\nu = (A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n)_i$ for a given time step i. Here, A_i represents the currently active thread, C_i the context switch number, and s_i the current (global and local) state. Further, for each of the n threads, l_i^j represents the current location of thread j and G_i^j represents the control flow guards accumulated in thread j along the path from l_0^j to l_i^j (although these are not shown in Figure 4). Notice how the transitions originating from node ν_1 as those originating from ν_7, produce the same program states. When we explore the node ν_7, we can simply eliminate the transitions that originate from it — provided that we realise that we have already explored another identical RT node. We thus maintain a set of hashes representing the states of RT nodes that we have already explored.

```
1  #include <pthread.h>
2  int x=0, y=0;
3  void t1(void* arg) { x++; }
4  void t2(void* arg) { x++; }
5  void t3(void* arg) { y++; }
6  int main(void) {
7    pthread_t id1, id2, id3;
8    pthread_create(&id1,NULL,t1,NULL);
9    pthread_create(&id2,NULL,t2,NULL);
10   pthread_create(&id3,NULL,t3,NULL);
11   return 0;
12 }
```

Fig. 3. A simple multi-threaded C program

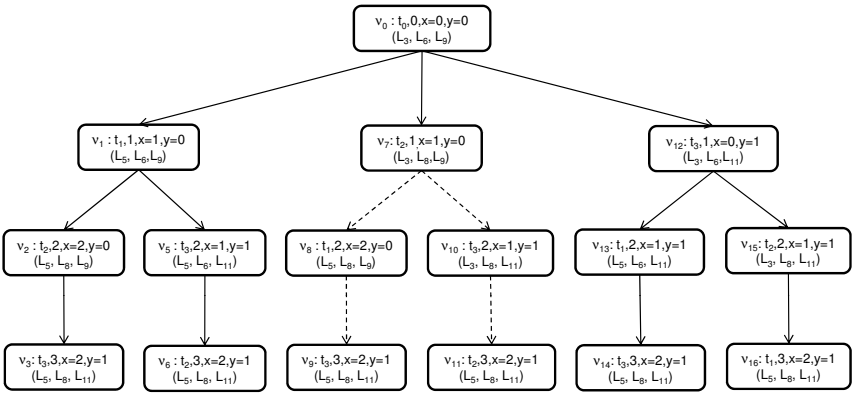


Fig. 4. Reachability tree for the program in Figure 3. Dashed edges represent transitions that can be eliminated by the state hashing technique.

4.1 Hashing Symbolic States

In explicit-state model checking, state hashing takes a state vector containing the current *values* of all program variables, and applies a hash function to compute a value that can then be stored to indicate a particular state has been explored.

State hashing, unfortunately, is not so simple for symbolic model checking, as the state vector does not simply contain values but is defined symbolically by the calculations and constraints that make up the variable assignments in the underlying static single assignment (SSA) form of the program. We thus implement a two-level hashing scheme: we use a node-level hash that represents a particular RT node, and a variable level hash that represents the constraints affecting a particular assignment to a variable. Since each new RT node can only change the (symbolic) value of at most one variable, the two-level hashing scheme reduces the computational effort, as it allows us to retain the hash values of the unchanged variables.

The node-level hash is created by taking the variable-level hashes of all variables in the current node and concatenating them, together with the program counter values of all existing threads, into a single data vector. This vector is then fed to a hashing function. Variable-level hashes are more complex. For each assignment encountered in the RT exploration we calculate a hash of the right hand side expression and record it with the left hand side variable name. This hash is created by serialising each operator and value in the expression to a data representation (i.e., a series of bytes) into a vector, which is then hashed.

For example, Figure 3 contains several assignments to the global variable `x` using the `++`-operator (converted to an addition internally). ESBMC automatically performs constant propagation and effectively converts the example to an explicit state check. We represent the first serialised increment expression as the text: "(+,(constant(0)),(constant(1)))" This demonstrates one of the simplest encodings of data possible with this method. Any set of operations on constant values

can also be expressed in this manner. Such expressions are, however, not yet symbolic—to support this we represent nondeterminstic values with a prefix and unique identifier. We also represent the use of existing variables in expressions with its current variable hash. To demonstrate this, reconsider Figure 3 and assume `x` is initialised to a nondeterministic value. The expressions representing the two increments of the `x` variable then become: "(+,(nondet(1)),(constant(1)))" and "(+,(hash(#1)),(constant(1)))" where #1 represents the hash value of the first expression. Significantly, no thread specific data is encoded in this representation, meaning that the same serialised representation is produced for whichever order of threads increments `x`. Thus the hash of any assignment is a direct product of all nondeterministic inputs, constant values and operators that represent the constraints on the assignment.

This method is limited, however, by the ordering of assignments—if the original example in Figure 3 had instead a thread that increased the `x` variable by 2, and another that increased `x` by 3, then at the end of execution the variable hash of `x` would be different depending on the thread ordering, even though the effective constraints on `x` for every interleaving are identical. This also affects arrays (including the heap, which is modelled as an array) and unions.

4.2 Selection of Hash Function

As hashing is a lossy abstraction of a tree node, we risk computing two identical hashes for two distinct nodes. Should this occur one node will be successfully explored and its hash stored; when the other is explored we will discover its hash in the visited states set, and incorrectly assume it has already been visited. This would cause an unexplored portion of the state space to be discarded.

We require a hash function that takes a stream of characters as input (serialised expressions) and produces a small output. We simply chose SHA256 [23] hashes due to its relatively large output bitwidth (compared to other hash functions) and its certification for use in cryptographic applications, aspects that assure us the likelihood of collisions is extremely low.

4.3 Comparison with Partial Order Reductions

Kahlon et al. [17] have developed a partial order reduction technique which, at the expense of additional constraints, permits the optimal elimination of unnecessary interleaves. They demonstrate their technique by testing the (unsatisfied) property *at some time all philosophers are eating* in a *dining philosophers* model. They present two versions of their technique, PPOR which is optimal for two threads and MPOR which, at the expense of further complexity, is optimal for any number of threads. Our state hashing (SHASH) technique, on the other hand, avoids complicating the SMT inputs, but at the expense of evaluating some redundant interleaves. We compare the speedups achieved by PPOR, MPOR and our state hashing in table 1. Note that our results are achieved with a dining philosophers implementation in C on our own implementation of ESBMC and the absolute times are thus not comparable with the Kahlon et al. model.

Table 1. Speedup of each optimization technique over unoptimized performance

Speedup			
Philosophers	MPOR	PPOR	SHASH
2	1	2	0.8
3	20	16	4.4
4	9.3	1.1	9.5

5 Experimental Evaluation

We have tested the work described here against a series of properties defining the behaviour of a pulse oximeter firmware, which is a piece of sequential software that is responsible for measuring the oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method [8]. The firmware of the pulse oximeter device is composed of device drivers (i.e., display, keyboard, serial, sensor, and timer) that contain hardware-dependent code, a system log component that allows the developer to debug the code through data stored on RAM memory, and an API that enables the application layer to call the services provided by the platform. The final version of the pulse oximeter firmware consists of approximately 3500 lines of ANSI-C code and 80 functions.

Here we report the results of verifying the pulse oximeter code against five liveness properties of the general form `G(p -> F q)` i.e., whenever an enabling condition `p` has become true, then eventually the property `q` is enabled. We formulated a test harness for each portion of the firmware being tested to simulate the activity that the LTL property checks. We then invoked ESBMC with a variety of loop unwind and context switch bounds to determine the effectiveness of state hashing. We also ran these tests against versions of the firmware deliberately altered to not match the LTL formula to verify that failing execution traces are identified.

All tests were run on the Iridis 3 compute cluster[2] with a memory limit of 4Gb and time limit of 4 hours to execute. The results are summarized in Table 2. Here, #L column contains the line count of the source file for the portion of firmware being tested, P/F records whether the test is expected to Pass or Fail, k the loop unwinding bound and C the context-bound specified for the test.

We then report the results for the original version of ESBMC[3] and the version with state hashing, respectively. For each version, we report the verification time in seconds, the number #I and #FI of generated and failing interleavings, respectively, and the result. Here, + indicates that ESBMC's result is as expected (i.e. all its interleavings were verified successfully if the test is expected to pass, and at least one interleaving is found to violate the LTL property if the test is expected to fail), while − indicates a false negative (i.e., ESBMC fails to find an

[2] 1008 Intel Nehalem compute nodes, each with two 4-core processors, up to 45Gb of RAM, and InfiniBand communications. Each test used only one core of one node.

[3] v1.16, available from www.esbmc.org

Table 2. Results from testing LTL properties against pulse oximeter firmware

					Original run			With state hashing		
Test name	#L	P/F	k	C	Time (s)	#I / #FI	Result	Time (s)	#FI / #I	Result
start_btn	856	Pass	1	20	207	7764/0	+	67	2245/0	+
		Pass	1	40	199	7764/0	+	71	2245/0	+
		Pass	2	20	2740	55203/0	+	479	11409/0	+
		Pass	2	40	14400	0/0	TO	14400	0/0	TO
		Fail	1	20	236	6719/231	+	81	1919/91	+
		Fail	1	40	244	6719/231	+	94	1919/91	+
		Fail	2	20	1344	29840/0	−	299	6911/0	−
		Fail	2	40	N/A0	0/0	MO	N/A	0/0	MO
up_btn	856	Pass	1	20	78	3775/0	+	32	1385/0	+
		Pass	1	40	83	3775/0	+	37	1385/0	+
		Pass	2	20	2777	102566/0	+	898	41389/0	+
		Pass	2	40	14400	0/0	TO	6012	111335/0	+
		Fail	1	20	90	3775/0	−	35	1385/0	−
		Fail	1	40	82	3775/0	−	33	1385/0	−
		Fail	2	20	2743	102564/0	−	914	40938/0	−
		Fail	2	40	14400	0/0	TO	4832	69275/3422	+
keyb_start	50	Pass	1	20	9668	92795/0	+	4385	49017/0	+
		Pass	1	40	9767	92795/0	+	4489	49017/0	+
		Pass	2	20	14400	0/0	TO	14400	0/0	TO
		Pass	2	40	14400	0/0	TO	14400	0/0	TO
		Fail	1	20	9795	92795/321	+	4836	49017/321	+
		Fail	1	40	9924	92795/321	+	4914	49017/321	+
		Fail	2	20	14400	0/0	TO	14400	0/0	TO
		Fail	2	40	14400	0/0	TO	14400	0/0	TO
baud_conf	178	Pass	1	20	18	485/0	+	16	419/0	+
		Pass	1	40	17	485/0	+	16	419/0	+
		Pass	2	20	2440	39910/0	+	971	17500/0	+
		Pass	2	40	2635	39910/0	+	1078	17500/0	+
		Fail	1	20	18	485/56	+	17	419/56	+
		Fail	1	40	18	485/56	+	16	419/56	+
		Fail	2	20	2583	39910/2002	+	1010	17500/880	+
		Fail	2	40	2851	39910/2002	+	1139	17500/880	+
serial_rx	584	Pass	1	20	334	5454/0	+	194	3108/0	+
		Pass	1	40	324	5454/0	+	212	3108/0	+
		Pass	2	20	10959	62332/0	+	4494	29257/0	+
		Pass	2	40	14400	0/0	TO	70	627/0	+
		Fail	1	20	215	3286/273	+	137	2030/257	+
		Fail	1	40	211	3286/273	+	135	2030/257	+
		Fail	2	20	3768	20917/0	−	1846	11388/0	−
		Fail	2	40	14400	0/0	TO	14400	0/0	TO

existing violation of the LTL property). TO indicates the check ran out of time and MO indicates it ran out of memory.

We first observe that ESBMC is generally able to verify all positive test cases, although it sometimes times out with increasing bounds. The situation is less clear for the tests designed to fail. Here, smaller unrolling and context switch bounds allow to correctly identify failing interleavings, but are sometimes not sufficient to expose the error (e.g., up_btn), and small increases in the unrolling bound generally require larger increases in the context bounds to expose the error, leading to time-outs or memory-outs in most cases. State hashing, however, improves the situation, and allows us to find even deeply nested errors.

Comparing the figures between tests performed with state hashing and those without, we see the total number of interleavings generated is often significantly reduced by state hashing. Out of all tests that completed the median reduction was 56%, the maximum 80% and minimum 13%. In all cases the use of state hashing reduced the amount of time required to explore all reachable states.

6 Related Work

SPIN [13] is a well known software model checker that operates on concurrent program models written in the Promela modelling language. It operates with explicit state and uses state hashing to reduce the fraction of state space it explores. SPIN also allows users to specify a LTL formula to verify against the execution of a model by using BA in a similar manner to this work. While SPIN is well established as a model checker the requirement to re-model codebases in Promela can be time consuming.

Java PathFinder is a Java Virtual Machine (JVM) that performs model checking on Java bytecode. It also operates with explicit state and uses *state matching* to reduce the search space, but can also operate symbolically for the purpose of test generation and coverage testing. Verification of LTL formula can be achieved with the JPF-LTL extension which also uses BA in a similar manner to this work, but "listens" to the execution of Java bytecode traces within the JVM rather than interleaving the BA with the program under test, resulting in a synchronous interleaving.

Partial order reductions (examined in section 4.3) improve the efficiency of software model checking significantly by identifying redundant interleavings of threads that need not be explored, although typically at the expense of a complex static analysis. Recent work [17] reduces this overhead and delivers a demonstrably optimal number of interleavings.

7 Conclusions and Future Work

Context-bounded model checking has been used successfully to verify multi-threaded applications written in low-level languages such as ANSI-C. The approach has, however, largely been confined to the verification of safety properties. In this paper, we have extended the approach to the verification of liveness

properties given as LTL formulas against an unmodified code base. We follow the usual approach of composing the BA for the never claim with the program, but work at the actual code level. We thus convert the BA further into a separate C monitor thread and check all interleavings between this monitor and the program using ESBMC. We use a state hashing scheme to handle the large number of interleavings and counter state space explosion.

The initial results are encouraging, and we were able to verify a number of liveness properties on the firmware of a medical device; in future work, we plan to extend the evaluation to a larger code base and wider variety of properties. The state hashing proved to be very useful, cutting verification times by about 50% on average. We expect that an improved hashing implementation (e.g., removing serialisation) will yield even better results. Our approach does still have some limitations. The (indiscriminate) composition of the monitor thread with the program under test leads to a very large number of interleavings that need to be explored, despite the improvements that the state space hashing provides. We thus plan to implement a special thread scheduling algorithm in ESBMC that schedules the monitor after changes to the observed variables and so achieves the effect of synchronous composition.

Acknowledgments. The second author would like to thank the support received from the Nokia Institute of Technology (INdT).

References

[1] Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The spec# programming system: Challenges and directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)

[2] Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9(5-6), 505–525 (2007)

[3] Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2(5), 1–64 (2006)

[4] Büchi, J.: On a Decision Method in Restricted Second Order Arithmetic. Studies in Logic and the Foundations of Mathematics, vol. 44, pp. 1–11 (1966)

[5] Clarke, E., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

[6] Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. FMSD 25, 105–127 (2004)

[7] Clarke, E., Lerda, F.: Model Checking: Software and Beyond. J. Universal Computer Science 13(5), 639–649 (2007)

[8] Cordeiro, L., et al.: Agile development methodology for embedded systems: A platform-based design approach. In: ECBS, pp. 195–202 (2007)

[9] Cordeiro, L., Fischer, B.: Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking. To appear in ICSE (2011)

[10] Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE, pp. 137–148 (2009)
[11] Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
[12] He, A., Wu, J., Li, L.: An Efficient Algorithm for Transforming LTL Formula to Büchi Automaton. In: ICICTA, pp. 1215–1219 (2008)
[13] Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
[14] Holzmann, G.: The SPIN Model Checker - primer and reference manual. Addison-Wesley, Reading (2004)
[15] Huth, M., Ryan, M.: Logic in Computer Science: modelling and reasoning about systems. Cambridge University Press, Cambridge (2004)
[16] Jonsson, B., Tsay, Y.: Assumption/guarantee specifications in linear-time temporal logic. Theor. Comput. Sci 167(1&2), 47–72 (1996)
[17] Kahlon, V., Wang, C., Gupta, A.: Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 398–413. Springer, Heidelberg (2009)
[18] Lahiri, S., Qadeer, S., Rakamaric, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
[19] Lamport, L.: A new approach to proving the correctness of multiprocess programs. TOPLAS 1(1), 84–97 (1979)
[20] Lamport, L.: What Good is Temporal Logic? Information Processing 83, 657–668 (1983)
[21] McMillan, K.: Symbolic Model Checking, vol. 1003, p. 15. Kluwer, Dordrecht (1993)
[22] Muchnick, S.: Advanced compiler design and implementation. Morgan Kaufmann, San Francisco (1997)
[23] Secure Hash Standard. National Institute of Standards and Technology. Federal Information Processing Standard 180-2 (2002)
[24] Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 82–97. Springer, Heidelberg (2005)
[25] Rozier, K., Vardi, M.: LTL Satisfiability Checking. STTE 12, 123–137 (2010)
[26] Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003)

Modular Modelling of Software Product Lines with Feature Nets*

Radu Muschevici, José Proença, and Dave Clarke

DistriNet & IBBT, Dept. Computer Science, Katholieke Universiteit Leuven, Belgium
{radu.muschevici,jose.proenca,dave.clarke}@cs.kuleuven.be

Abstract. Formal modelling and verification are critical for managing the inherent complexity of systems with a high degree of variability, such as those designed following the software product line (SPL) paradigm. SPL models tend to be large—the number of products in an SPL can be exponential in the number of features. Modelling these systems poses two main challenges. Firstly, a modular modelling formalism that scales well is required. Secondly, the ability to analyse and verify complex models efficiently is key in order to ensure that all products behave correctly. The choice of a system modelling formalism that is both expressive and well-established is therefore crucial. In this paper we show how SPLs can be modelled in an incremental, modular fashion using a formal method based on Petri nets. We continue our work on *Feature Petri Nets*, a lightweight extension to Petri nets, by presenting a framework for modularly constructing Feature Petri Nets to model SPLs.

1 Introduction

The need to tailor software applications to varying requirements, such as specific hardware, markets or customer demands, is growing. If each application variant is maintained individually, the overhead of managing all the variants quickly becomes infeasible [20]. Software Produce Line Engineering (SPLE) is seen as a solution to this problem. A Software Product Line (SPL) is a set of software products that share a number of core properties but also differ in certain, well-defined aspects. The products of an SPL are defined and implemented in terms of *features*, which are subsequently combined to obtain the final software products. The key advantage hereby over traditional approaches is that all products can be developed and maintained together. A challenge for SPLE is to ensure that all products meet their specifications without having to check each product individually, by checking the product line itself. This raises the need for novel SPL-specific formalisms to model SPLs and reason about and verify their properties.

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu), and the K.U.Leuven BOF-START project STRT1/09/031 Designer-TypeLab.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 318–333, 2011.

The main contribution of this paper is a modular modelling framework for specifying the behaviour of software product lines. We use *Feature Petri Nets* [18], or *feature nets* (FN) for short, as the modelling formalism. Feature nets are a Petri net extension that enables the specification of the behaviour of an entire software product line (a set of systems) in one single model. The behaviour of an FN is conditional on the features appearing in the product line. The paper makes three contributions. Firstly, it presents a variant of feature nets that improves upon their original definition [18]. Secondly, it gives a technique for constructing larger feature nets from smaller ones to model the addition of new features to an SPL. Thirdly, it provides correctness criteria for ensuring that the resulting composition preserves the behaviour of the original model(s).

Organisation: The next section motivates the need for feature nets. Section 3 formally introduces feature nets. Section 4 details an approach at modular modelling with FN, and Section 5 discusses behaviour preservation. Section 6 discusses related work. Section 7 presents our conclusions and future work.

2 Software Product Line Modelling Challenge

We illustrate the modelling challenge in software product line engineering (SPLE) using an example of a software product line of coffee vending machines. A manufacturer of coffee machines offers products to match different demands, from the basic black coffee dispenser to more sophisticated machines, such as ones that can add milk or sugar, or charge a payment for each serving. Each machine variant needs to run software adapted to the selected set of hardware features. Such a family of different software products that share functionality is typically developed using an SPLE approach, as one piece of software structured along distinct features. This has major advantages in terms of code reuse, maintenance overhead, and so forth. The challenge is ensuring that the software works appropriately in all product configurations.

Petri nets [17] are used to specify how systems behave. Fig. 1 presents an example of a Petri net for a coffee machine. This has a capacity for n coffee servings; it can brew and dispense coffee, and refill the machine with new coffee supplies. If we now add an optional *Milk* feature, so that the machine can also

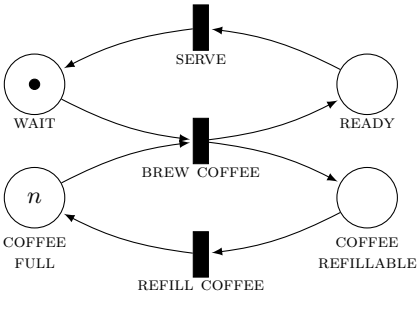


Fig. 1. Petri net model of a basic coffee machine that can only dispense coffee

add milk to a coffee serving, we need to adapt the Petri net, not only to include the functionality of adding milk, but also to be able to control whether or not this feature is present in the resulting software product.

To address the challenge of modelling a software product line with multiple features, which may or may not be included in any given product, we introduced *Feature Petri Nets* [18]. Feature Petri Net transitions are annotated with *application conditions* [21], which are propositional formula over features that reflect when the transition is enabled. In this paper we use a variation of Feature Petri Nets in which application conditions are placed on arcs, rather than transitions, called *arc-labelled Feature Petri Nets*, though we shall just call them feature nets. One advantage of feature nets is that they enable the *superposition* of the behaviour of the various products (given by different feature selections) in the same model.

Fig. 2 exemplifies a feature net of a coffee machine with a milk reservoir. It considers a product line whose products are over the set of features $\{Coffee, Milk\}$, where *Coffee* is compulsory and *Milk* is optional. The application condition above each arc reflects that the arc is present only when the condition evaluates to true. Only then does the arc affect behaviour. If the condition is false, the arc has no effect on behaviour. Consequently, the three transitions on the left-hand side can only fire when the *Coffee* feature is present; the two transitions on the right-hand side can be taken only when the feature *Milk* is present. Observe that the restriction of this example net to the transitions that can fire for feature selection $\{Coffee\}$ is exactly the Petri net in Fig. 1, after removing unreachable places.

3 Feature Nets

A feature net (FN) [18] is a Petri net variant used to model the behaviour of an entire software product line. For this purpose feature nets have *application conditions* [21] attached to their arcs. An application condition is a propositional logical formula over a set of features, describing the feature combinations to

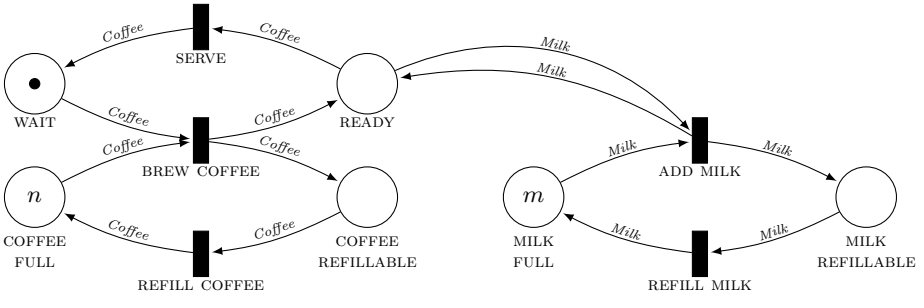


Fig. 2. Feature net of the product line $\{\{Coffee\}, \{Coffee, Milk\}\}$. Each arc has an application condition attached.

which the arc applies. If the application condition is false, it is as if the arc were not present.

We define feature nets and give their semantics. We adapt the definition of feature nets described in previous work [18], where application conditions apply to transitions instead of arcs. In that paper two semantic definitions of feature nets were presented. The first semantics *directly* models the FN for a given feature selection. The second semantics, which we use and adapt here, is given by *projecting* the FN for a given feature selection onto a Petri net by removing arcs with unsatisfied application conditions. These two notions have been shown to coincide [18]. We start with some necessary preliminaries, first by defining multisets and basic operations over multisets, then by defining feature nets and their behaviour. Our terminology is standard for Petri nets [8].

Definition 1 (Multiset). *A* multiset *over a set S is a mapping $M : S \to \mathbb{N}$.*

We view a set S as a multiset in the natural way, that is, $S(x) = 1$ if $x \in S$, and $S(x) = 0$ otherwise. We also lift arithmetic operators to multisets as follows. For any function $\odot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and multisets M_1, M_2, define $M_1 \odot M_2$ as $(M_1 \odot M_2)(x) = M_1(x) \odot M_2(x)$.

Definition 2 (Application condition [21]). *An* application condition *φ is a propositional formula over a set of features F, defined by the following grammar:*

$$\varphi \quad ::= \quad a \quad | \quad \varphi \wedge \varphi \quad | \quad \neg\varphi,$$

where $a \in F$. Write Φ_F to denote the set of all application conditions over F.

Definition 3 (Satisfaction of application conditions). *Given an application condition $\varphi \in \Phi_F$ and a set of features $FS \subseteq F$, called a* feature selection*, we say that FS satisfies φ, written as $FS \models \varphi$, defined as follows:*

$$\begin{aligned} FS &\models a && \text{iff } a \in FS \\ FS &\models \varphi_1 \wedge \varphi_2 && \text{iff } FS \models \varphi_1 \text{ and } FS \models \varphi_2 \\ FS &\models \neg\varphi && \text{iff } FS \not\models \varphi. \end{aligned}$$

Definition 4 (Feature Net). *A feature net is a tuple $N = (S, T, R, M_0, F, f)$, where S and T are two disjoint finite sets, R is a relation on $S \cup T$ (the* flow relation*) such that $R \cap (S \times S) = R \cap (T \times T) = \emptyset$, and M_0 is a multiset over S, called the* initial marking*. The elements of S are called* places *and the elements of T are called* transitions*. Places and transitions are called* nodes*. The elements of R are called* arcs*. Finally, F is set of features and $f : R \to \Phi_F$ is a function associating each arc with an application condition from Φ_F.*

Without f and F, a feature net is just a Petri net. Sometimes we omit the initial marking M_0. The function f determines a node's pre- and post-set, defined below.

Definition 5 (Marking of a feature net). *A marking M of a feature net (S, T, R, F, f) is a multiset over S. A place $s \in S$ is* marked *iff $M(s) > 0$.*

Definition 6 (Pre-sets and post-sets). *Given a node x of a feature net and a feature selection FS, the set ${}^{(FS)}x = \{y \mid (y,x) \in R, FS \models f(y,x)\}$ is the* pre-set *of x and the set $x^{(FS)} = \{y \mid (x,y) \in R, FS \models f(x,y)\}$ is the* post-set *of x.*

Definition 7 (Enabling). *Given a feature selection FS, a marking M* enables *a transition $t \in T$ if it marks every place in ${}^{(FS)}t$, that is, if $M \geq {}^{(FS)}t$.*

We now define the behaviour of feature nets for a given feature selection.

Definition 8 (Transition occurrence). *Let $N = (S, T, R, M_0, F, f)$ be a feature net and $FS \subseteq F$ a feature selection. A transition $t \in T$* occurs, *leading from a state with marking M_i to a state with marking M_{i+1}, denoted $M_i \xrightarrow{t,FS} M_{i+1}$, iff the following two conditions are met:*

$$M_i \geq {}^{(FS)}t \qquad \text{(enabling)}$$

$$M_{i+1} = (M_i - {}^{(FS)}t) + t^{(FS)} \qquad \text{(computing)}$$

The transition rule for FN is used to define traces that describe the FN's behaviour. We now define the semantics of a feature net by projecting it onto a Petri net for a given feature selection.

Definition 9 (Projection). *Given a feature net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the* projection *of N onto FS, denoted $N \downarrow FS$, is a Petri net (S, T, R', M_0), with $R' = \{(x,y) \mid (x,y) \in R, FS \models f(x,y)\}$.*

One *projects* N onto a feature selection FS by evaluating all application conditions $f(x,y)$ with respect to FS for all arcs $(x,y) \in R$. If FS does not satisfy $f(x,y)$, then (x,y) is removed from the Petri net.

The behaviour of a feature net is the union of the behaviour of the Petri nets obtained by projecting all possible feature selections. The behaviour of a Petri net $N = (S, T, R, M_0)$ is given by the set of all of its traces [12], written $\mathrm{Beh}(N) = \{M_0 \xrightarrow{t_1} \cdots \xrightarrow{t_s} M_n \mid M_i \subseteq S, i \in 1..n, M_{i-1} \xrightarrow{t_i} M_i\}$, and does not include application conditions nor feature selections.

Definition 10 (FN Behaviour). *Given an FN $N = (S, T, R, M_0, F, f)$, we define* $\mathrm{Beh}(N)$ *as follows:*

$$\mathrm{Beh}(N) = \bigcup_{FS \subseteq F} \mathrm{Beh}(N \downarrow FS).$$

A feature net combines the behaviour of a set of Petri nets in a single model. Feature nets do not exceed the expressive power of Petri nets. This is indicated by the fact that a feature net can be encoded as a set of Petri nets. Such an encoding involves two steps: first encoding a FN as a transition-labelled Feature Petri Net [18], and secondly describing the behaviour of the Feature Petri Net using a set of regular Petri nets. The first encoding replaces each transition attached to n arcs in R by 2^n transitions, one for each possible combination of

the possible arcs. The second encoding step into Petri nets can be achieved by encoding the satisfaction condition of FN transition occurrences by considering for each feature F two places, F ON and F OFF, marked in mutual exclusion depending on whether the feature is selected or not. The details of this encoding are in a previous paper [18].

Given that feature nets are as expressive as Petri nets, analysis techniques for Petri nets still apply to feature nets. At the same time, feature nets offer a concise way to describe the systems in an SPL.

4 Modular Modelling

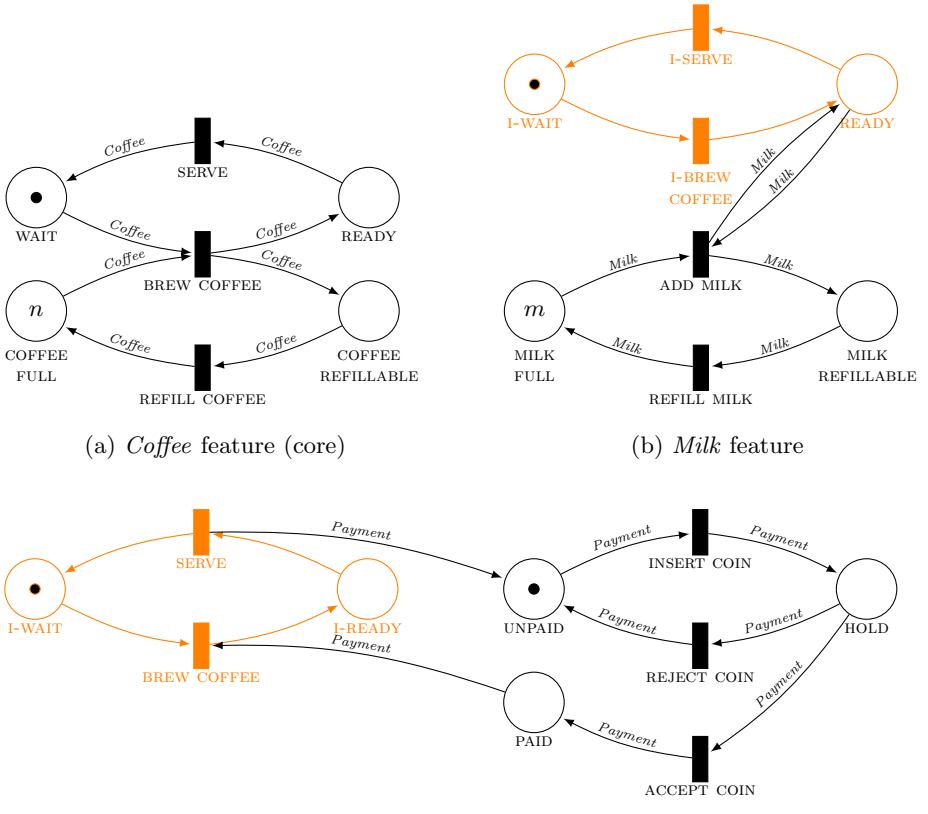
For a modelling formalism to be useful in practice, it needs to facilitate modular development techniques. This is especially important for modelling software product lines: a single SPL model combines the behaviour of a set of different systems, which are often too complex to develop simultaneously.

Modular approaches include top-down techniques, where initially an abstract model is sketched and more details are added incrementally, and bottom-up approaches, where subsystems are modelled separately and later plugged together to a global model. Petri nets support both approaches [12]. In the following we propose a bottom-up composition technique for feature nets. It is based on the idea of modelling features of the system individually and then combining them to obtain a model of the entire SPL. Our approach starts by building a model of the *core* system that is the behaviour which is common to all products of the SPL. Optional features are modelled as separate nets, which also specify how they interact with the core through an *interface*. Core and additional features are then composed stepwise, by incrementally applying each feature to the core. We show how this technique can be applied to modularly specify a coffee machine product line from the three features *Coffee*, *Payment* and *Milk*.

4.1 Feature Net Composition

We devise a modular modelling approach in which features are first expressed as separate FNs. A feature's interaction with the rest of the system (the *core*) is modelled using an *interface*. Features are modelled separately in such a way that they can be attached to the core, in order to incrementally build a larger model. The interface simulates the behaviour of the core that the features are designed to be plugged into. A feature modelled using this technique can be seen as a partially specified model of the entire SPL, where the feature's behaviour is fully specified, whereas everything else is underspecified. Composition then entails connecting the interface to the core to obtain a specification of the combined system.

The three features of our example coffee machine are modelled as separate FNs (Fig. 3). Apart from when a feature's behaviour is self-contained (such as the *Coffee* net in Fig. 3a) it will typically interact with other features that are part of the larger system. To faithfully model such interactions we include an interface. The aim of the interface is to abstract part of the larger system's



(a) *Coffee* feature (core)

(b) *Milk* feature

(c) *Payment* feature

Fig. 3. Individual feature nets modelling the features *Coffee*, *Milk* and *Payment* of a product line of coffee machines. Interfaces are highlighted in orange.

behaviour. The interface will also be used to show that the net exposes the same behaviour as it does when it is part of the combined system. For example, the model of *Milk* in Fig. 3b reflects the fact that adding milk depends on a state of the system in which a cup of fresh coffee is available. The larger system is represented abstractly by the highlighted interface, which models the availability of coffee in the place READY; a token in this place would denote a state in which a freshly brewed cup of coffee is available. Similarly, Fig. 3c models the fact that after a payment has been accepted, the overall system is able to BREW COFFEE, and after serving the coffee, the system goes back to an UNPAID state.

Constructing a model of the whole SPL is done by stepwise applying the delta nets of each feature to a core model. The intuition behind delta net application is that each interface is replaced with a more complex feature net. In our example, the first step could be to refine *Payment*'s interface by replacing it with the feature net for *Coffee*. In a second step, the feature *Milk* is refined by replacing its interface with the net obtained in the previous step.

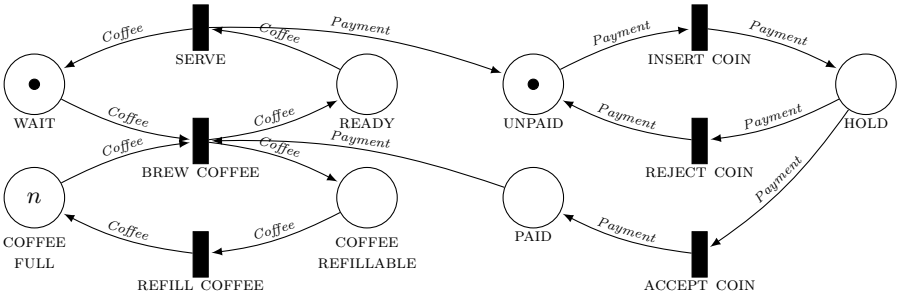


Fig. 4. A software product line over feature set $\{Coffee, Payment\}$ obtained by applying the delta net *Payment* (Fig. 3c) to the core net modelling *Coffee* (Fig. 3a)

We now formally introduce the application of a delta net to a core net.

Definition 11 (Delta Feature Net). *A delta feature net N is a FN with a designated interface, denoted $N = (S, T, R, F, f, S_I, T_I)$, where $S_I \subseteq S$, $T_I \subseteq T$.*

Delta feature nets specify the behaviour of features designed to be added to a larger system. A set of delta FN is combined with a stand-alone FN, the *core*, by sequentially *applying* each delta net to the core. Delta nets include an interface, which models interactions with the core. Such interactions are modelled by transitions or places common to both core and delta net.

Definition 12 (Delta Net Application). *Let $N = (S, T, R, F, f)$ be a feature net and $D = (S_d, T_d, R_d, F_d, f_d, S_I, T_I)$ a delta feature net with $S \cap S_d \neq \emptyset$. The* application *of D to N results in a net $N' = (S', T', R', F', f')$, written as $N \oplus D$, where*

$$S' = (S_d \setminus S_I) \cup S \qquad F' = F \cup F_d$$
$$T' = (T_d \setminus T_I) \cup T \qquad f' = (f \cup f_d) \restriction R'$$
$$R' = \{(s,t) \in (R \cup R_d) \mid s \in S', t \in T'\} \cup \{(t,s) \in (R \cup R_d) \mid t \in T', s \in S'\}.$$

When applying a delta net to a core, the interface is dropped and the two nets are "fused" along their common nodes. The arcs that previously connected the delta net interface now connect the core. The applicability of a delta net is limited to certain cores. Let S_B and T_B represent the border of the interface, that is, $S_B = \{s \in S_I \mid \exists t \in T_d \setminus T_I : (s,t) \in R'\}$ and $T_B = \{t \in T_I \mid \exists s \in S_d \setminus S_I : (t,s) \in R'\}$. A delta net is applicable to a core net if the border of the interface is preserved, that is, if $S \cap S_d = S_B$ and $T \cap T_d = T_B$.

We show how delta net application is used to build a model of the example coffee machines SPL. Starting with the separate sub-models in Fig. 3, delta nets are applied stepwise to a growing core. First, a model with the two features *Coffee* and *Payment* is composed by applying the delta net from Fig. 3c to the core shown in Fig. 3a. These nets have the two transitions SERVE and BREW COFFEE in common. The result after applying the delta feature net is the new core feature net shown in Fig. 4. In a second step, we add the *Milk* behaviour by applying the feature net in Fig. 3b to the core obtained in the previous step.

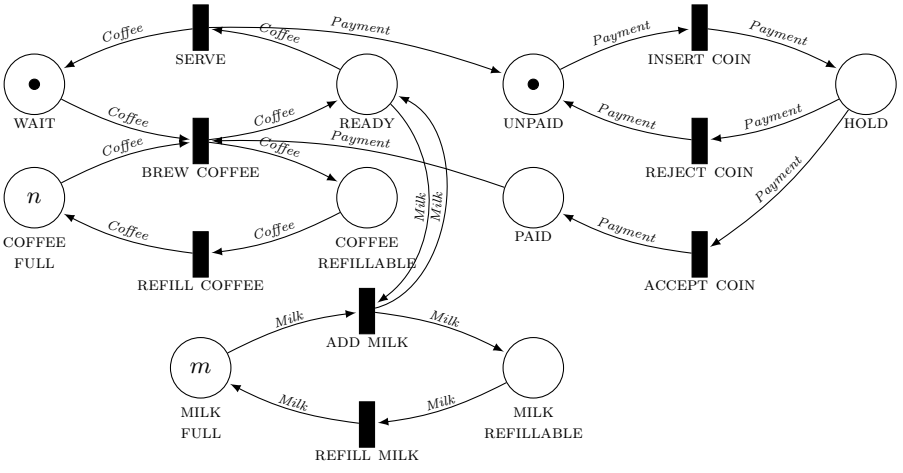


Fig. 5. FN model of an SPL over the feature set $\{Coffee, Payment, Milk\}$ obtained by sequential application of the delta nets for the features *Payment* (Fig. 3c) and *Milk* (Fig. 3b)

These two nets have the place READY in common. The result after delta net application is the model shown in Fig. 5. Note that the order in which we apply the two delta nets does not matter in this case, because neither feature (*Milk* or *Payment*) depends on the other. In general, features can depend on other features. This would be reflected by the design of their interfaces, effectively restricting the applicability and ensuring that the delta nets can only be applied in a valid order. As a consequence, delta net application is not commutative.

5 Correctness

When is the application of a delta net D to a core net N *correct*? We consider this application correct if the traces of N and D are in some way the same as the traces of $N \oplus D$, introduced in Definition 12, after projecting onto the transitions of N and D. However, there are various ways to compare these traces. We can focus only on the features used by the original nets or on the features used by the combined net. Also checking correctness of the core net can be different from checking correctness of the delta net. Finally, it might be enough to consider only trace inclusion between the original nets and the combined net. The three dimensions are summarised as:

- **Original** vs. **combined** features. When comparing the behaviour of one of the original nets with the combined net, we can either consider the combined features in the final net or just the features in one of the original nets.
- **Core** vs. **delta**. We can evaluate the correctness of the core or delta net behaviour, always in comparison to the combined net's behaviour.
- **Liveness**, **safety**, or **both**. Liveness states that a net cannot inhibit behaviour in the other net, while safety states that a net cannot introduce new

behaviour to the other net. For example, we say a delta application is safe with respect to the core net N if the traces of the combined net are included in the traces of N, when considering the common transitions.

By choosing different parameters along these dimensions we obtain different notions of correctness. We formulate a parametrised notion of correctness for the application of delta net D to a core net N as follows:

$$\forall FS \subseteq \Theta_1 : \text{Beh}(\Theta_2 \downarrow FS) \;\; \Theta_3 \;\; \text{Beh}((N \oplus D) \downarrow FS) \quad \text{(parametrised correctness)}$$

where Θ_1 can be either the full set of features or the features of the net Θ_2, Θ_2 can be either the core or the delta net, and Θ_3 is an inclusion or equivalence relation between the two sets of traces, with respect to a set of relevant transitions. When Θ_3 is a superset relation, it represents *safety*, since no new traces can be introduced by combining the two nets. On the other hand, a subset relation represents *liveness*, since all traces in the original net are still valid traces in the combined net. When we have both safety and liveness assurances, we say that the behaviour is *preserved*, and instantiate Θ_3 to be the equality of the traces with respect to the common transitions.

Not all combinations of these dimensions are desirable in all cases. For example, sometimes we might want to inhibit or extend the behaviour of a core net with respect to the combined set of features, breaking the liveness or safety criteria. However, it seems desirable to preserve this behaviour with respect to the features of the core net. In fact, it is open to debate which combination of these dimensions are ideal. In this paper, we provide sufficient conditions to guarantee:

1. *Preservation* of the behaviour of N with respect to the *original* features.
2. *Preservation* of the behaviour of D with respect to the *combined* features.
3. *Safety* of the behaviour of N with respect to the *combined* features.

5.1 Mathematical Preliminaries

We defined liveness and safety as inclusion of traces with respect to a relevant set of traces. We formalise this concept below.

Definition 13 (Behaviour inclusion $\subseteq_{Ts}$). *Let $N_i = (S_i, T_i, R_i)$ be a pair of Petri nets, for $i \in 1..2$, and Ts be a set of transitions. We say that the behaviour of N_1 is included by the behaviour of N_2 with respect to Ts, written* $\text{Beh}(N_1) \subseteq_{Ts} \text{Beh}(N_2)$, *if* $\text{Beh}(N_1) \restriction Ts \subseteq \text{Beh}(N_2) \restriction Ts$, *where* $\text{Beh}(N) \restriction Ts = \{tr \restriction Ts \mid tr \in \text{Beh}(N)\}$ *and:*

$$M \restriction Ts = \varepsilon \qquad (M \xrightarrow{t} tr) \restriction Ts = \begin{cases} t \cdot (tr \restriction Ts) & \text{if } t \in Ts \\ tr \restriction Ts & \text{otherwise.} \end{cases}$$

Similarly, we write $\supseteq_{Ts}$ and $=_{Ts}$ to represent superset inclusion and equality for the transitions in Ts.

We now define *weak bisimulation* between two feature nets, which we will use to relate the interface of a delta net with the net to which the delta is applied to, based on the notion of bisimulation described by Schnoebelen and Sidorova [22].

Definition 14 (Weak bisimulation). *Let $N_i = (S_i, T_i, R_i, M_{0i}, F_i, f_i)$ be two feature nets, for $i \in 1..2$, $\mathcal{M}_i$ the set of markings of N_i, and $\mathcal{B} \subseteq (\mathcal{M}_1 \times \mathcal{M}_2) \cup (T_1 \times T_2)$ a relation over markings and transitions. Recall also the notion of occurrence of transitions introduced in Definition 8. In the following we write $t \in \mathcal{B}$ to denote that t is in the domain or codomain of $\mathcal{B}$. $\mathcal{B}$ is a weak bisimulation if, for any feature selection FS:*

1. $M_{01}\ \mathcal{B}\ M_{02}$
2. $\forall (M_1, M_2) \in \mathcal{B}$, *if* $M_1 \xrightarrow{t_1, FS} M_1'$ *and* $t_1 \notin \mathcal{B}$, *then* $M_1'\ \mathcal{B}\ M_2$;
3. $\forall (M_1, M_2) \in \mathcal{B}$, *if* $M_1 \xrightarrow{t_1, FS} M_1'$ *and* $t_1 \in \mathcal{B}$, *then there exists* $t_2 \in T_2$ *and* M_2' *such that* $M_2 \xrightarrow{t_2, FS}_{\mathcal{B}} M_2'$, $M_1'\ \mathcal{B}\ M_2'$, *and* $t_1\ \mathcal{B}\ t_2$;
4. *conditions (2) and (3) also hold for* $\mathcal{B}^{-1}$;

where $M \xrightarrow{t, FS}_{\mathcal{B}} M'$ *denotes that there are n transitions* $t_1 \ldots t_n$ *such that* $M \xrightarrow{t_1, FS} \cdots \xrightarrow{t_n, FS} M_n \xrightarrow{t, FS} M'$ *and* $\forall j \in 1..n : t_j \notin \mathcal{B}$.
If a weak bisimulation exists between N_1 and N_2 we say that they are weakly bisimilar, written $N_1 \approx N_2$.

Let C be the feature net for the *Coffee* feature (Fig. 3a), and P the delta net dealing with *Payment* (Fig. 3c). The interface of P can be seen as a feature net P_I. It holds that $C \approx P_I$. Furthermore, there exists a bisimulation $\mathcal{B}$ that relates the transitions with the same name of the two nets, namely SERVE and BREW COFFEE. More specifically, the relation $\mathcal{B}$ below is a bisimulation, where we write $\mathcal{M}_C$ and $\mathcal{M}_{P_I}$ to denote all the markings of C and P_I, respectively.

$$\{(M, M') \mid M \in \mathcal{M}_C, M' \in \mathcal{M}_{P_i}, M(\text{WAIT}) = 1, M'(\text{WAIT}) = 1\} \cup$$
$$\{(M, M') \mid M \in \mathcal{M}_C, M' \in \mathcal{M}_{P_i}, M(\text{WAIT}) = 0, M'(\text{WAIT}) = 0\} \cup$$
$$\{(\text{SERVE}, \text{SERVE}), (\text{BREW COFFEE}, \text{BREW COFFEE})\}$$

5.2 Preservation of the Core Behaviour for the Original Features

Our first criterion compares the core net with the combined net, considering only the features originally present in the core net. We require the behaviour of the core net to be *preserved* in the combined net, that is, their traces must coincide with respect to the transitions in the core net. We formalise this criterion as follows.

Criterion 1 (preservation/core/original). *Let $N = (S, T, R, F, M_0, f)$ be a core net and D a delta net. We say that $N \oplus D$ preserves the behaviour of N for the features in F iff*

$$\forall FS \subseteq F \ :\ \text{Beh}(N \downarrow FS) =_T \text{Beh}(N \oplus D \downarrow FS).$$

To verify that a delta net application obeys the above correctness criteria, it is sufficient (although not necessary) to verify the following condition. Check that

the arcs between the interface and the non-interface nodes of D require at least one 'new' feature to be present. By new feature we mean a feature that is not in F. This syntactic check ensures that, when considering only the features from the core net, the arcs connecting it to the delta net will never be active.

Theorem 1. *Let $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ be a delta net, $N = (S, T, R, M_0, F, f)$ a feature net, and $R_I \subseteq R_d$ be the set of arcs connecting interface nodes $(S_I \cup T_I)$ to non-interface nodes. The behaviour of N is preserved by $N \oplus D$ for the features in F (Criterion 1) if:*

$$\forall (x, y) \in R_I \ : \ \forall FS \in F_d \cup F \ : \ FS \models f(x, y) \quad \rightarrow \quad FS \cap (F_d \backslash F) \neq \emptyset. \tag{1}$$

A proof for this theorem can be found in the accompanying technical report [19]. In both our examples of delta applications, that is, adding payment to a coffee machine and adding milk to the resulting net, the condition in Equation (1) holds. The intuition is that, for example, when the *Payment* feature is not available, the *Coffee* feature net is detached from the *Payment* feature net in the combined net. Hence its behaviour is not affected by the *Payment* net and is preserved.

5.3 Preservation of the Delta Behaviour for the Combined Features

We now define the second correctness criterion.

Criterion 2 (preservation/delta/combined). *Let $N = (S, T, R, M_0, F, f)$ be a core net and $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ a delta net. We say that $N \oplus D$ preserves the behaviour of D with respect to features from the combined net iff*

$$\forall FS \subseteq F \cup F_d \ : \ \mathrm{Beh}(D \downarrow FS) =_{T_d \backslash T_I} \mathrm{Beh}(N \oplus D \downarrow FS).$$

As with the correctness Criterion 1, we present a sufficient condition that guarantees the preservation of the Criterion 2. However, as opposed to the previous case, this condition is based on a *semantic property* of the the interface and the core net.

Theorem 2. *Let $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ be a delta net, $N_I = (S_I, T_I, R_I, M_{0D}, F_d, f_d)$ be the interface of D, $N = (S, T, R, F, f)$ a (core) feature net, and $R_B \subseteq R_d$ denote the arcs connecting interface to non-interface nodes. The behaviour of D is preserved by $N \oplus D$ (Criterion 2) if $N \approx N_I$ and there is a weak bisimulation $\mathcal{B} \subseteq (\mathcal{M}_1 \times \mathcal{M}_2) \cup (T_1 \times T_2)$ such that:*

$$\{(t, t) \mid t \in T \cap T_I\} \subseteq \mathcal{B}, \tag{2}$$

$$\forall s \in S \cap S_I, (M, M') \in \mathcal{B} : M(s) = M'(s), \tag{3}$$

$$\forall (s, t) \in R_B, s \in S_I, (M, M') \in \mathcal{B} \ : \ (M - \{s \mapsto 1\}) \ \mathcal{B} \ (M' - \{s \mapsto 1\}) \tag{4}$$

$$\forall (t, s) \in R_B, s \in S_I, (M, M') \in \mathcal{B} \ : \ (M + \{s \mapsto 1\}) \ \mathcal{B} \ (M' + \{s \mapsto 1\}) \tag{5}$$

For Equation (4) we assume that, if $M(s) = M'(s) = 0$, then subtracting $\{s \mapsto 1\}$ does not change the markings.

The proof for Theorem 2 can be found in the accompanying technical report [19]. Recall our running examples. As explained in the end of Section 5.1, there is a weak bisimulation between the interface of the delta net for payment P and the core net for coffee C. This simulation obeys Equation (2) because the shared transitions are related by $\mathcal{B}$, Equation (3) because there places of C and P are disjoint, and Equation (4) because, in this case, $\mathrm{dom}(R) \cap S_I = \emptyset$. Hence the composition $CP = C \oplus P$ is correct with respect to Criterion 2. Consider now the application of the delta net for milk M to the previously obtained core CP. A possible weak bisimulation between CP and the interface of M relates equal markings of the places READY in CP and READY in M, as well as of the places WAIT and I-WAIT. Note that, in order to use Theorem 2, we need to include markings for any number of tokens in READY, because of Equations (4) and (5). Hence, Equation (2) trivially holds, and our specific bisimulation relation also captures Equation (3). We conclude that the composition $CP \oplus M$ is also correct with respect to Criterion 2.

5.4 Safety of the Core Behaviour for the Combined Features

Our last correctness criterion compares the core net with the combined net with respect to all features, as opposed to the first criterion that only considered the features of the core net. When including the features in the delta net, we consider it *safe* to inhibit traces that were initially possible, provided that no new traces are introduced. We formalise safety using trace inclusion.

Criterion 3 (safety/core/combined). *Let $N = (S, T, R, M_0, F, f)$ be a core net and $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ a delta net. We say that $N \oplus D$ is safe with respect to D and to the combined features iff*

$$\forall FS \subseteq F \cup F_d \ : \ \mathrm{Beh}(N \downarrow FS) \supseteq_T \mathrm{Beh}(N \oplus D \downarrow FS).$$

We claim that, when applying a delta net connecting only places from the interface to the rest of the delta, the delta net application is safe with respect to D and the combined features.

Theorem 3. *When applying a delta net $D = (S_d, T_d, R_d, M_{0D}, F_d, f_d, S_I, T_I)$ to a core net N, we guarantee that $N \oplus D$ is safe with respect to D and the combined features if:*

$$\forall s \in S_I, t \in T_d \backslash T_I \ : \ (s, t) \notin R_D. \tag{6}$$

The theorem is easily justified by the fact that the core net will only be connected to the rest of the delta net through transitions. When the application of a delta net respects Equation (6), we are increasing the pre- and post-sets of these transitions, thereby further restricting when they can be fired.

We exemplify the application of two delta nets in this paper: the *Payment* and the *Milk* nets (Fig. 3c and 3b). The first net obeys the condition in Theorem 3, hence the correctness Criterion 3 holds. The second delta net has arcs connecting places from the interface to a non-interface transition, invalidating Equation (6). However, in this case the safety criterion is preserved, because a token that exits the core when firing ADD MILK is transported back to its origin in the same step.

6 Related Work

Our research relates to Petri net based formalisms, and to the behavioural specification of software product lines. We highlight the most relevant works in these areas. Petri net composition and decomposition strategies that preserve some properties of the initial net(s) have been studied thoroughly [4,23,22,12]. In *Open Petri Nets* [2], places designated as open represent an interface towards the environment. Open nets are composed by fusing common open places, and the composition operation is shown to preserve behaviour with respect to an inverse decomposition operation. Our Petri net model uses a similar notion of interface, which includes an abstraction of the net that will be matched during application. We use an incremental approach using application of deltas instead of a symmetric composition operation, guided by the intuition that larger systems are build by extending more fundamental systems. The main focus of open Petri nets is the study of properties in a category of nets, while we have a more practical focus on the incremental development of nets. Various formalisms have been adopted for specifying the behaviour of software product lines, with the aim of providing a basis for analysis and verification of such models. A survey of formal methods for software product lines has recently been published [5]. *UML activity diagrams* have been used to model the behaviour of SPL by superimposing several such diagrams in a single model [7]. Attached to the activity diagram's elements are "presence expressions," which are similar to application conditions. Compared to activity diagrams, Petri nets have a stronger formal foundation, with a larger spectrum of analysis and verification techniques, although, several studies have expressed the semantics of UML diagram using Petri nets (e.g. [10]). Gruler et al. extended Milner's CCS with a product line variant operator that allows an alternative choice between two processes [14]. The *PL-CCS* calculus includes information about variability: by defining dependencies between features, one can control the set of valid configurations [13]. Variability is often modelled using transition systems enhanced with product-related information. *Modal transition systems* (MTS) [15] allow optional transitions, lending themselves as a tool for modelling a set of behaviours at once [11]. Generalised extended MTS [9] introduce cardinality-based variability operators and propose to use temporal logic formulas to associate related variation points. Asirelli et al. encode MTS using propositional deontic logic formulas and provide a framework for reasoning about behavioural aspects [1]. *Modal I/O automata* [16] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms for component composition are provided to support a product line theory. These approaches do not relate behaviour to elements of a structural variability model. *Featured transition systems* (FTS) [6] are an extension of labeled transition systems. Similar to feature nets, transitions are explicitly labeled with respect to a feature model, and a feature selection determines the subset of active transitions. In FTS, transitions are mapped to single features. Transition priorities are used to deal with undesired non-determinism when selecting from transitions associated to different features. With application

conditions, priorities are no longer required because we can negate the features in other transitions to obtain the same effect.

7 Conclusion and Future Work

This paper introduces a modular framework for modelling systems with a high degree of variability, addressing an important challenge in software product line engineering. The modelling formalism used is feature nets, a lightweight Petri net extension in which the presence of arcs is conditional on the presence of certain features through *application conditions*. We present an approach to composing behavioural models from separately engineered models of individual features. Three correctness criteria for such compositions are also presented.

Feature nets capture the behaviour of entire product lines in a single, concise model, opening the way for efficient analysis and verification. We will follow this direction in future work, applying model checking techniques to our models and studying the question of verification. The practical applicability of our proposed approach will be examined in a future study, which will also determine how well the approach scales, considering that features are not always independent.

References

1. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: A deontic logical framework for modelling product families. In: Benavides et al. [3], pp. 37–44
2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Mathematical Structures in Computer Science 15(01), 1–35 (2005)
3. Benavides, D., Batory, D.S., Grünbacher, P. (eds.): International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), vol. 37. Universität Duisburg-Essen (2010)
4. Berthelot, G.: Transformations and decompositions of nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986, Part 1. LNCS, vol. 254, pp. 359–376. Springer, Heidelberg (1987)
5. Clarke, D.: Quality Assurance for Diverse Systems, ch. 5, pp. 27–37. Deliverable 1.2 of the EternalS Coordination Action (FP7-247758), supported by the 7th Framework Programme of the EC within the FET scheme (2011), https://www.eternals.eu/sites/default/file/D1_2_TF1_stateOfTheArt.pdf
6. 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: International Conference on Software Engineering, pp. 335–344. IEEE Press, Los Alamitos (2010)
7. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
8. Desel, J., Esparza, J.: Free choice Petri nets. Cambridge University Press, New York (1995)
9. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: International Software Product Line Conference, pp. 193–202. IEEE Press, Los Alamitos (2008)

10. Farooq, U., Lam, C.P., Li, H.: Transformation methodology for UML 2.0 activity diagram into colored Petri nets. In: Advances in Computer Science and Technology, pp. 128–133. ACTA Press (2007)
11. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: International Workshop on the Role of Software Architecture in Analysis and Testing, pp. 39–48. ACM Press, New York (2006)
12. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer, Secaucus (2001)
13. Gruler, A., Leucker, M., Scheidemann, K.: Calculating and modeling common parts of software product lines. In: International Software Product Line Conference, pp. 203–212. IEEE Press, Los Alamitos (2008)
14. Gruler, A., Leucker, M., Scheidemann, K.: Modeling 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)
15. Larsen, K., Thomsen, B.: A modal process logic. In: Third Annual Symposium on Logic in Computer Science, pp. 203–210. IEEE Press, Los Alamitos (1988)
16. Larsen, K., Nyman, U., Wąsowski, 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)
17. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
18. Muschevici, R., Clarke, D., Proença, J.: Feature Petri Nets. In: International Software Product Line Conference, vol. 2, pp. 99–106. Lancaster University (2010)
19. Muschevici, R., Proença, J., Clarke, D.: Modular modelling of software product lines with feature nets. Tech. Rep. CW 609, KU Leuven, Belgium (2011), http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW609.abs.html
20. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer, Heidelberg (2005)
21. Schaefer, I.: Variability modelling for model-driven development of software product lines. In: Benavides, et al. [3], pp. 85–92
22. Schnoebelen, P., Sidorova, N.: Bisimulation and the reduction of Petri nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 409–423. Springer, Heidelberg (2000)
23. Souissi, Y., Memmi, G.: Composition of nets via a communication medium. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 457–470. Springer, Heidelberg (1991)

Synchronizing Asynchronous Conformance Testing

Neda Noroozi[1,2], Ramtin Khosravi[3],
Mohammad Reza Mousavi[1], and Tim A.C. Willemse[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Fanap Corporation (IT Subsidiary of Pasargad Bank), Tehran, Iran
[3] University of Tehran, Tehran, Iran

Abstract. We present several theorems and their proofs which enable using synchronous testing techniques such as input output conformance testing (**ioco**) in order to test implementations only accessible through asynchronous communication channels. These theorems define when the synchronous test-cases are sufficient for checking all aspects of conformance that are observable by asynchronous interaction with the implementation under test.

1 Introduction

Due to the ubiquitous presence of distributed systems (ranging from distributed embedded systems to the Internet), it becomes increasingly important to establish rigorous model-based testing techniques with an asynchronous model of communication in mind. This fact has been noted by the pioneering pieces work in the area of formal conformance testing, e.g., see [7, Chapter 5], [10] and [11], and has been addressed extensively by several researchers in this field ever since [2,4,5,6,12,13].

We stumbled upon this problem in our attempt to apply input-output conformance testing (**ioco**) [8,9] to an industrial embedded system from the banking domain [1]. A schematic view of the implementation under test (IUT) and its environment is given in Figure 1.(a). The IUT is an Electronic Funds Transfer (EFT) switch, which provides a communication mechanism among different components of a card-based financial system. On one side of the IUT, there are components that the end-user deals with, such as Automated Teller Machines (ATMs), Point-of-Sale (POS) devices and e-Payment applications. On the other side, there are Core-Banking systems and the inter-bank network connecting EFT switches of different financial institutions.

To test the EFT switch, an automated on-line test-case generator is connected to it; the tester communicates (using an adapter) via a network with the IUT. This communication is inherently asynchronous and hence subtleties concerning asynchronous testing arise naturally in our context. A simplified specification of the switch in which these subtleties appear is depicted in Figure 1.(b). In this figure, the EFT switch sends a purchase request to the core banking system and either receives a response or after an internal step (e.g., an internal time-out, denoted by τ) sends a reversal request to the POS. In the synchronous setting, after sending a purchase request and receiving a response, observing a reversal request will lead to the fail verdict. This is justified by the fact that receiving a response should force the system to take the left-hand-side

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 334–349, 2011.

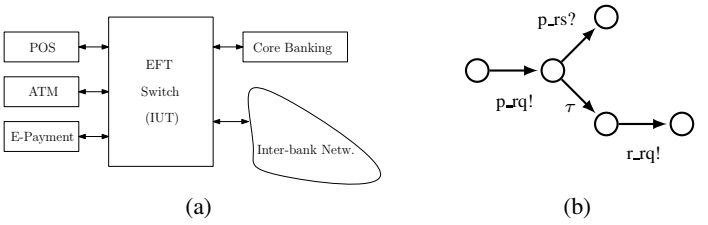


Fig. 1. EFT Switch and a simplified specification

transition at the moment of choice in the depicted specification. However, in the asynchronous setting, a response is put on a channel and is yet to be communicated to the IUT. It is unclear to the remote observer when the response is actually consumed by the IUT. Hence, even when a response is sent to the system the observer should still expect to receive a reversal request.

The problems encountered in our practical case study have been encountered by other researchers. It is well-known that not all systems are amenable to asynchronous testing since they may feature phenomena (e.g., a choice between accepting input and generating output) that cannot be reliably observed in the asynchronous setting (e.g., due to unknown delays). In other words, to make sure that test-cases generated from the specification can test the IUT by asynchronous interactions and reach verdicts that are meaningful for the original IUT, either the class of IUTs, or the class of specifications, or the test-case generation algorithm (or a combination thereof) has to be adapted.

Related work. In [12, Chapter 8] and [13], both the class of IUTs has been restricted (to the so-called *internal choice* specifications) and further the test-case generation algorithm is adapted to generate a restricted set of test-cases. Then, it is argued (with a proof sketch) that in this setting, the verdict obtained through asynchronous interaction with the system coincides with the verdict (using the same set of restricted test-cases) in the synchronous setting. We give a full proof of this result in Section 3 and report a slight adjustment to it, without which a counter-example is shown to violate the property. It remains to be investigated what notion of conformance testing is induced by the class of test-cases proposed in [12,13].

In [6] a method is presented for generating test-cases from the synchronous specification that are sound for the asynchronous implementation. The main idea is to saturate a test-case with observation delays caused by asynchronous interactions. In this paper, we adopt a restriction imposed on the implementation inspired by [6, Theorem 1] and prove that in the setting of **ioco** testing this is sufficient for using synchronous test-case for the asynchronous implementation (dating back to [7]).

In [4,5] the asynchronous test framework is extended to the setting where separate test-processes can observe input and output events and relative distinguishing power of these settings are compared. Although this framework may be natural in practice, we avoid following the framework of [4,5] since our ultimate goal is to compare asynchronous testing with the standard **ioco** framework and the framework of [4,5] is notationally very different. For the same reason, we do not consider the approach of [2], which uses a stamping mechanism attached to the IUT, thus observing the actual order input and output before being distorted by the queues.

To summarize, the present paper re-visits the much studied issue of asynchronous testing and formulates and proves some theorems that show when it is (im)possible to synchronize asynchronous testing, i.e., interaction with an IUT through asynchronous channels and still obtain verdicts that coincide with that of testing the IUT using the synchronous interaction mechanisms.

Structure of the paper. After presenting some preliminaries in Section 2, we give a full proof of the main result of [12, Chapter 8] and [13] (with a slight modification) in Section 3. Then, in Section 4, we re-formulate the same results in the pure **ioco** setting. Finally, in Section 5, we show that the restrictions imposed on the implementation in Section 4 are not only sufficient to obtain the results but also necessary and hence characterize the implementations for which asynchronous testing can be reduced to synchronous testing. The paper is concluded in Section 6.

2 Preliminaries

In this section, we review some common formal definitions from the literature of labeled transition systems and **ioco** testing [9].

Specifications, actions and traces. In our model-based testing approach, systems are typically formalized using variations of a labeled transition system (LTS). Let τ be a constant representing an unobservable action.

Definition 1 (LTS). *A labeled transition system (LTS) is a 4-tuple $\langle S, L, \rightarrow, s_0 \rangle$, where S is a set of states, L is a finite alphabet that does not contain τ, $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, and $s_0 \in S$ is the initial state.*

Fix an arbitrary LTS $\langle S, L, \rightarrow, s_0 \rangle$. We shall often refer to the LTS by referring to its initial state s_0. Let $s, s' \in S$ and $x \in L \cup \{\tau\}$. We write $s \xrightarrow{x} s'$ rather than $(s, x, s') \in \rightarrow$; moreover, we write $s \xrightarrow{x}$ when $s \xrightarrow{x} s'$ for some s', and $s \not\xrightarrow{x}$ when not $s \xrightarrow{x}$. The transition relation is generalized to (weak) traces by the following deduction rules:

$$\frac{}{s \stackrel{\epsilon}{\Longrightarrow} s} \qquad \frac{s \stackrel{\sigma}{\Longrightarrow} s'' \quad s'' \xrightarrow{x} s' \quad x \neq \tau}{s \stackrel{\sigma x}{\Longrightarrow} s'} \qquad \frac{s \stackrel{\sigma}{\Longrightarrow} s'' \quad s'' \xrightarrow{\tau} s'}{s \stackrel{\sigma}{\Longrightarrow} s'}$$

In line with our notation for transitions, we write $s \stackrel{\sigma}{\Longrightarrow}$ if there is a s' such that $s \stackrel{\sigma}{\Longrightarrow} s'$, and $s \stackrel{\sigma}{\not\Longrightarrow}$ when no s' exists such that $s \stackrel{\sigma}{\Longrightarrow} s'$.

Definition 2 (Traces and Enabled Actions). *Let $s \in S$ and $S' \subseteq S$. We define:*

1. $\mathsf{traces}(s) =_{def} \{\sigma \in L^* \mid s \stackrel{\sigma}{\Longrightarrow}\}$, *and we define* $\mathsf{traces}(S') =_{def} \bigcup_{s \in S'} \mathsf{traces}(s)$
2. $\mathbf{init}(s) =_{def} \{a \in L \cup \{\tau\} \mid s \xrightarrow{a}\}$, *and we define* $\mathbf{init}(S') =_{def} \bigcup_{s \in S'} \mathbf{init}(s)$,
3. $\mathbf{Sinit}(s) =_{def} \{a \in L \mid s \stackrel{a}{\Longrightarrow}\}$, *and we define* $\mathbf{Sinit}(S') =_{def} \bigcup_{s \in S'} \mathbf{Sinit}(s)$.

A state in an LTS is said to *diverge* if it is the source of an infinite sequence of τ-labeled transitions. An LTS is *divergent* if one of its reachable states diverges.

Inputs, Outputs and Quiescence. In LTSs labels are treated uniformly. When engaging in an interaction with an actual implementation, the initiative to communicate is often not fully symmetric: the implementation is stimulated and observed. We therefore refine the LTS model to incorporate this distinction.

Definition 3 (IOLTS). *An input-output labeled transition system (IOLTS) is an LTS* $\langle S, L, \rightarrow, s_0 \rangle$*, where the alphabet* L *is partitioned into a set* L_I *of inputs and a set* L_U *of outputs.*

Throughout this paper, whenever we are dealing with an IOLTS (or one of its refinements), we tacitly assume that the given alphabet L for the IOLTS is partitioned in sets L_I and L_U. In our examples we distinguish inputs from outputs by annotating them with question- (?) and exclamation-mark (!), respectively. Note that these annotations are *not* part of action names.

Quiescence, defined below, is an essential ingredient in the more advanced conformance testing theories. In its traditional phrasing, it characterizes system states that do not produce outputs and which are stable, i.e., those that cannot evolve to another state by performing a silent action.

Definition 4 (Quiescence). *State* $s \in S$ *is called* quiescent*, denoted by* $\delta(s)$*, iff* $\mathbf{init}(s) \subseteq L_I$*. We say* s *is* weakly quiescent*, denoted by* $\delta_q(s)$*, iff* $\mathbf{Sinit}(s) \subseteq L_I$*.*

The notion of weak quiescence is appropriate in the asynchronous setting, where the lags in the communication media interfere with the observation of quiescence: an observer cannot tell whether a system is engaged in some internal transitions or has come to a standstill. By the same token, in an asynchronous setting it becomes impossible to distinguish divergence from quiescence; we re-visit this issue in our proofs of synchronizing asynchronous conformance testing.

Testing hypotheses. Several formal testing theories build on the assumption that the implementations can be modeled by a particular IOLTS; this assumption is part of the so-called *testing hypothesis* underlying the testing theory. Not all theories rely on the same assumptions. We introduce two models, viz., the *input output transition systems*, used in Tretmans' testing theory [9] and the *internal choice input output transition systems*, introduced by Weiglhofer and Wotawa [12,13].

Tretmans' input-output transition systems are basically plain IOLTSs with the additional assumption that inputs can always be accepted.

Definition 5 (IOTS). *A state* $s \in S$ *in an IOLTS* $M = \langle S, L, \rightarrow, s_0 \rangle$ *is* input-enabled *iff* $L_I \subseteq \mathbf{Sinit}(s)$*. The IOLTS* M *is an* input output transition system *(IOTS) iff every state* $s \in S$ *is input-enabled.*

From hereon, we denote the class of input output transition systems ranging over L_I and L_U by $\mathsf{IOTS}(L_I, L_U)$. Weiglhofer and Wotawa's internal choice input output transition systems relax Tretmans' input-enabledness requirement; at the same time, however, they impose an additional restriction on the presence of inputs.

Definition 6 (Internal choice IOTS). *An IOLTS* $\langle S, L, \rightarrow, s_0 \rangle$ *is an* internal choice input output transition system *(IOTS*$^{\sqcap}$*) if:*

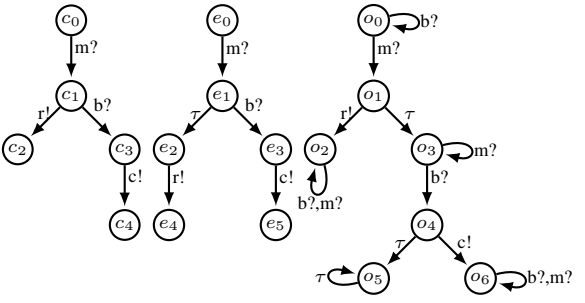


Fig. 2. IOLTS with different moments of choice (*m:money, r:refund, b:button, c:coffee)*

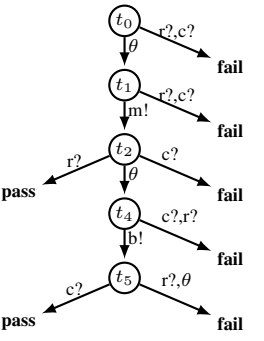
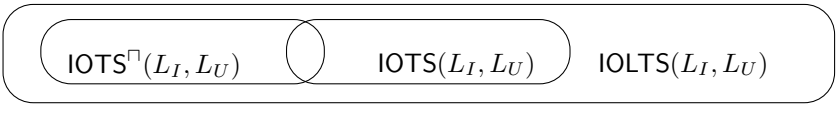


Fig. 3. A test case

1. *quiescent states are input-enabled, i.e., for all $s \in S$, if $\delta(s)$, then $L_I \subseteq \mathbf{Sinit}(s)$*
2. *only quiescent states may accept inputs, i.e., for all $s \in S$, if $\mathbf{init}(s) \cap L_I \neq \emptyset$ then $\delta(s)$.*

We denote the class of IOTS$^{\sqcap}$ models ranging over L_I and L_U by $\mathsf{IOTS}^{\sqcap}(L_I, L_U)$. The following Venn-diagram visualizes the relation between the two different testing hypotheses.



We note that the intersection between $\mathsf{IOTS}^{\sqcap}(L_I, L_U)$ and $\mathsf{IOTS}(L_I, L_U)$ is in a sense only fulfilled by the most superficial models, viz., those IOLTSs that never provide proper outputs. If requirement 2 is dropped from Definition 6, then clearly $\mathsf{IOTS}^{\sqcap}(L_I, L_U)$ subsumes $\mathsf{IOTS}(L_I, L_U)$.

Example 1. The two labeled transition systems c_0 and e_0 in Figure 2 model a coffee machine which after receiving money, either refunds or accepts it, lets the coffee button be pressed and delivers coffee consequently. IOLTS o_0 in Figure 2 models a disordered coffee machine which after pressing coffee button may or may not deliver coffee. In IOLTS c_0, after doing the first transition, inserting money, there is a choice between input and output. Although IOLTS e_0 does not feature an immediate race between input and output actions, the possibility of output $r!$ can be ruled out by providing input $b?$. In the IOLTS o_0, however, there is a moment of time after which no output can be observed, i.e., after taking the unobservable transition the system reaches the quiescent state and the input $b?$ is accepted by the system.

IOLTSs c_0 and e_0 are not internal choice IOTSs while o_0 is. None the aforementioned IOLTSs are IOTSs; they can be made IOTSs by adding self-loops for all absent input transitions at each and every state.

Testing. We next define the notion of a test case. We assume that it can, in the most general case, be described by a tree-shaped IOLTS. Such a test case prescribes when

an input should be fed to the implementation-under-test and when its possible outputs should be observed. In a test case, the *observation* of quiescence is modeled using a special θ symbol.

Definition 7 (Test case). *A test case is an IOLTS $\langle S, L, \rightarrow, s_0 \rangle$, where S is a finite set of states reachable from $s_0 \in S$, the terminal states* **pass** *and* **fail** *are part of S, and we have $\theta \in L_I$. In addition, the transition relation $\rightarrow$ is acyclic and deterministic such that:*

1. **pass** *and* **fail** *states appear only as targets of transitions labeled by an element of L_I, and*
2. *for all $s \in S \setminus \{\mathbf{pass}, \mathbf{fail}\}$, we require that $\mathbf{init}(s) = (L_I \setminus \{\theta\}) \cup \{x\}$ for some $x \in L_U \cup \{\theta\}$.*

We denote the class of test cases ranging over inputs L_I and outputs L_U by $\mathsf{TTS}(L_U, L_I)$.

Notice that the observation θ is an *input* to a test case; this is in line with the view that outputs produced by an implementation are inputs to a test case. Moreover, we note that a test case has no transitions labeled with the silent action τ.

We formalize the way a test case communicates with an actual implementation, modeled by an IOLTS.

Definition 8 (Synchronous execution). *Let $M = \langle S, L, \rightarrow, s_0 \rangle$ be an IOLTS, and let $\langle T, L', \rightarrow, t_0 \rangle$ be a test case, such that $L_I = L'_U$ and $L_U = L'_I \setminus \{\theta\}$. Let $s, s' \in S$ and $t, t' \in T$. Then the synchronous execution of the test case and M is defined through the following inference rules:*

$$\frac{s \xrightarrow{\tau} s'}{t \rceil\!\rceil s \xrightarrow{\tau} t \rceil\!\rceil s'} \; (R1) \qquad \frac{t \xrightarrow{x} t' \qquad s \xrightarrow{x} s'}{t \rceil\!\rceil s \xrightarrow{x} t' \rceil\!\rceil s'} \; (R2) \qquad \frac{t \xrightarrow{\theta} t' \qquad \delta(s)}{t \rceil\!\rceil s \xrightarrow{\theta} t' \rceil\!\rceil s} \; (R3)$$

Finally, we state what it means for an implementation to *pass* a test case.

Definition 9 (Verdict). *Let implementation M be given by IOLTS $\langle S, L, \rightarrow, s_0 \rangle$, and let $\langle T, L \cup \{\theta\}, \rightarrow, t_0 \rangle$ be a test case. We say that state $s \in S$* passes *the test case, denoted s* **passes** *t_0 iff there is no $\sigma \in (L \cup \{\theta\})^*$ and no state $s' \in S$, such that $t_0 \rceil\!\rceil s \stackrel{\sigma}{\Longrightarrow} \mathbf{fail} \rceil\!\rceil s'$.*

3 Adapting IOCO to Asynchronous Setting

In order to perform conformance testing in the asynchronous setting in [12] and [13] both the class of implementations and test cases are restricted. Then, it is argued (with a proof sketch) that in this setting, the verdict obtained through asynchronous interaction with the system coincides with the verdict (using the same set of restricted test-cases) in the synchronous setting. In this section, we re-visit the approach of [12] and [13], give full proof of their main result and point out a slight imprecision in it.

3.1 Test Cases for Internal Choice Implementations

Asynchronous communication delays obscure the observation of the tester; for example, the tester cannot precisely establish when the input sent to the system is actually consumed by it.

Internal choice test-cases, formally defined below, only allow for providing an input if quiescence has been observed beforehand.

Definition 10 (Internal choice test case). *A test case* $\langle S, L, \rightarrow, s_0 \rangle$ *is an* internal choice test case *(abbreviated to TTS$^{\sqcap}$) if for all* $s \in S$*, all* $x \in L_U$ *and all* $\sigma \in L^*$*, if* $\sigma x \in \mathsf{traces}(s)$ *then* $\sigma = \sigma' \cdot \theta$*.*

We denote the class of internal choice test cases ranging over inputs L_I and outputs L_U by $\mathsf{TTS}^{\sqcap}(L_U, L_I)$.

Example 2. Figure 3 shows an internal choice test case for o_0 in Figure 2. In this test case, inputs for the implementation are enabled only in states reached by a θ-transition.

The property given below illustrates that, indeed, the interaction between an internal choice test case and an IOLTS proceed in an orchestrated fashion: the IOLTS is only provided stimuli whenever it has reached a stable situation.

Property 1. Let $\langle S, L, \rightarrow, s_0 \rangle$ be an arbitrary IOLTS and $\langle T, L', \rightarrow, t_0 \rangle$ be an internal choice test case. Let $x \in L'_U \setminus \{\theta\}$ $(= L_I)$, $\sigma \in L'^*$, $s, s' \in S$ and $t, t' \in T$. We have the following property:

$$t \rceil s \stackrel{\sigma \cdot x}{\Longrightarrow} t' \rceil s' \text{ implies } \exists \sigma' \in L'^* : \sigma = \sigma' \cdot \theta$$

3.2 Asynchronous Communication

Asynchronous communication, as described in [7, Chapter 5], can be simulated by modelling the communications with the implementation through two dedicated FIFO channels. One is used for sending the inputs to the implementation, whereas the other is used to queue the outputs produced by the implementation. We assume that the channels are unbounded. By adding channels to an implementation, its visible behavior changes. This is formalized below.

Definition 11 (Queue operator). *Let* $\langle S, L, \rightarrow, s_0 \rangle$ *be an arbitrary IOLTS,* $\sigma_i \in L_I^*$*,* $\sigma_u \in L_U^*$ *and* $s, s' \in S$*. The unary queue operator* $_{[\sigma_u \ll - \ll \sigma_i]}$ *is then defined by the following axioms and inference rules:*

$$[\sigma_u \ll s \ll \sigma_i] \xrightarrow{a} [\sigma_u \ll s \ll \sigma_i \cdot a], \qquad a \in L_I \qquad \textit{(A1)}$$

$$[x \cdot \sigma_u \ll s \ll \sigma_i] \xrightarrow{x} [\sigma_u \ll s \ll \sigma_i], \qquad x \in L_U \qquad \textit{(A2)}$$

$$\frac{s \xrightarrow{\tau} s'}{[\sigma_u \ll s \ll \sigma_i] \xrightarrow{\tau} [\sigma_u \ll s' \ll \sigma_i]} \qquad \textit{(I1)}$$

$$\frac{s \xrightarrow{a} s' \quad a \in L_I}{[\sigma_u \ll s \ll a \cdot \sigma_i] \xrightarrow{\tau} [\sigma_u \ll s' \ll \sigma_i]} \qquad \textit{(I2)}$$

$$\frac{s \xrightarrow{x} s' \quad x \in L_U}{[\sigma_u \ll s \ll \sigma_i] \xrightarrow{\tau} [\sigma_u \cdot x \ll s' \ll \sigma_i]} \quad (I3)$$

We abbreviate $[\langle\rangle \ll s \ll \langle\rangle]$ *to* $Q(s)$*. Given an initial state* s_0 *of an IOLTS* M*, the initial state of* M *in queue context is given by* $Q(s_0)$*.*

Observe that for an arbitrary IOLTS M with initial state s_0, $Q(s_0)$ is again an IOLTS. We have the following property, relating the traces of an IOLTS to the traces it has in the queued context.

Property 2. Let $\langle S, L, \rightarrow, s_0\rangle$ be an arbitrary IOLTS. Then for all $s, s' \in S$, we have $s \stackrel{\sigma}{\Longrightarrow} s'$ implies $Q(s) \stackrel{\sigma}{\Longrightarrow} Q(s')$.

The possibility of internal transitions is not observable to the remote asynchronous observer and hence, in [12,13], weak quiescence is adopted to denote quiescence in the queue context.

Definition 12 (Synchronous execution in the queue context). *Let* $M = \langle S, L, \rightarrow, s_0\rangle$ *be an IOLTS, and let* $\langle T, L', \rightarrow, t_0\rangle$ *be a test case, such that* $L_I = L'_U$ *and* $L_U = L'_I \setminus \{\theta\}$*. Let* $s, s' \in S$ *and* $t, t' \in T$*. Then the synchronous execution of the test case and* $Q(M)$ *is defined through the following inference rules:*

$$\frac{[\sigma_u \ll s \ll \sigma_i] \xrightarrow{\tau} [\sigma'_u \ll s' \ll \sigma'_i]}{t \rceil\!\| [\sigma_u \ll s \ll \sigma_i] \xrightarrow{\tau} t \rceil\!\| [\sigma'_u \ll s' \ll \sigma'_i]} \quad (R1')$$

$$\frac{t \xrightarrow{x} t' \qquad [\sigma_u \ll s \ll \sigma_i] \xrightarrow{x} [\sigma'_u \ll s' \ll \sigma'_i]}{t \rceil\!\| [\sigma_u \ll s \ll \sigma_i] \xrightarrow{x} t' \rceil\!\| [\sigma'_u \ll s' \ll \sigma'_i]} \quad (R2')$$

$$\frac{t \xrightarrow{\theta} t' \qquad \delta_q([\sigma_u \ll s \ll \sigma_i])}{t \rceil\!\| [\sigma_u \ll s \ll \sigma_i] \xrightarrow{\theta} t' \rceil\!\| [\sigma_u \ll s \ll \sigma_i]} \quad (R3')$$

The property below characterizes the relation between the test runs obtained by executing an internal choice test case in the synchronous setting and by executing a test case in the queued setting.

Property 3. Let $\langle S, L, \rightarrow, s_0\rangle$ be an IOLTS and let $\langle T, L', \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. Consider arbitrary states $s, s' \in S$ and $t, t' \in T$ and an arbitrary test run $\sigma \in L'^*$. We have the following properties:

1. $t \rceil\!\| s \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\| s'$ implies $t \rceil\!\| Q(s) \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\| Q(s')$
2. $\mathbf{Sinit}(t \rceil\!\| s) = \mathbf{Sinit}(t \rceil\!\| Q(s))$.

The proposition below proves to be essential in establishing the correctness of our main results in the remainder of Section 3. It essentially establishes the links between the internal behaviors of an implementation in the synchronous and the asynchronous settings.

Proposition 1. *Let $\langle S, L, \rightarrow, s_0\rangle$ be an IOLTS and let $\langle T, L', \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. For all states $t \in T$, $s, s' \in S$, all $\sigma_i \in L_I^*$ and $\sigma_u \in L_U^*$, we have:*

1. $s \stackrel{\epsilon}{\Longrightarrow} s'$ *iff* $t \rceil\!\rceil s \stackrel{\epsilon}{\Longrightarrow} t \rceil\!\rceil s'$ *(RI*)*
2. $[\sigma_u \ll s \ll \sigma_i] \stackrel{\epsilon}{\Longrightarrow} [\sigma_u \ll s' \ll \sigma_i]$ *iff* $s \stackrel{\epsilon}{\Longrightarrow} s'$ *(II*).*

3.3 Sound Verdicts of Internal Choice Test Cases

In [13,6], it is argued that providing inputs to an IUT only after observing quiescence (i.e., in a stable state), eliminates the distortions in observable behavior, introduced by communicating to the IUT using queues. Hence, a subset of synchronous test-cases, namely those which only provide an input after observing quiescence, are safe for testing asynchronous systems. This is summarized in the following (non)theorem from [13,12] (and paraphrased in [6]):

Claim (Theorem 1 in [13]). Let M be an arbitrary IOTS$^{\sqcap}$ with initial state s_0, and let $\langle T, L, \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. Then s_0 **passes** t_0 iff $Q(s_0)$ **passes** t_0.

In [6], the claim is taken for granted, and, unfortunately, in [13,12] only a proof sketch is provided for the above claim; the proof sketch is rather informal and leaves some room for interpretation, as illustrated by the following excerpt:

> "...An implementation guarantees that it will not send any output before receiving an input after quiescence is observed..."

As it turns out, the above result does not hold in its full generality, as illustrated by the following example.

Example 3. Consider the internal choice test case with initial state t_0 in Figure 3. Consider the implementation modeled by the IOTS$^{\sqcap}$ depicted in Figure 2, starting in state o_0. Clearly, we find that o_0 **passes** t_0; however, in the asynchronous setting, $Q(o_o)$ **passes** t_0 does not hold. This is due to the divergence in the implementation, which gives rise to an observation of quiescence in the queued context, but not so in the synchronous setting.

The claim *does* hold for non-divergent internal choice implementations. Note that divergence is traditionally also excluded from testing theories such as **ioco**. In this sense, assuming non-divergence is no restriction. Apart from the following theorem, we tacitly assume in all our formal results to follow that the implementation IOLTSs are non-divergent.

Theorem 1. *Let $M = \langle S, L, \rightarrow, s_0\rangle$ be an arbitrary IOTS$^{\sqcap}$ and let $\langle T, L' \cup \{\tau\}, \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. If M is non-divergent, then s_0* **passes** *t_0 iff $Q(s_0)$* **passes** *t_0.*

Given the pervasiveness of the original (non-)theorem, a formal correctness proof of our corrections to this theorem (i.e., *our* Theorem 1) is highly desirable. In the remainder of this section, we therefore give the main ingredients for establishing a full proof for Theorem 1. We start by establishing a formal correspondence between observations of quiescence in the synchronous setting and observations of *weak* quiescence in the asynchronous setting. (Due to space limit, some proofs are omitted here, but can be found in the technical report [3].)

Lemma 1. *Let $\langle S, L, \rightarrow, s_0\rangle$ be an IOTS$^{\sqcap}$. Let $s \in S$ be an arbitrary state. Then $\delta_q(Q(s))$ implies $\delta(s')$ for some $s' \in S$ satisfying $s \stackrel{\epsilon}{\Longrightarrow} s'$.*

The above lemma results that all inputs a TTS$^{\sqcap}$ gives as stimuli to an implementation, modeled as an IOTS$^{\sqcap}$, can be consumed. Note that this is a non-trivial statement, given that an IOTS$^{\sqcap}$ is not input-enabled in all states. The proposition below as a consequence of the given property, states that every test execution can lead to a state in which both communication queues are empty.

Proposition 2. *Let $\langle S, L, \rightarrow, s_0\rangle$ be an IOTS$^{\sqcap}$, and let $\langle T, L', \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. Assume arbitrary states $t' \in T$ and $s, s' \in S$, and an arbitrary test run $\sigma \in L'^*$. Then for all $\sigma_i \in L_I^*$ and $\sigma_u \in L_U^*$:*

$$t_0 \rceil\!\rceil Q(s) \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\rceil_{[\sigma_u \ll s' \ll \sigma_i]} \text{ implies } \exists s'' \in S \bullet t_0 \rceil\!\rceil Q(s) \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\rceil Q(s'')$$

As a consequence of the above proposition, we find the following corollary. It states that each asynchronous test execution can be chopped into individual observations such that before and after each observation the communication queue is empty.

Corollary 1. *Let $\langle S, L, \rightarrow, s_0\rangle$ be an IOTS$^{\sqcap}$, and let $\langle T, L', \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. Assume arbitrary states $t' \in T$ and $s, s' \in S$, and an arbitrary test run $\sigma \in L'^*$ and $x \in L'$. Then $t_0 \rceil\!\rceil Q(s) \stackrel{\sigma \cdot x}{\Longrightarrow} t' \rceil\!\rceil Q(s')$ implies $\exists t'' \in T, s'' \in S \bullet t_0 \rceil\!\rceil Q(s) \stackrel{\sigma}{\Longrightarrow} t'' \rceil\!\rceil Q(s'') \stackrel{x}{\Longrightarrow} t' \rceil\!\rceil Q(s')$. Moreover, if $x = \theta$ then $\delta_q(Q(s'))$.*

The lemma below establishes a correspondence between the test runs that can be executed in the asynchronous setting and those runs one would obtain in the synchronous setting. The lemma is basic to the correctness of our main results in this section.

Lemma 2. *Let $\langle S, L, \rightarrow, s_0\rangle$ be an IOTS$^{\sqcap}$, and let $\langle T, L', \rightarrow, t_0\rangle$ be a TTS$^{\sqcap}$. Let $s, s' \in S$ and $t' \in T$ be arbitrary states. Then, for all $\sigma \in L'^*$, such that $t_0 \rceil\!\rceil Q(s) \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\rceil Q(s')$, there is a non-empty set $\mathbb{S} \subseteq \{s'' \in S \mid s' \stackrel{\epsilon}{\Longrightarrow} s''\}$ such that*

1. *$\{s'' \in S \mid \delta(s'') \wedge s' \stackrel{\epsilon}{\Longrightarrow} s''\} \subseteq \mathbb{S}$ if $\exists \sigma' \in L'^* \bullet \sigma = \sigma' \cdot \theta$*
2. *$s' \in \mathbb{S}$ if $\nexists \sigma' \in L'^* \bullet \sigma = \sigma' \cdot \theta$*
3. *$\forall s'' \in \mathbb{S} \bullet t_0 \rceil\!\rceil s \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\rceil s''$.*

Proof. We prove this lemma by induction on the length of $\sigma \in L'^*$.

- Induction basis. Assume that the length of σ is 0, i.e., $\sigma = \epsilon$. Assume that $t_0 \rceil\!\rceil Q(s) \stackrel{\epsilon}{\Longrightarrow} t_0 \rceil\!\rceil Q(s')$. By Proposition 1(2) we have $s \stackrel{\epsilon}{\Longrightarrow} s'$. Set $\mathbb{S} = \{s'' \mid s' \stackrel{\epsilon}{\Longrightarrow} s''\}$. Let $s'' \in \mathbb{S}$ be an arbitrary state. Proposition 1(1) leads to $t_0 \rceil\!\rceil s \stackrel{\epsilon}{\Longrightarrow} t_0 \rceil\!\rceil s'$ and $t_0 \rceil\!\rceil s' \stackrel{\epsilon}{\Longrightarrow} t_0 \rceil\!\rceil s''$; by transitivity, we have the desired $t_0 \rceil\!\rceil s \stackrel{\epsilon}{\Longrightarrow} t_0 \rceil\!\rceil s''$. It is also clear that $s' \in \mathbb{S}$. We thus find that $\mathbb{S}$ meets the desired conditions.
- Inductive step. Assume that the statement holds for all σ' of length at most $n-1$. Suppose that the length of σ is n. Assume that $t_0 \rceil\!\rceil Q(s) \stackrel{\sigma}{\Longrightarrow} t' \rceil\!\rceil Q(s')$. By Corollary 1, there is some $s_{n-1} \in S$, a $t_{n-1} \in T$ and $\sigma_{n-1} \in L'^*$ and $x \in L'$, such that $\sigma = \sigma_{n-1} \cdot x$ and $t_0 \rceil\!\rceil Q(s) \stackrel{\sigma_{n-1}}{\Longrightarrow} t_{n-1} \rceil\!\rceil Q(s_{n-1}) \stackrel{x}{\Longrightarrow} t' \rceil\!\rceil Q(s')$.
  By induction, there must be a set $\mathbb{S}_{n-1} \subseteq \{s'' \in S \mid s_{n-1} \stackrel{\epsilon}{\Longrightarrow} s''\}$, such that

1. $\{s'' \in S \mid \delta(s'') \wedge s_{n-1} \stackrel{\epsilon}{\Longrightarrow} s''\} \subseteq \mathbb{S}_{n-1}$ if $\exists \sigma' \in L'^* \bullet \sigma = \sigma' \cdot \theta$
2. $s_{n-1} \in \mathbb{S}_{n-1}$ if $\nexists \sigma' \in L'^* \bullet \sigma = \sigma' \cdot \theta$
3. $\forall s'' \in \mathbb{S}_{n-1} \bullet t_0 \rceil\!\rceil s \stackrel{\sigma_{n-1}}{\Longrightarrow} t_{n-1} \rceil\!\rceil s''$.

We next distinguish three cases: $x \in L_I$, $x \in L_U$ and $x \notin L_I \cup L_U$.

1. Case $x = \theta$. We thus find that $t_{n-1} \rceil\!\rceil Q(s_{n-1}) \stackrel{\theta}{\Longrightarrow} t_n \rceil\!\rceil Q(s')$. As a result of Corollary 1, we have $\delta_q(s')$. We then find as a result of Lemma 1, there must be some state $s'' \in S$ such that $s_{n-1} \stackrel{\epsilon}{\Longrightarrow} s' \stackrel{\epsilon}{\Longrightarrow} s''$ and $\delta(s'')$. Consider the set $\mathbb{S}_n = \{s'' \in S \mid \delta(s'') \wedge s' \stackrel{\epsilon}{\Longrightarrow} s''\}$.
   Let s'' be an arbitrary state in $\mathbb{S}_n$. Distinguish between cases $s_{n-1} \notin \mathbb{S}_{n-1}$ and $s_{n-1} \in \mathbb{S}_{n-1}$. In the case, $s_{n-1} \notin \mathbb{S}_{n-1}$, we know from the construction of $\mathbb{S}_{n-1}$ that $s'' \in \mathbb{S}_{n-1}$ and $s'' \stackrel{\epsilon}{\Longrightarrow} s''$ always holds. In the case $s_{n-1} \in \mathbb{S}_{n-1}$, we have that $s_{n-1} \stackrel{\epsilon}{\Longrightarrow} s' \stackrel{\epsilon}{\Longrightarrow} s''$. We thus find that $\forall s'' \in \mathbb{S}_n \; \exists \bar{s} \in \mathbb{S}_{n-1} \bullet t_0 \rceil\!\rceil s \stackrel{\sigma_{n-1}}{\Longrightarrow} t_{n-1} \rceil\!\rceil \bar{s} \stackrel{\epsilon}{\Longrightarrow} t_{n-1} \rceil\!\rceil s'' \stackrel{\theta}{\longrightarrow} t' \rceil\!\rceil s''$.
   Thus $\mathbb{S}_n$ has the desired requirement that $t_0 \rceil\!\rceil s \stackrel{\sigma_{n-1} \cdot x}{\Longrightarrow} t' \rceil\!\rceil s''$ for all $s'' \in \mathbb{S}_n$. Also, $\{s'' \in S \mid \delta(s'') \wedge s' \stackrel{\epsilon}{\Longrightarrow} s''\} \subseteq \mathbb{S}_n$ is concluded from construction of $\mathbb{S}_n$. Hence, $\mathbb{S}_n$ satisfies all desired conditions.
2. Case $x \in L_I$. By Property 1, we find that the last step in σ_{n-1} must be θ. It follows from corollary 1 that $Q(s_{n-1})$ is weakly quiescent and consequently $\delta_q(s_{n-1})$. By induction we have that $\{s'' \in S \mid \delta(s'') \wedge s_{n-1} \stackrel{\epsilon}{\Longrightarrow} s''\} \subseteq \mathbb{S}_{n-1}$. Consider the set $\mathbb{S}_n = \{s'' \in S \mid s' \stackrel{\epsilon}{\Longrightarrow} s''\}$.
   Transition $t_{n-1} \rceil\!\rceil Q(s_{n-1}) \stackrel{x}{\Longrightarrow} t' \rceil\!\rceil Q(s')$ implies that $s_{n-1} \stackrel{x}{\Longrightarrow} s'$. By Lemma 1 and Definition 6, we know that $\exists \bar{s} \in S$ such that $s_{n-1} \stackrel{\epsilon}{\Longrightarrow} \bar{s} \stackrel{x}{\Longrightarrow} s'$ and $\delta(\bar{s})$. From construction of $\mathbb{S}_{n-1}$, we know that $\bar{s}$ is in $\mathbb{S}_{n-1}$. We thus have $\forall s'' \in \mathbb{S}_n \; \exists \bar{s} \in \mathbb{S}_{n-1} \bullet t_0 \rceil\!\rceil s \stackrel{\sigma_{n-1}}{\Longrightarrow} t_{n-1} \rceil\!\rceil \bar{s} \stackrel{x}{\Longrightarrow} t' \rceil\!\rceil s''$.
   It is clear form construction of $\mathbb{S}_n$ that $s' \in \mathbb{S}_n$ as the required condition that $s' \in \mathbb{S}_n$ if the last step of σ is not θ-labeled transition. We thus find that $\mathbb{S}_n$ fulfills all desired requirements.
3. Case $x \in L_U$. Analogous to the previous case.

We are now in a position to establish the correctness of Theorem 1. We provide the proof below:

Proof (Theorem 1). We prove the theorem by contraposition.

1. Case $\Rightarrow$. Suppose not $Q(s)$ **passes** t_0. By Definition 9 and Proposition 2, $t_0 \rceil\!\rceil Q(s) \stackrel{\sigma'}{\Longrightarrow} \mathbf{fail} \rceil\!\rceil Q(s')$, for some $\sigma' \in L'^*$ and $s' \in S$. As a result of Lemma 2, there is a non-empty set $\mathbb{S} \subseteq \{s'' \in S \mid s' \stackrel{\epsilon}{\Longrightarrow} s''\}$ such that for all $s'' \in \mathbb{S}$, $t_0 \rceil\!\rceil s \stackrel{\sigma'}{\Longrightarrow} \mathbf{fail} \rceil\!\rceil s''$, which was what we needed to prove.
2. Case $\Leftarrow$. Assume, that not s **passes** t_0. Then there are $\sigma' \in L'^*$ and $s'' \in S$, $t_0 \rceil\!\rceil s \stackrel{\sigma'}{\Longrightarrow} \mathbf{fail} \rceil\!\rceil s''$. Using Property 3 leads to $t_0 \rceil\!\rceil Q(s) \stackrel{\sigma'}{\Longrightarrow} \mathbf{fail} \rceil\!\rceil Q(s'')$.

4 Adapting Asynchronous Setting to IOCO

In this section, we re-cast the results of the previous section to the setting with **ioco** test-cases. We first define **ioco** and then show that the results of the previous section

cannot be trivially generalized to the **ioco**-setting. Then using an approach inspired by [7, Chapter 5] and [6], we show how to re-formulate Theorem 1 in this setting.

4.1 Input Output Conformance

The **ioco** testing theory formalizes the conformance of an implementation to its specification. In this theory, implementations are assumed to behave according to an (unkown) IOTS; as a consequence, implementations are assumed to be input enabled. Contrary to implementations, specifications are not required to be input enabled; this facilitates under-specifying the behavior of a system. Informally, the **ioco** conformance relation captures whether the observable behaviors of the implementation are valid observable behaviors, given a specification. The observable behaviors are essentially augmented traces, called *suspension traces*, consisting of inputs, outputs and quiescence.

For a given set of states S of an arbitrary IOLTS with transition relation $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$, suspension traces are defined through an auxiliary transition relation $\Longrightarrow_\delta \subseteq S \times (L \cup \{\delta\})^* \times S$, specified by the following deduction rules:

$$\frac{}{s \overset{\epsilon}{\Longrightarrow}_\delta s} \qquad \frac{s \overset{\sigma}{\Longrightarrow}_\delta s' \qquad \delta(s')}{s \overset{\sigma\delta}{\Longrightarrow}_\delta s'} \qquad \frac{s \overset{\sigma}{\Longrightarrow}_\delta s'' \qquad s'' \overset{x}{\Longrightarrow} s'}{s \overset{\sigma x}{\Longrightarrow}_\delta s'}$$

Henceforth, given an alphabet L, we write L_δ to denote the set $L \cup \{\delta\}$.

Definition 13 (Suspension traces, Out and After). *Let $\langle S, L, \rightarrow, s_0 \rangle$ be an IOLTS. Let $s \in S$ be an arbitrary state, $S' \subseteq S$ and $\sigma \in L_\delta^*$.*

1. *The set of* suspension traces *of s, denoted* $\mathsf{Straces}(s)$ *is the set* $\{\sigma \in L_\delta^* \mid s \overset{\sigma}{\Longrightarrow}_\delta\}$*; we set* $\mathsf{Straces}(S') = \bigcup_{s' \in S'} \mathsf{Straces}(s')$
2. *The* outputs *of s, denoted* $\mathbf{out}(s)$ *is the set* $\{x \in L_U \mid s \overset{x}{\longrightarrow}\} \cup \{\delta \mid \delta(s)\}$*; we set* $\mathbf{out}(S') = \bigcup_{s' \in S'} \mathbf{out}(s')$
3. *The σ-*reachable states *of s, denoted s* $\mathbf{after}\, \sigma$ *is the set* $\{s' \in S \mid s \overset{\sigma}{\Longrightarrow}_\delta s'\}$*; we set* S' $\mathbf{after}\, \sigma = \bigcup_{s' \in S'} s'$ $\mathbf{after}\, \sigma$.

The above abbreviations are used in the intensional characterization of the **ioco** testing relation, given below.

Definition 14 (ioco). *Let $\langle I, L, \rightarrow, i_0 \rangle$ be an IOTS, and let IOLTS $\langle S, L, \rightarrow, s_0 \rangle$ be a specification. We say that implementation i_0 is* input-output conform *specification s_0, denoted i_0* $\mathbf{ioco}$ *s_0, iff*

$$\forall \sigma \in \mathsf{Straces}(s_0) \bullet \mathbf{out}(i_0\ \mathbf{after}\ \sigma) \subseteq \mathbf{out}(s_0\ \mathbf{after}\ \sigma)$$

The **ioco** testing relation has been shown to admit a sound and complete test case generation algorithm, see, e.g., [9]. Soundness means, intuitively, that the algorithm will never generate a test case that, when executed on an implementation, leads to a *fail* verdict if the test runs are in accordance with the specification. Completeness is more esoteric: if the implementation has a behavior that is not in line with the specification, then there is a test case that, in theory, has the capacity to detect that non-conformance.

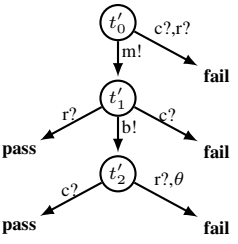


Fig. 4. An **ioco** test case

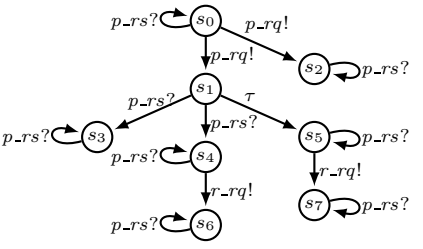


Fig. 5. A delay right-closed IOTS

As the exact workings of the algorithm are impertinent to our main results in this section, we will forego an explanation of it.

In the following example, we motivate that the definitions and the constraints used in the previous section cannot be used for the **ioco** setting.

Example 4. Figure 4 shows a test case for IOLTS o_0 in Figure 2, which is an internal choice IOTS. Assume that at the same time o_0 is also used as the implementation; o_0 is not input-enabled in all states, and making it input-enabled violates the internal choice assumption. In fact, as observed in Section 2, the intersection of IOTSs and internal choice IOTSs only include pathological IOTSs that do not produce any output. For the purpose of this example, we use the theory of **ioco** on internal choice IOTSs nevertheless.

For o_0 as specification and implementation, we have that o_0 **ioco** o_0. However, we can reach a fail verdict for o_0 under the queue context when using the test case t'_0. Consider the sequence $m?b?r!$; in the queue context, the execution $t'_0 \rceil Q(o_0) \xrightarrow{m?} t'_1 \rceil_{[\epsilon \ll o_0 \ll m?]} \overset{\epsilon}{\Longrightarrow} t'_1 \rceil Q(o_1) \overset{\epsilon}{\Longrightarrow} t'_1 \rceil_{[r! \ll o_2 \ll \epsilon]} \xrightarrow{b?} t'_2 \rceil_{[r! \ll o_2 \ll b?]} \xrightarrow{r!} \mathbf{fail} \rceil_{[\epsilon \ll o_2 \ll b?]}$ is possible, which leads to the **fail** state. Note that the fail verdict is reached even if we omit divergence from the implementation o_0. This shows that Theorem 1 cannot be trivially generalized to the **ioco** setting (even when excluding divergence and allowing for non-input-enabled states).

4.2 Synchronizing Theorem for ioco

In this section, we investigate implementations for which **ioco** test cases cannot distinguish between synchronous and asynchronous modes of testing. To this end, we consider the relation between traces of a system and those of the system in queue context.

Definition 15 (Delay relation). *Let L be a finite alphabet partitioned in L_I and L_U. The* delay relation $@ \subseteq L_\delta^* \times L_\delta^*$ *is defined by the following deduction rules:*

$$\frac{}{\sigma \mathbin{@} \sigma}\ REF \qquad \frac{\rho_i, \sigma_i \in L_I^* \qquad \sigma_u \in L_U^*}{\rho_i \cdot \sigma_u \cdot \sigma_i \mathbin{@} \rho_i \cdot \sigma_i \cdot \sigma_u}\ PUSH \qquad \frac{\sigma \mathbin{@} \sigma' \qquad \rho @ \rho'}{\sigma \cdot \rho \mathbin{@} \sigma' \cdot \rho'}\ COM$$

Proposition 3. *Let $\langle S, L, \rightarrow, s_0 \rangle$ be an IOTS. Let $s \in S$ and $\sigma \in L_\delta^*$. Then $\sigma \in$ $\mathsf{Straces}(Q(s))$ implies there is a $\sigma' \in \mathsf{Straces}(s)$ such that $\sigma' \mathbin{@} \sigma$.*

Definition 16 (Delay right-closed IOTS). *Let $M = \langle S, L, \rightarrow, s_0 \rangle$ be an IOTS. A set $L' \subseteq L_\delta^*$ is* delay right-closed *iff for all $\sigma \in L'$ and $\sigma' \in L_\delta^*$, if $\sigma \mathbin{@} \sigma'$ then $\sigma' \in L'$. The IOTS M is* delay right-closed *iff* $\mathsf{Straces}(s_0)$ *is delay right-closed.*

We denote the class of delay right-closed IOTSs ranging over L_I and L_U by $\mathsf{IOTS}^{@}(L_I, L_U)$. The property below gives an alternative characterisation of delay right-closed IOTSs.

Property 4. Let $M = \langle I, L, \rightarrow, i_0 \rangle$ be an IOTS. The IOTS M is *delay right-closed* if for all $\sigma \in L_\delta^*$, all $x \in L_U$ and $a \in L_I$, we have:

$$\sigma \cdot x \cdot a \in \mathsf{Straces}(i_0) \text{ then } \sigma \cdot a \cdot x \in \mathsf{Straces}(i_0)$$

Example 5. Consider the IOTS s_0 given in Figure 5. It is not hard to check that s_0 is delay right-closed.

As stated in the following theorem, the verdicts obtained by executing an arbitrary test case on a delay right-closed IOTS do not depend on the execution context. That is, the verdict does not change when the communication between the implementation and the test case is synchronous or asynchronous.

Theorem 2. *Let $\langle I, L, \rightarrow, i_0 \rangle$ be a delay right-closed IOTS and let $\langle T, L', \rightarrow, t_0 \rangle$ be an arbitrary test case. Then i_0 $\mathbf{passes}$ t_0 iff $Q(i_0)$ $\mathbf{passes}$ t_0.*

Before we address the proof of the above theorem, we first establish the correctness of the lemma below, stating that the suspension traces of a delay right-closed IOTS, as observed in an asynchronous setting are indistinguishable from the set of suspension traces observable in the synchronous setting.

Lemma 3. *Let $\langle S, L, \rightarrow, s_0 \rangle$ be a delay right-closed IOTS. Then* $\mathsf{Straces}(Q(s_0)) = \mathsf{Straces}(s_0)$.

Proof. We divide the proof obligation into two parts: $\mathsf{Straces}(Q(s_0)) \subseteq \mathsf{Straces}(s_0)$ and $\mathsf{Straces}(s_0) \subseteq \mathsf{Straces}(Q(s_0))$. It is not hard to verify that the latter holds vacuously, even for arbitrary IOTSs.

It therefore remains to show that $\mathsf{Straces}(Q(s_0)) \subseteq \mathsf{Straces}(s_0)$. Consider a $\sigma \in \mathsf{Straces}(Q(s_0))$; by Proposition 3, $\exists \sigma' \in \mathsf{Straces}(s_0) \bullet \sigma' \mathbin{@} \sigma$. As s_0 is delay right-closed, we obtain the required $\sigma \in \mathsf{Straces}(s_0)$.

The above lemma is at the basis of the correctness of Theorem 2.

Proof (Theorem 2). Using the lemma given above, the proof of the theorem follows from the observation that for all test cases $\langle T, L', \rightarrow, t_0 \rangle$ and all $\sigma \in L'^*$:

$$\exists i' \in I \bullet t_0 \rceil i_0 \stackrel{\sigma}{\Longrightarrow} \mathbf{fail} \rceil i' \text{ iff } \exists i' \in I, \sigma_i \in L_I^*, \sigma_u \in L_U^* \bullet t_0 \rceil Q(i_0) \stackrel{\sigma}{\Longrightarrow} \mathbf{fail} \rceil_{[\sigma_u \ll i' \ll \sigma_i]}$$

Theorem 3. *Let $\langle I, L, \rightarrow, i_0 \rangle$ be a delay right-closed IOTS and let IOLTS $\langle S, L, \rightarrow, s_0 \rangle$ be a specification. Then i_0 $\mathbf{ioco}$ s_0 iff $Q(i_0)$ $\mathbf{ioco}$ s_0.*

Proof. Follows from the existence of a sound and complete test suite that can test for **ioco**, and the proof of Theorem 2.

5 Necessary and Sufficient Conditions

In the previous section, we presented a class of implementation, called delay right-closed, whose synchronous and asynchronous test executions lead to the same verdict. We now show that delayed right-closedness of implementations is also a necessary condition to ensure the same verdict in the synchronous and the asynchronous setting.

Theorem 4. *Let* $M = \langle I, L, \rightarrow, i_0 \rangle$ *be an IOTS. If for every test case* $\langle T, L', \rightarrow, t_0 \rangle$, *we have* i_0 **passes** $t_0 \Leftrightarrow Q(i_0)$ **passes** t_0, *then* M *is a delay right-closed IOTS.*

Proof. We prove the theorem by contraposition, i.e., we show that if we test a non-delay right-closed IOTS, there is a test case that can detect this by giving a *pass* verdict in the synchronous setting but a *fail* verdict in the asynchronous setting.

Let $\langle I, L, \rightarrow, i_0 \rangle$ be an IOTS that is not delay right-closed. Thus, there is some $x \in L_U$, $a \in L_I$ such that $\sigma \cdot x \cdot a \in \mathsf{Straces}(i_0)$, but not $\sigma \cdot a \cdot x \in \mathsf{Straces}(i_0)$. Let $\langle T, L', \rightarrow, t_0 \rangle$ be a test case such that there is a $t' \in T$ satisfying:

1. $t_0 \stackrel{\sigma}{\Longrightarrow} t'$,
2. $t' \stackrel{a}{\longrightarrow} t''$, and $t'' \stackrel{x}{\longrightarrow}$ **fail**.
3. for all σ' such that $t_0 \stackrel{\sigma'}{\Longrightarrow}$ **fail** we have $\sigma' = \sigma \cdot a \cdot x$.

Observe that the existence of such a test case is immediate. Then there are $\sigma_i \in L_I^*$, $\sigma_u \in L_U^*$ and a state $i \in (i_0 \,\mathbf{after}\, \sigma)$ such that $t_0 \rceil Q(i_0) \stackrel{\sigma \cdot a \cdot x}{\Longrightarrow} \mathbf{fail} \rceil_{[\sigma_u \ll i_{\ll \sigma_i \cdot a]}}$, i.e., not $Q(i_0)$ **passes** t_0. However, we do not have $t_0 \rceil i_0 \stackrel{\sigma \cdot a \cdot x}{\Longrightarrow} \mathbf{fail} \rceil i$. By construction of the test case, we find that i_0 **passes** t_0.

6 Conclusions

In this paper, we presented theorems which allow for using test-cases generated from ordinary specifications in order to test asynchronous systems. These theorems establish sufficient conditions when the verdict reached by testing the asynchronous system (remotely, through FIFO channels) corresponds with the local testing through synchronous interaction. In the case of **ioco** testing theory, we show that the presented sufficient conditions are also necessary.

It remains to find an intensional characterization of the notion of conformance induced by the class of test-cases generated in the approach of [13]. The presented conditions for synchronizing **ioco** are semantic in nature and we intend to formulate syntactic conditions that imply the semantic conditions presented in this paper. For example, it is interesting to find out which composition of programming constructs and / or patterns of interaction satisfy the constraints established in this paper. The research reported in this paper is inspired by our practical experience with testing asynchronous systems reported in [1]. We plan to apply the insights obtained from this theoretical study to our practical cases and find out to what extent the constraints of this paper apply to the implementation of our case studies.

Acknowledgments. We would like to thank Sjoerd Cranen (TU/e) and Maciej Gazda (TU/e) for their useful comments and suggestions.

References

1. Asadi, H.R., Khosravi, R., Mousavi, M.R., Noroozi, N.: Towards model-based testing of electronic funds transfer systems. In: Proc. of FSEN 2011. LNCS. Springer, Heidelberg (2011)
2. Jard, C., Jéron, T., Tanguy, L., Viho, C.: Remote testing can be as powerful as local testing. In: Proc. of FORTE XII. IFIP Proc., vol. 156, pp. 25–40. Kluwer, Dordrecht (1999)
3. Noroozi, N., Khosravi, R., Mousavi, M.R., Willemse, T.A.C.: Synchronizing Asynchronous Conformance Testing. Computer Science Report, no. 11-10, 16 pp. Technische Universiteit Eindhoven, Eindhoven (2011)
4. Petrenko, A., Yevtushenko, N.: Queued testing of transition systems with inputs and outputs. In: Proc. of FATES 2002, pp. 79–93 (2002)
5. Petrenko, A., Yevtushenko, N., Huo, J.: Testing transition systems with input and output testers. In: Hogrefe, D., Wiles, A. (eds.) TestCom 2003. LNCS, vol. 2644, pp. 129–145. Springer, Heidelberg (2003)
6. Simao, A., Petrenko, A.: From test purposes to asynchronous test cases. In: Proc. of ICSTW 2010, pp. 1–10. IEEE CS, Los Alamitos (2010)
7. Tretmans, J.: A formal Approach to conformance testing. PhD thesis, Univ. of Twente, The Netherlands (1992)
8. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools 3, 103–120 (1996)
9. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
10. Tretmans, J., Verhaard, L.: A queue model relating synchronous and asynchronous communication. In: Proc. of PSTV 1992. IFIP Tr., vol. C-8, pp. 131–145. North-Holland, Amsterdam (1992)
11. Verhaard, L., Tretmans, J., Kars, P., Brinksma, E.: On asynchronous testing. In: Proc. of IWPTS 1993. IFIP Tr., vol. C-11, pp. 55–66. North-Holland, Amsterdam (1993)
12. Weiglhofer, M.: Automated Software Conformance Testing. PhD thesis, TU Graz (2009)
13. Weiglhofer, M., Wotawa, F.: Asynchronous input-output conformance testing. In: Proc. of COMPSAC 2009, pp. 154–159. IEEE CS, Los Alamitos (2009)

Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications

Kosuke Ono[1], Yoichi Hirai[1], Yoshinori Tanabe[2],
Natsuko Noda[3], and Masami Hagiya[1]

[1] University of Tokyo
[2] National Institute of Informatics, Japan
[3] NEC Corporation

Abstract. Hadoop MapReduce is a framework for distributed computation on key-value pairs. The goal of this research is to verify actual running code of MapReduce applications. We first constructed an abstract model of MapReduce computation with the proof assistant Coq. In the model, mappers and reducers in MapReduce computation are modeled as functions in Coq, and a specification of a MapReduce application is expressed in terms of invariants among functions involving its mapper and reducer. The model also provides modular proofs of lemmas that do not depend on applications. To achieve the goal, we investigated the feasibility of two approaches. In one approach, we transformed verified mapper and reducer functions into Haskell programs and executed them under Hadoop Streaming. In the other approach, we verified JML annotations on Java programs of the mapper and reducer using Krakatoa, translated them into Coq axioms, and proved Coq specifications from them. In either approach, we were able to verify correctness of MapReduce applications that actually run on the Hadoop MapReduce framework.

1 Formalizing MapReduce Applications

As a method for treating a large amount of data, the MapReduce programming paradigm [4] has obtained much attention. One reason for its popularity is the Apache Hadoop project[1], which releases an open-source MapReduce framework.

The purpose of this research is to develop methods to guarantee that an application on the MapReduce framework satisfies a specification, using formal methods from software engineering. Unlike related work, our goal is to verify actual, running code of a MapReduce application. To achieve the goal, we investigated the feasibility of two approaches. In one approach, we transform verified mapper and reducer functions into Haskell programs and executed them under Hadoop Streaming. In the other approach, we write specifications of the mapper and reducer functions as annotations in Java Modeling Language (JML) [10] on hand-written Java programs of those functions, and use Krakatoa [11] to verify that the annotated programs satisfy the specifications.

[1] http://hadoop.apache.org/

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 350–365, 2011.

Both approaches are based on an abstract model of Hadoop MapReduce computation and specifications of Hadoop libraries formalized with the proof assistant system Coq [1]. In either approach, we were able to verify correctness of a MapReduce application that runs under the Hadoop MapReduce framework. The Hadoop libraries provide their features with some informal specifications, for example what is given to mappers, combiners, and reducers as input, and how their output is processed. In this research, we formalized the informal specifications in Coq. In addition, specifications of the mapper, combiner and reducer of an application are also given in Coq. Using these specifications of the Hadoop libraries and the application specific functions, we can prove the desired specification of the application, also formalized in Coq. In such proofs, we often need to write lemmas in some specific patterns such as handling invariants and lifting properties from lower level functions to integrated functions. To ease the burden, we developed a Coq library based on the module system of Coq [3]. In the next section, we describe details of the formal proof in Coq. For simplicity, we only treat cases where the combiner and reducer functions are identical. In the next section and later, the combiner does not appear explicitly.

The contributions of this research are summarized as follows.

- We constructed a Coq model of the Hadoop MapReduce framework by formalizing the informal specifications of the Hadoop libraries.
- Based on the model, we formally described correctness of MapReduce applications, and verified typical applications, WordCount and InvertedIndex. The verification is modular in that it consists of common parametrized proofs and instantiations of parameters for each application.
- We took two approaches for verifying actual application code in accordance with its specification.

To our knowledge, it is the first attempt to formally verify actual running code of MapReduce applications.

The rest of the paper is organized as follows. The next section formalizes MapReduce applications in Coq, including WordCount and InvertedIndex, after presenting the abstract model of Hadoop MapReduce computation. Section 3 explains the first approach in which Haskell programs are extracted from Coq proofs and executed under Haskell_Hadoop and Hadoop Streaming. In Section 4, verification of Java programs by Krakatoa in the second approach is described in detail. Section 6 discusses related work.

2 Formalizing MapReduce Applications in Coq

MapReduce [4,9] is a framework for distributed computation involving the mapper function that transforms input key-value pairs to intermediate key-value pairs, and the reducer function that takes a number of pairs sharing the same key and produces the final result of the computation. There is an open-source implementation called Hadoop MapReduce [14]. We made an abstract model of Hadoop MapReduce computation, and verified applications called WordCount and InvertedIndex. They are the most typical MapReduce applications [8].

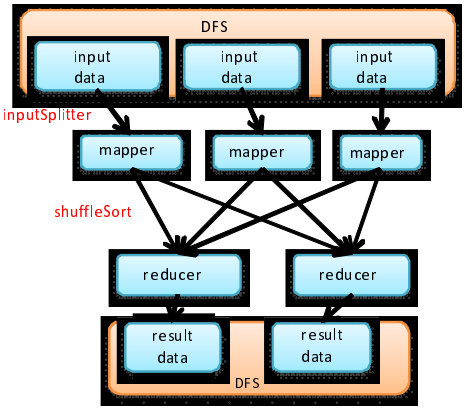


Fig. 1. MapReduce computation

Figure 1 shows MapReduce computation on the Hadoop platform in general. In our functional view of MapReduce computation, input data in DFS (Distributed File System) are split by the function named inputSplitter and fed to the mapper function as key-value pairs. The key-value pairs produced by the mapper function are then sorted and combined by the function named shuffleSort which aggregates the values having the same key and sends the key and the list of the values to the reducer. The final result of the reducer is stored in DFS.

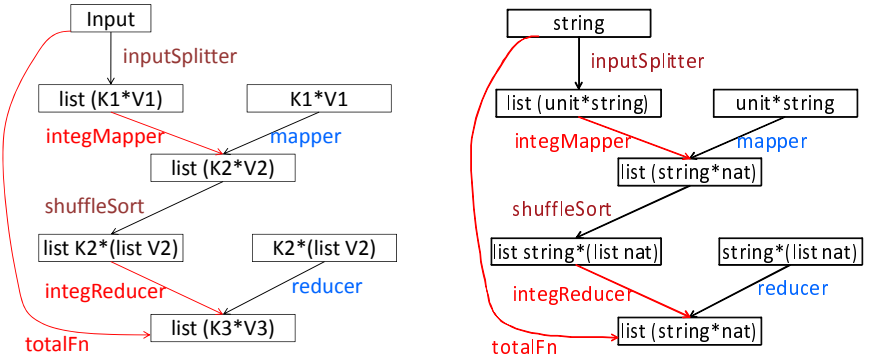
The type of the mapper function is `K1*V1->list (K2*V2)`, where `K1`, `V1`, `K2` and `V2` are type parameters that depend on applications. `K1*V1` denotes the type of a pair that consists of an element from `K1` and one from `V1`, and `list (K2*V2)` denotes the type of a list consisting of pairs in `K2*V2`. The type of `shuffleSort` is `list (K2*V2) -> list (K2*(list V2))`. Each element in a list produced by `shuffleSort` is fed to the reducer function whose type is `K2*(list V2) -> list (K3*V3)`.

For verifying such applications, we developed an abstract model of Hadoop MapReduce computation in Coq. The model consists of specifications of the Hadoop libraries and lemmas independent of applications, which do not depend on applications, with their proofs. The proofs are parametrized in the sense that they contain parameters that depend on applications, including the type parameters mentioned above, the mapper and reducer functions themselves, and proofs of the assumptions on which the parametrized proofs depend.

In the rest of this section, we briefly explain the parametrization and the proofs of the two applications WordCount and InvertedIndex.

2.1 Abstract Model of Hadoop MapReduce Computation

Figure 2(a) illustrates our model of the MapReduce computation under the Hadoop framework. We define an entire MapReduce application as the function `totalFn` which depends on the `mapper` and `reducer` functions given for the application. The other functions in the figure model the internal structure



(a) Abstract model (b) Concretized model for WordCount

Fig. 2. Coq model of MapReduce computation

of the Hadoop library, and they are common to all applications. The function `inputSplitter` splits its input into many parts, and constructs a list of key-value pairs. The list is passed to a function `integMapper`, which invokes the `mapper` function for each key-value pair. The output of `integMapper` is the concatenation of the output lists from the invoked `mapper` function. It is then given to `shuffleSort`, which, for each key, gathers all the values and makes a pair of the key and the list of gathered values, and produces the list of all such pairs as its output, as mentioned previously. Then the output is given to a function called `integReducer`. Similar to `integMapper`, it invokes the `reducer` function for each pair, and concatenates the output from the `reducer` as its own output.

The specification of the MapReduce application is formalized in terms of invariants expressed as functions to the type `Inv`, which also depends on the application. In Figure 3, there are four functions to `Inv`. The function `sifInput` is given the input to the whole MapReduce computation and return `Inv`. The function `sifKV3` is given a pair of `K3*V3` and returns `Inv`. It is lifted to the function from `list (K3*V3)` to `Inv`. Here, lifting means defining a function from `list (K3*V3)` to `Inv` as a fold operation using `sifKV3`. And the lifted function is combined with the function `totalFn`. Correctness of the application is defined as the equality between two functions, `sifInput` and the function thus obtained from `totalFn` and `sifKV3`. The other functions, `sifKV1` and `sifKV2`, are defined for interpolated the gap between the two functions, i.e., the correctness is proven step by step by setting appropriate lemmas and proving those lemmas.

The following code fragment is taken from the parametrized proofs. The first lemma says that `ifLKV1` is equal to the composition of `ifLKV2` and `integMapper`, where `ifLKV1` is the result of lifting `sifKV1` to lists.

```
Definition ifInput :Input -> Inv := sifInput.
Definition ifLKV1 :list (Key1 * Value1) -> Inv := lift sifKV1.
Definition ifLKV2 :list (Key2 * Value2) -> Inv := lift sifKV2.
Definition ifLK2LV2 :list (Key2 * list (Value2)) -> Inv
  := lift (valAgg sifKV2).
```

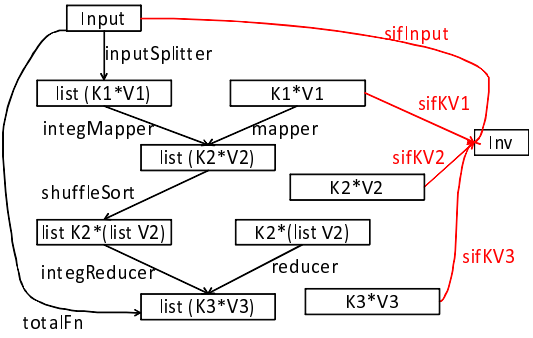


Fig. 3. Handling invariants

```
Definition ifLKV3 :list (Key3 * Value3) -> Inv := lift sifKV3.

Lemma corIntegMapper: forall lkv: list (Key1 * Value1),
  ifLKV1 lkv = ifLKV2 (integMapper lkv).
Proof.
  intros; unfold ifLKV1, ifLKV2, integMapper.
  rewrite <- (lift_comm sifKV1); auto.
  apply sCorMapper.
Qed.

Lemma corIntegReducer: forall lklv: list (Key2 * (list Value2)),
  ifLK2LV2 lklv = ifLKV3 (integReducer lklv).
Proof.
  intros; unfold ifLK2LV2, ifLKV3, integReducer.
  rewrite <- (lift_comm (valAgg sifKV2)); auto.
  apply sCorReducer.
Qed.
```

In order to lift an operation on elements to an operation on lists, we need a monoid structure that depends on applications. As shown in the following code, `MonoidParam` is defined as the module type of monoids. For each application, a concrete monoid is to be defined as a module of type `MonoidParam`.

```
Module Type MonoidParam.
  Parameter M: Set.
  Parameter zero: M.
  Parameter mnplus: M -> M -> M.
  Axiom assoc: forall a b c: M,
    mnplus (mnplus a b) c = mnplus a (mnplus b c).
  Axiom comm: forall a b: M, mnplus a b = mnplus b a.
  Axiom mnplus_zero_l: forall a: M, mnplus zero a = a.
End MonoidParam.
```

In the rest of this section, we explain how we verified two concrete applications: WordCount and InvertedIndex.

2.2 Proving WordCount Specifications

WordCount is an application that counts the occurrences of words in a given input file. Since a file is represented as a string, the actual type of `Input` for WordCount is `string`. We define `K1` as `unit` and `V1` as `string` because the key to the mapper function of WordCount is not used. Since the `mapper` function produces a list of pairs of a string and a natural number, we define `K2` as `string` and `V2` as `nat`. The `reducer` function produces a list of a pair (i.e. a singleton) of a string and a natural number. We define `K3` as `string` and `V3` as `nat`. Please refer to Figure 2(b).

In order to verify an application in Coq, we prepare implementation of the mapper and reducer functions, and a specification of the application, i.e., the `Inv` type and appropriate functions to `Inv`. We then prove that the implementation satisfies the specification step by step. In the case of WordCount, the actual types of `Inv` is `string -> nat` and that of `sifInput` is `wordCountStr`, which is defined as follows:

```
Definition wordCountStr : string -> string -> nat :=
    fun input => wordCountList (tokenizer input).
```

where `tokenizer` is a function that divides the input string into a list of "words." This property could be expressed as a specification of the function `tokenizer` with a suitable definition of words, but we do not take this approach because the definition may vary on the environment, such as the operating system and the locale. Rather, we consider that `tokenizer` implicitly defines what a word is. This is convenient because standard tokenizers are provided in many environments.

In this case study, we need to express a fact that `inputSplitter` does not split the input in the middle of a word. Using `tokenizer`, it can be expressed by the following axiom.

```
Axiom tokenizer_inputSplitter: forall (s: string),
  tokenizer s = fold_right (@app string) nil
    (map (fun us => tokenizer (snd us)) (inputSplitter s)).
```

It means that "the return value of `tokenizer` for the input is identical to the concatenation of the return values of `tokenizer` for each chunk of the input divided by `inputSplitter`.

```
Definition natMulPlus (ln:list nat): nat := fold_right plus 0 ln.
Definition wordCountList: list string -> string -> nat :=
    fun lw => fun word =>
      natMulPlus (map (fun w => if string_dec w word then 1 else 0) lw).
```

The function `wordCountStr` takes an input value of the `string` type representing a file and another `string` value representing a word, and returns a value of the `nat` type. The function divides an input string into a list of words, and then counts up if each element of the word list is equal to the given string.

The `mapper` and `reducer` functions are defined as follows.

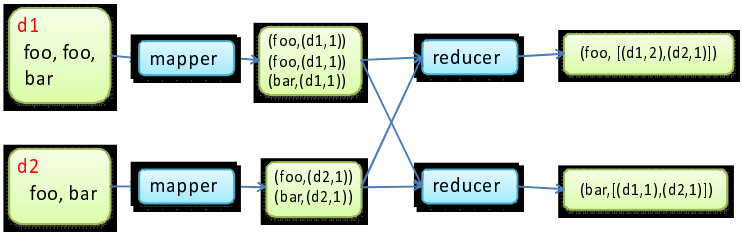


Fig. 4. Application InvertedIndex

```
Definition mapper (input: unit*string): list (string*nat):=
  match input with (u,s) => map (fun w => (w,1)) (tokenizer s) end.

Definition reducer (red_input: string*(list nat)): list (string*nat):=
  match red_input with (w,ln) => (w,fold_right plus 0 ln)::nil end.
```

The `mapper` function takes an input key-value pair, divides the string data into words, and emits a list of intermediate key-value pairs. Each word serves as a key and the constant one serves as a value. The `reducer` function sums up the numbers in a given list associated with a given string and emits a result list of key-value pairs.

The following three lemmas are used to prove the following theorem, which guarantees that the WordCount application meets its specification. The lemmas `sCorMapper` and `sCorReducer` are referred to in the lemmas `corIntegMapper` and `corIntegReducer` in the previous section, respectively. The theorem is derived from these lemmas. Here, some symbols like `WCBasis.mapper` consist of a module name and a module-local name.

```
Lemma sCorInputSplitter: forall i: string,
  sifInput i = lift sifKV1 (TextInput.inputSplitter i).

Lemma sCorMapper: forall kv: unit * string,
  sifKV1 kv = mulPlus (map sifKV2 (WCBasis.mapper kv)).

Lemma sCorReducer: forall klv: string * list nat,
  (valAgg sifKV2) klv = mulPlus (map sifKV3 (WCBasis.reducer klv)).

Theorem correct: forall (s w: string) (n: nat),
  n > 0 -> (n = wordCountStr s w <-> In (w, n) (WCMRSystem.totalFn s)).
```

2.3 Proving InvertedIndex Specifications

InvertedIndex is an application similar to WordCount. InvertedIndex counts how many times each word occurs in each file, while WordCount counts how many times each word occurs in all files. (Figure 4)

First, we prepared the specification of the InvertedIndex application as follows. The `Input` type for this application is `list (string*string)`. The `Inv` type is `string -> string -> nat`, and the actual function for `sifInput` is called `InvertedIndex`, which takes a list of string pairs as input, and returns a function

of the Inv type. The first element of each pair in the list represents a file ID, and the second element represents the contents of the file. The function returned by InvertedIndex takes a word and a file ID, and returns the number of occurrences of the word in the file.

```
Definition IICountList (lfw:list (string*string))
                       (word' file':string) : nat :=
  natMulPlus (map (fun fw => if (string_dec (snd fw) word')
                             then (if (string_dec (fst fw) file')
                                   then 1 else 0)
                             else 0) lfw).

Definition InvertedIndex (input: list (string*string)) (word file:string): nat:=
  IICountList (flat_map (fun fs => map (fun w:string => (fst fs,w))
(tokenizer (snd fs))) input) word file.
```

The mapper and reducer functions are defined as follows.

```
Definition mapper (input: string*string): list (string*(string*nat)) :=
  match input with (f,s) => map (fun w => (w,(f,1))) (tokenizer s) end.

Definition reducer (red_input:string*(list (string*nat))) :
    list (string*(list (string*nat))):=
  match red_input with (w,lsn) => (w, group_snd lsn) :: nil end.
```

where group_snd is a function that receives a list of pairs of a string key and a natural number value, calculates the sum of the corresponding values for each key, and returns the list of pairs of the key and the sum.

The following are the lemmas and the theorem correspond to those of WordCount at the end of the previous section. The strategy for proving them is almost identical with that for the WordCount application.

```
Lemma sCorInputSplitter: forall i: list (string*string),
  sifInput i = lift sifKV1 (TextInput.inputSplitter i).

Lemma sCorMapper: forall kv: string * string,
  sifKV1 kv = mulPlus (map sifKV2 (IIBasis.mapper kv)).

Lemma sCorReducer: forall klv: string * (list (string*nat)),
  (valAgg sifKV2) klv = mulPlus (map sifKV3 (IIBasis.reducer klv)).

Theorem correct: forall (input:list (prod string string))
                        (f: string) (n: nat)
                        (wlfn:prod string (list (prod string nat))),
  (sif_str_nat' wlfn (fst wlfn) f = InvertedIndex input (fst wlfn) f
   -> sif_str_nat' wlfn (fst wlfn) f  > 0
   -> In (fst wlfn, snd wlfn) (IIMRSystem.totalFn input))
  /\ (In (fst wlfn,snd wlfn) (IIMRSystem.totalFn input)
      -> sif_str_nat' wlfn (fst wlfn) f
         = InvertedIndex input (fst wlfn) f).
```

3 Extracting Haskell Programs from Coq Proofs

Coq officially supports extraction of OCaml, Scheme and Haskell programs from Coq proofs. Using this extraction functionality, we can execute function

```
Extraction Language Haskell.
Extract Constant Word => "String"
Extract Constant Weq => "(Prelude.==)".
Extract Inductive nat => Int {"0" "succ"}
Extract Inductive list => "([])" ["[]" "(:)"].
Extract Inductive prod => "(,)" [ "(,)" ].
Extract Inductive unit => "()" [ "()" ].
Extract Inductive string => "String" ["String" "String"].
Extract Constant string_dec => "Prelude.==".
Extract Constant tokenizer => "words".
```

Fig. 5. Directives for extraction

definitions verified by Coq under the real MapReduce framework. Our first approach uses this functionality.

We chose Haskell [12] as the target language of program extraction from Coq proofs because the lazy evaluation order supported by Haskell makes the automatically extracted mapper and reducer functions comparable in performance with their ordinary implementation in Java. If we used OCaml or Scheme, which do not support lazy evaluation, the extracted code can unnecessarily consume a huge amount of memory for storing the whole input and output in the memory. Of course, careful programmers can avoid such a wrong style of programming writing. However this time, the programs are generated automatically without intervention of resource conscious programmers.

Extraction of Haskell programs. In order to extract Haskell programs from Coq proofs, it is necessary to specify some directives to the extractor of Coq. In the case of WordCount, we used the directives shown in Figure 5. Haskell provides a tokenizer function called `words`.

Since the Coq extractor does not allow to convert user-defined function, we further replace the plus function with the `+` operator in Haskell.

Hadoop Streaming and Haskell_Hadoop. Hadoop Streaming [14] is an interface library distributed with Hadoop. It enables programming languages other than Java to describe a MapReduce application. In Hadoop Streaming, both the mapper and reducer functions are supposed to receive data from the standard input and produce data to the standard output. Haskell_Hadoop[2] is a library that allows Haskell programs to run on Hadoop Streaming. In Haskell_Hadoop, the type `Map` of the mapper function and the type `Reduce` of the reducer function are defined as `String -> [String]` and `String -> [String] -> [String]`, respectively. Their definitions had to be modified as follows to make the extracted programs work:

```
type Map = (WcK1, WcV1) -> [(WcK2, WcV2)]
type Reduce = (WcK2, [WcV2]) -> [(WcK3, WcV3)]
```

In the case of WordCount, `WcK1`, `WcV1`, `WcK2`, `WcV2`, `WcK3` and `WcK3` are defined as `unit`, `string`, `string`, `int`, `string` and `int`, respectively. We have prepared conversion functions for input and output of the mapper and reducer functions.

[2] https://github.com/paulgb/haskell_hadoop

4 Verification of Java Programs by Krakatoa

In this section, we explain the second approach: application code written in Java is formally verified. Under the Hadoop framework, mappers and reducers (and combiners) are typically written in Java as methods of interfaces defined in the Hadoop libraries. In this approach, we should verify that the methods of the mapper and the reducer implemented in Java satisfy the properties formalized in Coq. We take the following steps.

1. Describe the properties the Java methods should satisfy in the Java Modeling Language (JML) [10].
2. Apply static analysis to the Java methods and obtain the proof obligations (the sufficient conditions for the methods to satisfy the properties described in JML).
3. Try proving the obligations automatically using the Krakatoa interface to different theorem provers.
4. Prove the remaining obligations by hand.
5. Transform the properties described in JML into Coq formulas.
6. Prove the properties given in Coq from the above formulas.

Details of each step are explained below.

JML has been widely used for describing specifications of Java programs expressed as pre-conditions and post-conditions for methods. A number of tools that support JML have been developed, such as ESC/Java2 [2] and Krakatoa [11]. They generate proof obligations for verifying that a method in question obeys the contract, i.e., if the pre-conditions are satisfied when the method is called, then the post-conditions should be satisfied when the execution of the method is completed. In this research, we use Krakatoa, because it not only supports JML in the above-mentioned aspects, but also generates proof obligations as lemmas in Coq, which is suitable for our approach.

Although our approach is considered applicable to a wide range of MapReduce applications, we concentrate on the WordCount application in this paper so that the reader easily understands the concepts from the example.

Java Code and JML specification. The `map` and `reduce` methods for the WordCount application are shown in Figure 6. There are differences between the above Java implementation and the Coq model described in Section 2. In the Coq model, we assume that key-value pairs are given to the `mapper_ext` and `reducer_ext` functions in the form of a list, while an iterator is given to the methods `map` and `reduce` in Java. Moreover, the `mapper_ext` and `reducer_ext` functions in the Coq model return a list, while the Java code produces key-value pairs one by one by calling the method `collect`.

To fill the gap, we have added a virtual field, whose value is a list, to the iterators handled in the Java code. For example, the input to the reducer is given as an argument named `values`, which is an iterator of the type `IteratorLw` and iterates on `LongWritable`. The iterator has two main methods. One is `hasNext`,

```
public void map(Object key, Text line,
      OutputCollector output,
      Reporter reporter) throws IOException {
  LongWritable one = new LongWritable(1);
  Text word = new Text();
  String str = line.toString();
  StringTokenizer tnz
        = new StringTokenizer(str);
  while (tnz.hasMoreTokens()) {
    word.set(tnz.nextToken());
    output.collect(word, one);
  }
}
```

```
public void reduce(Text key,
      IteratorLw values,
      OutputCollectorTextLw output,
      Reporter reporter) throws IOException {
  long sum = 0;
  while (values.hasNext() == true) {
    LongWritable v = values.next();
    sum += v.get();
  }
  output.collect(key, new LongWritable(sum));
}
```

Fig. 6. The `map` and `reduce` methods

```
type list;

logic list nil();
logic LongWritable headLw(list ls);
logic list tail(list ls);

logic integer totalLw(list ls);
```

```
axiom ax1: totalLw(nil()) == 0;

predicate sp1{L}(list l);
axiom sp1_def{L}: \forall list l;
  sp1{L}(l) <==>
    (totalLw(l) == headLw(l).value
                   + totalLw(tail(l)));
axiom ax2{L}:  \forall list l; sp1{L}(l);
```

Fig. 7. List Type for JML

which returns true if and only if the iterator has not yet visited all members. The method `next` returns the current member, whose type is `LongWritable`, and visits the next member. For this iterator, we have introduced a virtual field that holds the list of members that have not yet been visited. Krakatoa (or rather JML) supports such virtual fields, but does not have built-in list types. We therefore have defined our own version of a list type and its basic operations as in Figure 7. It also defines a function called `totalLw`, which computes the sum of the integers (the values of `LongWritable` objects) in a list.

Using the list type thus defined, we can naturally describe the specification of the iterator as shown in Figure 8. The virtual field is declared as a model field with the name `lst`, using the keyword `model`. In the JML specifications for methods `next` and `hasNext`, the keywords `requires` and `ensures` specify the pre-conditions and post-conditions of the method, respectively. The keyword `assigns` describes which fields in the class are modified by the method. The keyword `\result` refers to the value returned by the method, and `\old` denotes the corresponding value at the time the method is called. It is easy to see that the pre- and post-conditions added to the `next` and `hasNext` methods in the following code express the above-mentioned properties.

For the mapper and reducer specifications, we further defined predicates shown in Table 1. Using the predicates, the mapper and reducer specifications are described as in Table 2.

Generation of proof obligations. In order to show that the implementation of the `map` and `reduce` methods satisfies the specifications described in the previous section, we need to prove proof obligations generated by Krakatoa. Verification of the methods is completed by giving formal proofs to proof obligations.

```
public class IteratorLw {
  //@ model list lst;

  /*@ requires lst != nil();
    @ assigns lst;
    @ ensures tail(\old(lst)) == lst
    @     && \result == headLw(\old(lst));
    @*/
  public LongWritable next();

  /*@ assigns \nothing;
    @ ensures \result <==> (lst != nil());
    @*/
  public boolean hasNext();
}
```

Fig. 8. Specifications for an iterator class. JML specifications for other classes are defined in a similar manner and their details are omitted here.

In general, generating proof obligations for a Java method requires a loop invariant for each loop in the method. Since Krakatoa does not provide support for generating loop invariants automatically, they should be given by hand. In the case of our `map` and `reduce` methods, it is relatively easy to find appropriate loop invariants by examining their post-conditions. The details are omitted here because they are not necessary for the following discussions.

The generated proof obligations are classified into two kinds:

(A) Those directly related to the post-conditions and the loop invariants.
(B) Those related to safety of Java code. They guarantee that execution of the code does not lead to operations that are not allowed in the Java language, such as field access on the null object, array access beyond boundary, etc.

In this case study, Krakatoa generated 40 obligations in total. The breakdown is: 14 in (A) and 10 in (B) for `map`, and 8 in both (A) and (B) for `reduce`.

Proving proof obligations. The next step is to prove the proof obligations. As much automation as possible is desirable. A theorem prover Alt-Ergo can be used for this purpose. Among the 40 proof obligations in the previous section, Alt-Ergo succeeded in proving 35 obligations, but could not prove the remaining five obligations; four in (A) for `map` and one in (B) for `reduce`.

Krakatoa can also use other theorem provers like Simplify [5] and Z3 [13], but none was able to solve a proof obligation that Alt-Ergo failed to solve.

As a last resort, Krakatoa can translate proof obligations into Coq. We applied this translation to the proof obligations that the theorem prover failed to solve, and finally proved all the translated obligations in Coq by hand.

Table 1. Auxiliary functions and predicates

Name	Kind	Meaning
`numAppear1(s,w)`	function	The number of occurrences of word `w` in string `s`.
`numAppear2(l,w,c)`	function	The number of occurrences of the pair `(w,c)` in `l`, which is a list of string-integer pairs.
`nonNullList(l)`	predicate	List `l` does not contain `null` as an element.
`headTLText(l)`	function	The string part of the first string-integer pair in `l`, which is a list of string-integer pairs.
`headTLLw(l)`	function	The integer part of the first string-integer pair in `l`, which is a list of string-integer pairs.

From JML to Coq. In order to locate the above verification result in the whole process of verifying a MapReduce application, it is necessary to translate the JML specifications of the `map` and `reduce` method into axioms in Coq. In this example on the WordCount application, we can proceed as follows.

Let us first examine the pre-conditions of `map`. There are four conditions. The first three require that the parameters key, line and output not be null. These conditions are necessary in JML simply because the Hadoop framework is realized in Java, and the framework itself guarantees the conditions. They do not appear in our Coq model. The fourth condition is about the initial value of the virtual field, and it is also unrelated to the Coq model. Similarly, the pre-conditions of `reduce` are all unrelated to the Coq model. In summary, it is unnecessary to translate the pre-conditions into Coq.

However, the post-conditions should be translated into axioms in Coq. As for the `reduce` method, they are translated as in Table 3. Human intervention was needed here because Krakatoa does not know which variable of JML should correspond the return value of `mapper_ext` and `reduce_ext` functions in Coq.

For example, the first post-condition asserts that the list produced by the `reduce` method is not empty. Since the output of the `reduce` method corresponds to the value of the reducer function in the Coq model, we can naturally write an axiom which says that the value of `reducer` is not empty. Other post-conditions and those of the `map` method are translated similarly.

We anticipate that translation of post-conditions can be automatically done in general. We plan to write an automatic translator after examining a number of applications and gaining experiences in translation by hand.

Proof of theorems. The last step is to prove the properties that the `mapper_ext` and `reducer_ext` functions should satisfy. For this purpose, it is sufficient that the `mapper_ext` and `reducer_ext` functions here are the same as those defined in Section 2. In fact, we can prove the following theorems using the axioms obtained by translating the post-conditions.

```
Theorem mapper_coq: forall (u: unit)(s: string),
  mapper (u, s) = mappper_ext (u, s).

Theorem reducer_coq: forall (w: string) (ln: list nat),
  reducer (w, ln) = reducer_ext (w, ln).
```

Table 2. Pre- and post-conditions of `map` and `reduce`

	pre-condition	post-condition
map	– key != null – line != null – output != null – output.lst == nil()	– \forall String w; numAppear1(\old(line.value), w) == numAppear2(output.lst, w, 1) – \forall String w; \forall int i; numAppear2(output.lst, w, i) > 0 ==> i == 1
reduce	– key != null – values != null – output != null – output.lst == nil() – nonNullList(values.lst)	– output.lst != nil() – tail(output.lst) == nil() – headTLText(output.lst).value == key.value – headTLLw(output.lst).value == totalLw(\old(values.lst))

Table 3. Translation of post-conditions

JML post-condition	Coq axiom
`output.lst != nil()`	`Axiom wcReducerJML\_1: forall (k: string) (vs: list nat), reducer (k, vs) <> nil.`
`tail(output.lst) == nil()`	`Axiom wcReducerJML_2: forall (k: string) (vs: list nat), tail (reducer (k, vs)) = nil.`
`headTLText(output.lst).value == key.value`	`Axiom wcReducerJML_3: forall (k: string) (vs: list nat), fst (headKV3 (reducer (k, vs))) = k.`
`headTLLw(output.lst).value == totalLw(\old(values.lst))`	`Axiom wcReducerJML_4: forall (k: string) (vs: list nat), snd (headKV3 (reducer (k, vs))) = totalLw vs.`

5 Experiments

We compared the time consumption and outputs of the two approaches by executing word count solvers by both approaches. Time consumption is shown in Table 4. Both approaches gave the same outputs except the order of tuples.

Table 4. Execution time for both approaches. All experiments were conducted on four machines running CentOS 5.6 (x86_64) on Xeon 2.4GHz (2 cores) with 6GB Memory and 10GB Disk. Java environment was SUN JDK 6 Update 24. Hadoop was version 0.20.203.0. The Haskell enviroment consisted of GHC 6.12.3, Haskell Platform 2010.2.0.0. One machine was used as the master node, where the name node, the secondary name node and job tracker node were executed each assigned 1GB memory. The other machines were used as slave nodes. In each machine, we deployed a data node, a task tracker, and six children. We assigned 1GB memory for each data node and task tracker and 200 MB memory for each child.

input files	language	execution time
2 MB× 4	java	33s
2 MB× 4	streaming with haskell	1m35s
2 MB× 10	java	41s
2 MB× 10	streaming with haskell	3m47s

6 Related Work

Yang et al. [15] models the MapReduce framework using CSP. However, they do not provide a method for specifying or proving properties of applications using the MapReduce framework. Dörre, Apel and Lengauer [6] employs the type checker of Java 5 compiler in order to ensure that no type error occurs during execution of a MapReduce application. However, the types cannot express specifications involving values of the computation result. Lämmel [9] models the MapReduce framework as a higher-order function in Haskell. In his model, the MapReduce framework takes a mapper function and a reducer function as arguments. He also tries to capture the formal specification of the MapReduce framework in a equation between programs. However, he "contend[s] that the

formal property is (too) complicated," and he does not show how to use the formal property to show correctness of MapReduce applications.

Hübel [7] has implemented a variant of MapReduce in the strongly typed language Haskell. Although he does not show how to specify and prove properties of MapReduce applications, it is probable that our program extraction approach can be adapted to his framework so that we can produce verified application under his framework from a Coq proof script.

7 Conclusion and Future Work

We took two approaches to formally verify actual running code of a Hadoop MapReduce application. In the first approach, the mapper and reducer functions in Coq were transformed into Haskell programs and executed under Haskell_Hadoop and Hadoop Streaming. In the second approach, JML annotations of Java programs were verified with Krakatoa and translated into Coq axioms, which were used to prove Coq specifications. Both approaches are based on a Coq model of Hadoop MapReduce computation.

The Coq model was constructed in a modular fashion. We expressed a specification of a MapReduce application in terms of invariants among functions from key-value pairs to an application-specific type. A proof of a specification in the model can be constructed by providing a monoid structure, functions expressing the invariants, and lemmas related to those functions. As concrete applications, we verified WordCount and InvertedIndex and briefly described their proofs.

This research was conducted as a joint-work of a university team with a private company, which aims at applying formal verification techniques in actual, profitable software developments. The future work follows:

- We plan to verify more MapReduce applications, in particular, applications such as page ranking that require iteration of MapReduce computation.
- While verifying more applications, we will evaluate the feasibility of our modular proofs. In particular, we will investigate the range of MapReduce applications that can be specified in terms of invariants among functions.
- Translation from JML annotations to Coq axioms should be automated.
- Showing the correctness of inputSplitter and tokenizer is not a part of the verification of an MapReduce application but part of the verification of the Hadoop libraries. It is a challenging research issue to analyze and verify the Hadoop framework itself.

References

1. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
2. Chalin, P., Kiniry, J., Leavens, G., Poll, E.: Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)

3. Chrząszcz, J.: Implementing modules in the Coq system. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 270–286. Springer, Heidelberg (2003)
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51, 107–113 (2008)
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52, 365–473 (2005)
6. Dörre, J., Apel, S., Lengauer, C.: Static type checking of Hadoop MapReduce. In: MapReduce 2011. ACM, New York (to appear, 2011)
7. Hübel, T.: The Holumbus Framework. Master's thesis, Wedel University of Applied Sciences (2008)
8. Jimmy Lin, C.D.: Data-Intensive Text Processing with MapReduce. Morgan and Claypool (2010)
9. Lämmel, R.: Google's MapReduce programming model – revisited. Science of Computer Programming 70(1), 1–30 (2008)
10. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Muller, P., Kiniry, J., Chalin, P., Zimmerman, D.M.: JML reference manual (2011), http://www.eecs.ucf.edu/~leavens/JML/documentation.shtml
11. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. Journal of Logic and Algebraic Programming 58(1-2), 89–106 (2004)
12. Marlow, S.: Haskell 2010 language report (2010), http://www.haskell.org/onlinereport/haskell2010/
13. de Moura, L., Bjøner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Inc., Sebastopol (2009)
15. Yang, F., Su, W., Zhu, H., Li, Q.: Formalizing MapReduce with CSP. In: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2010, pp. 358–367. IEEE, Los Alamitos (2010)

ProMoVer: Modular Verification of Temporal Safety Properties*

Siavash Soleimanifard[1], Dilian Gurov[1], and Marieke Huisman[2]

[1] Royal Institute of Technology, Stockholm, Sweden
[2] University of Twente, Enschede, Netherlands

Abstract. This paper describes ProMoVer, a tool for fully automated procedure–modular verification of Java programs equipped with method–local and global assertions that specify safety properties of sequences of method invocations. Modularity at the procedure–level is a natural instantiation of the modular verification paradigm, where correctness of global properties is relativized on the local properties of the methods rather than on their implementations, and is based here on the construction of maximal models for a program model that abstracts away from program data. This approach allows global properties to be verified in the presence of code evolution, multiple method implementations (as arising from software product lines), or even unknown method implementations (as in mobile code for open platforms). ProMoVer automates a typical verification scenario for a previously developed tool set for compositional verification of control flow safety properties, and provides appropriate pre– and post–processing. Modularity is exploited by a mechanism for proof reuse that detects and minimizes the verification tasks resulting from changes in the code and the specifications. The verification task is relatively light–weight due to support for abstraction from private methods and automatic extraction of candidate specifications from method implementations. We evaluate the tool on a number of applications from the smart card domain.

1 Introduction

In modern computing systems, code changes frequently. Modules (or components) evolve rapidly or exist in multiple versions customized for various users, and in mobile contexts, a system may even automatically reconfigure itself. As a result, systems are no longer developed as monolithic applications; instead they are composed of ready–made off–the–shelf components, and each component may be dynamically replaced by a new one that provides improved or additional functionality. This static and dynamic *variability* makes it more important to provide formal correctness guarantees for the behaviour of such systems, but at the same time also more difficult. *Modularity* of verification is a key to providing such guarantees in the presence of variability.

* Soleimanifard's work is funded by the ContraST project of the Swedish Research Council VR, and Gurov's work by the EU FET project FP7–ICT–2009–3 HATS.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 366–381, 2011.

In modular verification, correctness of the software components is specified and verified independently (*locally*) for each module, while correctness of the whole system is specified through a *global* property, the correctness of which is verified relative to the local specifications rather than relative to the actual implementations of the modules. It is this relativization that enables verification of global properties in the presence of static and dynamic variability. In particular, it allows an independent evolution of the implementations of individual modules, only requiring the re–establishment of their local correctness.

Hoare logic provides a popular framework for modular specification and verification of software, where it is natural to take the individual procedures as modules, in order to achieve scalability, see *e.g.*, [18]. While Hoare logic allows the *local effect* of invoking a given procedure to be specified, temporal logic is better suited for capturing its *interaction with the environment*, such as the allowed sequences of procedure invocations. This paper shows that procedure–modular verification is also appropriate for safety temporal logic: for each procedure the local property specifies its legal call sequences, while the system's global property specifies the allowed interactions of the system as a whole. Thus, temporal specifications provide a meaningful abstraction for procedures.

To support our approach, we have developed a fully automated verification tool, ProMoVer, which can be tried via a web–based interface [20]. It takes as input a Java program annotated with global and method–local correctness assertions written in temporal logic and it automatically invokes a number of tools from CVPP, a previously developed tool set for compositional verification [13], to perform the individual local and global correctness checks. Essentially, ProMoVer is a wrapper that performs a standard verification scenario in the general tool set, to demonstrate that procedure–modular verification of temporal safety properties can be applied automatically. Importantly, ProMoVer only requires the public procedures to be annotated; the private ones are being considered merely as an implementation means. In addition, ProMoVer provides a facility to extract a method's legal call sequences by means of static analysis, given a concrete procedure implementation. A user thus does not have to write annotations explicitly; it suffices to inspect the extracted specifications and remove superfluous constraints that might hinder possible evolution of the code. Finally, ProMoVer also practically supports modularity by providing proof storage and reuse: only the properties that are affected by a change (either in implementation or in specification) are reverified, all other results are reused.

We show validity of the approach on some typical Java Card e-commerce applications. Such security–relevant applications are an important target for formal verification techniques. Here, we verify the absence of calls to non–atomic methods within transactions. Such properties, specifying legal call sequences for security–related methods, are an important class of platform–specific security properties. The ProMoVer web interface allows the user to verify such properties, for which a ready–made formalization is provided.

To allow efficient algorithmic modular verification, the tool set currently abstracts away from all data, thus considering safety properties of the control flow;

in particular, method calls in Java programs are over–approximated by non–deterministic choice on possible method implementations that the virtual call resolution might resolve to. This rather severe restriction on the program model is imposed by the maximal model construction that is the core of our modular verification technique (see [9] for a proof of soundness and completeness for this program model). Still, many useful properties can be expressed at this level of abstraction. These include platform–specific security properties as discussed above, and application–specific properties such as: (*i*) a method that changes sensitive data is only called from within a dedicated authentication method, i.e., unauthorized access is not possible; or (*ii*) in a voting system, candidate selection has to be finished, before the vote can be confirmed. Extending the technique with data, either over finite domains or over pointer structures, will allow for a wider range of properties and possible applications, but requires a non–trivial generalisation of the maximal model construction, and needs to be combined with abstraction techniques to control the complexity of verification and of model extraction from a program. We are currently investigating this.

Control flow safety properties can be expressed in various formalisms, *e.g.*, automata–based or process–algebraic notations, as well as in temporal logics such as LTL [22] and the safety fragment of the modal μ-calculus [15]. Internally, CVPP uses the latter, but ProMoVer allows the user to write the specifications in LTL, which is usually considered more intuitive. It is future work to extend ProMoVer also with other notations, in particular graphical ones.

ProMoVer currently handles procedure–modular verification of control–flow properties for sequential programs. The restriction to modularity at procedure level is not fundamental, and will be relaxed in future versions. As mentioned above, we are working on extending the method with data. The underlying theory for modelling multi–threaded programs has been developed earlier (see [12]), but the model checking problem is not decidable in general and has to be approximated suitably.

From a more practical point of view, the two main limitations are performance and the effort needed to write specifications. With respect to the first one, known theoretical bottlenecks are the maximal model construction and model checking of global properties (both are exponential in the size of the formula), as well as the efficient extraction of precise program models (in particular concerning virtual call resolution and exception propagation). The support for proof reuse is our main means of addressing these bottlenecks. As to the second limitation, to reduce the effort needed to write specifications, ProMoVer provides a library of common platform-specific global properties, and can extract specifications from a given implementation, as explained above.

The work in this paper is closely related to the development of CVPP [13]. As already pointed out, ProMoVer is essentially a wrapper that automates a typical verification scenario for CVPP, where modularity is applied at the procedure–level. In addition, ProMoVer provides support for proof reuse, and specification extraction, a collection of ready–formalised properties, and translates between the different intermediate formats and formalisms. Preliminary

results on an earlier version of ProMoVer were reported at a workshop [21]. The present paper extends and completes this work. In particular, we have added several facilities to improve the usability of the tool, in the form of automated support for proof reuse, specification extraction, and private method abstraction (see Section 4). Furthermore, we have adapted and extended significantly the experimental evaluation of the tool (see Section 5).

Related Work. A non–compositional verification method based on a program model closely related to ours is presented by Alur *et al.* [3]. It proposes a temporal logic CaRet for nested calls and returns (generalized to a logic for nested words in [1]) that can be used to specify regular properties of local paths within a procedure that skips over calls to other procedures. esp is another example of a successful system for non–compositional verification of temporal safety properties, applied to C programs [5]. It combines a number of scalable program analyses to achieve precise tracking (simulation) of a given property on multiple stateful values (such as file handles), identified through user–defined source code patterns. Maven is a modular verification tool addressing temporal properties of procedural languages, but in the context of aspects [7]. Recent work by Alur and Chauhuri proposes a unification of Hoare–style and Manna–Pnueli–style temporal reasoning for procedural programs, presenting proof rules for procedure–modular temporal reasoning [2].

Overview. The rest of this paper is organized as follows. Section 2 presents the use of ProMoVer from a user's point–of–view. Section 3 recapitulates the verification framework, describing the underlying program model and logic, and the compositional verification method based on constructing maximal models. Then, Section 4 describes the ProMoVer tool, while Section 5 describes three small but realistic case studies using the tool. Finally, the last section draws conclusions and suggests directions for future research.

2 ProMoVer: A User's View

We start by illustrating how ProMoVer is used on a small example. Both local method and global program properties are provided as assertions in the form of program annotations. We use a JML–like syntax for annotations (*cf.* [17]). ProMoVer is procedure–modular in the sense that correctness of the global program property is relativized on the local properties of the individual methods. Thus, the overall verification task divides into two independent subtasks:

(*i*) a check that each method implementation satisfies its local property, and
(*ii*) a check that the composition of local properties entails the global property.

Notice that the second subtask only relies on the local properties and does not require the implementations of the individual methods. Thus, changing a method implementation does not require the global property to be reverified, only the local property. If the second subtask fails, ProMoVer provides a counter example in the form of a program behaviour that violates the respective property.

```
// @global_ltl_prop: even -> X ((even && !entry) W odd)
public class EvenOdd {
   /** @local_interface: required odd
    *  @local_ltl_prop: G (X (!even || !entry) && (odd -> X G even)) */
   public boolean even(int n) {
      if (n == 0) return true; else return odd(n-1); }

   /** @local_interface: required even
    *  @local_ltl_prop: G (X (!odd || !entry) && (even -> X G odd)) */
   public boolean odd(int n) {
      if (n == 0) return false; else return even(n-1); }
}
```

Fig. 1. A simple annotated Java program

In addition to the properties, the technique also requires global and local *interfaces*. A global interface consists of a list of the methods *provided* (i.e., implemented) and *required* (i.e., used) by the program. The local interface of method m contains a list of the methods *required* by the method (as the provided method is obvious). ProMoVer can extract both global and local interfaces from method implementations.

Example 1. Consider the annotated Java program in Figure 1. It consists of two methods, `even` and `odd`. The program is annotated with a global control flow safety property, and every method is annotated with a local property and an interface specifying the required methods. As mentioned above, the interfaces can be extracted from the method implementations. The local method specifications also can be extracted by ProMoVer, see Section 4.

Here we give an intuitive description of the properties specified in the example; a formal definition of the temporal logic LTL is given below in Definition 4. The global property expresses that "in every program execution starting in method `even`, the first call is not to method `even` itself". The local property of method `even` expresses that "method `even` can only call method `odd`, and after returning from the call, no other method can be called". The local property of method `odd` is symmetric.

As explained above, the annotated program is correct if (*i*) methods `even` and `odd` meet their respective local properties, and (*ii*) the composition of local properties entails the global one. In fact, the annotated program is correct and our tool therefore returns an affirmative result.

Example 2. If we change the global property of the previous example to "in every program execution starting in method `even`, no call to method `odd` is made", the tool detects this and rechecks the global property for the already computed composition of local properties. The local properties do not have to be reverified. The verification of the global property fails. As a counter example, ProMoVer returns the following program execution that is allowed by the local properties, but violates the global one:

$$(\texttt{even}, \varepsilon) \xrightarrow{\text{even call odd}} (\texttt{odd}, \texttt{even}) \xrightarrow{\text{odd ret even}} (\texttt{even}, \varepsilon)$$

adapted for user understandability by replacing program points with the names of the methods they belong to (*cf.* Definition 3).

3 Framework for Modular Specification and Verification

Next, we briefly present the formal framework underlying the ProMoVer tool that supports this style of procedure–modular verification. It is heavily based on our earlier work on compositional verification [9,8].

3.1 Program Model and Logic

First, we formally define the program model and property specification logic.

Definition 1 (Model). *A* model *is a (Kripke) structure* $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$ *where* S *is a set of states,* L *a set of labels,* $\rightarrow \subseteq S \times L \times S$ *a labeled transition relation,* A *a set of atomic propositions, and* $\lambda : S \rightarrow \mathcal{P}(A)$ *a valuation, assigning to each state* s *the set of atomic propositions that hold in* s*. An* initialized model *is a pair* $(\mathcal{M}, E)$ *with* $\mathcal{M}$ *a model and* $E \subseteq S$ *a set of initial states.*

Our program model is based on the notion of *flow graph*, abstracting away from all data in the original program. It is essentially a collection of *method graphs*, one for each method of the program. Let *Meth* be a countably infinite set of methods names. A method graph is an instance of the general notion of initialized model.

Definition 2 (Method graph). *A* method graph *for method* $m \in Meth$ *over a set* $M \subseteq Meth$ *of method names is an initialized model* $(\mathcal{M}_m, E_m)$ *where* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ *is a finite model and* $E_m \subseteq V_m$ *is a non-empty set of* entry nodes *of* m*.* V_m *is the set of* control nodes *of* m*,* $L_m = M \cup \{\varepsilon\}$*,* $A_m = \{m, r\}$*, and* $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$ *so that* $m \in \lambda_m(v)$ *for all* $v \in V_m$ *(i.e., each node is tagged with its method name). The nodes* $v \in V_m$ *with* $r \in \lambda_m(v)$ *are* return points*.*

Notice that methods can have multiple entry points. Flow graphs that are extracted from program source have single entry points, but the maximal models that we generate for compositional verification may have several.

Every flow graph $\mathcal{G}$ is equipped with an *interface* $I = (I^+, I^-)$, denoted $\mathcal{G} : I$, where $I^+, I^- \subseteq Meth$ are the *provided* and externally *required* methods, respectively. These are needed to construct maximal flow graphs (see Section 3.2).

A flow graph is *closed* if its interface does not require any methods, and it is *open* otherwise. Flow graph *composition* is defined as the disjoint union $\uplus$ of their method graphs.

Example 3. Figure 2 shows the flow graph of the program from Figure 1. Its interface is $(\{\texttt{even}, \texttt{odd}\}, \emptyset)$, thus the flow graph is closed. It consists of two method graphs, for method `even` and method `odd`, respectively. Entry nodes are depicted as usual by incoming edges without source.

Flow graph *behavior* is also defined as an instance of an initialized model, induced through the flow graph structure. We use transition label τ for internal transfer of control, m_1 call m_2 for the invocation of method m_2 by method m_1 when method m_2 is provided by the program and m_1 call! m_2 when method m_2 is external, and m_2 ret m_1 respectively m_2 ret? m_1 for the corresponding return from the call.

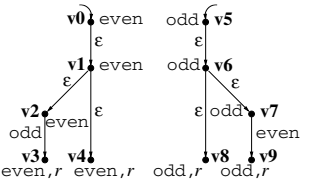


Fig. 2. Flow graph of EvenOdd

Definition 3 (Behavior). *Let $\mathcal{G} = (\mathcal{M}, E) : (I^+, I^-)$ be a flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The behaviour of $\mathcal{G}$ is defined as initialized model $b(\mathcal{G}) = (\mathcal{M}_b, E_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_b, A_b, \lambda_b)$, such that $S_b = (V \cup I^-) \times V^*$, i.e., states are pairs of control points v or required method names m, and stacks σ, $L_b = \{m_1\ k\ m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\}, m_1, m_2 \in I^+\} \cup \{m_1\ \mathsf{call!}\ m_2 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{m_2\ \mathsf{ret?}\ m_1 \mid m_1 \in I^+, m_2 \in I^-\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\lambda_b((m, \sigma)) = m$, and $\rightarrow_b \subseteq S_b \times L_b \times S_b$ is defined by the following rules:*

$$
\begin{aligned}
&[\text{transfer}]\ (v,\sigma) \xrightarrow{\tau} (v',\sigma) && \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r\\
&[\text{call}]\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma) && \text{if } m_1, m_2 \in I^+, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r,\\
&&& \quad v_2 \models m_2, v_2 \in E\\
&[\text{ret}]\ (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1,\sigma) && \text{if } m_1, m_2 \in I^+, v_2 \models m_2 \wedge r, v_1 \models m_1\\
&[\text{call!}]\ (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call!}\ m_2} (m_2, v_1' \cdot \sigma) && \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \xrightarrow{m_2}_{m_1} v_1', v_1 \models \neg r\\
&[\text{ret?}]\ (m_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret?}\ m_1} (v_1,\sigma) && \text{if } m_1 \in I^+, m_2 \in I^-, v_1 \models m_1
\end{aligned}
$$

The set of initial states is defined by $E_b = E \times \{\varepsilon\}$, where ε denotes the empty sequence over $V \cup I^-$.

Notice that return transitions always hand back control to the caller of the method. Calls to external methods are modeled with intermediate state, from which only an immediate return is possible. In this way possible callbacks from external methods are not captured in the behaviour. This simplification is justified, since we abstract away from data in the model and the behaviour is thus context–free, but has to be kept in mind when writing specifications; in particular one cannot specify that callbacks are not allowed.

Example 4. Consider the flow graph from Example 3. An example run through its (branching, infinite–state) behaviour, from an initial to a final state, is:

$$(v_0,\varepsilon) \xrightarrow{\tau} (v_1,\varepsilon) \xrightarrow{\tau} (v_2,\varepsilon) \xrightarrow{\mathsf{even\ call\ odd}} (v_5,v_3) \xrightarrow{\tau} (v_6,v_3) \xrightarrow{\tau} (v_8,v_3) \xrightarrow{\mathsf{odd\ ret\ even}} (v_3,\varepsilon)$$

Now, consider just the method graph of method `even` as an open flow graph, having interface $(\{\texttt{even}\}, \{\texttt{odd}\})$. The *local contribution* of method `even` to the above global behaviour is the following run:

$$(v_0,\varepsilon) \xrightarrow{\tau} (v_1,\varepsilon) \xrightarrow{\tau} (v_2,\varepsilon) \xrightarrow{\mathsf{even\ call!\ odd}} (\texttt{odd}, v_3) \xrightarrow{\mathsf{odd\ ret?\ even}} (v_3,\varepsilon)$$

Pushdown systems. (PDS) are an alternative way to express flow graph behaviour. We exploit this by using PDS model checking, concretely the tool MOPED [14], for verifying program behaviour against temporal formulas.

As mentioned above, safety properties can be expressed in many different formalisms. In this paper, we use *safety LTL* which consists of the safety–fragment of *Linear Temporal Logic* (LTL), using the weak until–operator. Internally, however, the whole machinery is based on the safety fragment of the modal μ-calculus. Safety LTL is somewhat less expressive than the latter and can be uniformly encoded in it. This translation is implemented as part of PROMOVER. In our LTL formulas, we use an additional atomic proposition `entry` that holds for entry nodes. It is removed by the translation into the modal μ-calculus.

Definition 4 (Safety LTL). *Let $p \in A_b \cup \{\texttt{entry}\}$ and $m \in M$. The formulae of* Safety LTL *are inductively defined by:*

$$\phi ::= p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \texttt{X}\,\phi \mid \texttt{G}\,\phi \mid \phi_1 \,\texttt{W}\, \phi_2$$

Satisfaction on states $(\mathcal{M}_b, s) \models \phi$ for LTL is defined in the standard fashion [22]: formula $\texttt{X}\,\phi$ holds of state s in model $\mathcal{M}_b$ if ϕ holds in the next state of every run starting in s; $\texttt{G}\,\phi$ holds if for every run starting in s, ϕ holds in all states of the run; and $\phi \,\texttt{W}\, \psi$ holds in s if for every run starting in s, either ϕ holds in all states of the run, or ψ holds in some state and ϕ holds in all previous states.

Example 5. Consider the global property of class `EvenOdd` in Figure 1 (where `&&` is ASCII notation for $\wedge$) and its intuitive meaning in Example 1. Flow graph extraction and construction ensures that entry nodes are only accessible via calls; hence, if control starts and remains in method `even`, execution can be at an entry node only as the result of a self–call. The formula thus states that "if program execution starts in method `even`, method `even` is not called until method `odd` is reached", which coincides with the interpretation given in Example 1.

3.2 Compositional Verification

Our method for *algorithmic compositional verification* is based on the construction of maximal flow graphs from component properties. For a given property ψ and interface I, consider the set of all flow graphs with interface I satisfying ψ. A *maximal flow graph* for ψ and I, denoted $\mathcal{M}ax(\psi, I)$, satisfies exactly those properties that hold for all members of the set. Thus, the maximal flow graph can be used as a representative of the set for the purpose of property verification. For details the reader is referred to [9].

For a system with k components, our principle of compositional verification based on maximal flow graphs can be presented as a proof rule with $k + 1$ premises, that states that the composition of components $\mathcal{G}_1 : I_1, ..., \mathcal{G}_k : I_k$ satisfies a global property ϕ if there are local properties ψ_i such that (i) each component $\mathcal{G}_i$ satisfies its local property ψ_i, and (ii) the composition of the k maximal flow graphs $\mathcal{M}ax(\psi_I, I_i)$ satisfies ϕ.

$$\frac{\mathcal{G}_1 \models \psi_1 \;\; \cdots \;\; \mathcal{G}_k \models \psi_k \qquad \biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi}{\biguplus_{i=1,\ldots,k} \mathcal{G}_i \models \phi}$$

As mentioned above, in the context of ProMoVer, we consider individual program methods as components. If we instantiate the above compositional verification principle to procedure–modular verification, we obtain the verification tasks stated informally in Section 2 (where M is the set of program methods, with $k = |M|$, and ψ_i and $\mathcal{C}_i$ are the specification and the implementation of method m_i, respectively):

(i) **Checking $\mathcal{C}_i \models \psi_i$ for $i = 1, ..., k$:** For each method $m_i \in M$, (a) extract the method flow graph $\mathcal{G}_i$ from $\mathcal{C}_i$, and (b) model check $\mathcal{G}_i$ against ψ_i. For the latter, we exploit the fact that flow graphs are *Kripke structures*, and apply standard finite–state model checking.

(ii) **Checking $\biguplus_{i=1,\ldots,k} \mathcal{M}ax(\psi_i, I_i) \models \phi$:** ($a$) Construct maximal flow graphs $\mathcal{M}ax(\psi_i, I_i)$ for all method specifications ψ_i and interfaces I_i, then (b) compose the graphs, resulting in flow graph $\mathcal{G}_{\mathcal{M}ax}$, and finally ($c$) model check $\mathcal{G}_{\mathcal{M}ax}$ against global property ϕ. For the latter, represent the behaviour of $\mathcal{G}_{\mathcal{M}ax}$ as a PDS and use a standard PDS model checker.

Example 6. Consider again the annotated Java program from Example 1. ProMoVer first extracts the method flow graphs of methods `even` and `odd`, denoted $\mathcal{G}_{even}$ and $\mathcal{G}_{odd}$, respectively. Next, ProMoVer checks $\mathcal{G}_{even} \models \psi_{even}$ and $\mathcal{G}_{odd} \models \psi_{odd}$ by standard finite state model checking. Independently, it constructs the maximal flow graphs of methods `even` and `odd`, denoted $\mathcal{M}ax(\psi_{even}, I_{even})$ and $\mathcal{M}ax(\psi_{odd}, I_{odd})$, respectively, and composes the graphs to obtain $\mathcal{G}_{\mathcal{M}ax} = \mathcal{M}ax(\psi_{even}, I_{even}) \uplus \mathcal{M}ax(\psi_{odd}, I_{odd})$. Finally, ProMoVer translates $\mathcal{G}_{\mathcal{M}ax}$ to a PDS and model checks the latter against the global property.

4 The ProMoVer Tool

Next we describe the internals of ProMoVer. As mentioned above, ProMoVer essentially is a wrapper for cvpp [13], with extra features such as specification extraction, private method abstraction, a property specification library and support for proof reuse. All features are implemented in Python. ProMoVer can be tested via a web interface [20].

CVPP Wrapper. Figure 3 shows schematically how ProMoVer combines the individual cvpp tools. An annotated Java program, as exemplified in Section 2, is given as input. The *pre–processor* parses the annotations, using the Java Doclet API [6], and then passes properties and interfaces on to the different cvpp tools.

Task (i) first invokes the Analyzer to extract the method graphs of the program. This builds on Sawja [11] to extract flow graphs from Java bytecode. Then

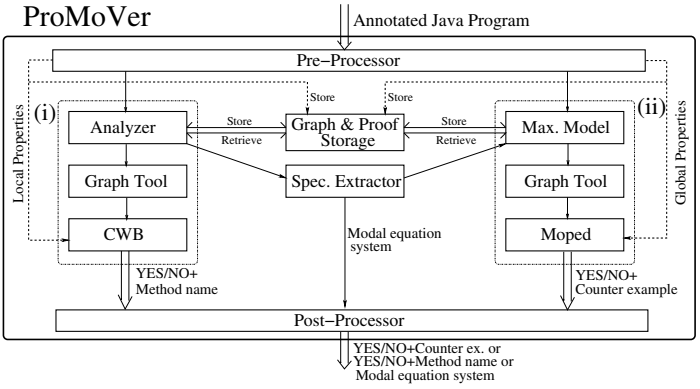


Fig. 3. Overview of ProMoVer and its underlying tool set

the Graph Tool is used. This implements several algorithms on flow graphs, including flow graph composition $\uplus$ and translations of flow graphs into different formats. Here the Graph Tool is used to translate the flow graph of each method into a CCS model. These are then model checked against the respective local method specifications using the *Concurrency Workbench* (cwb) [4].

Task (*ii*) first constructs a maximal flow graph for every method using the Maximal Model Tool. Then the Graph Tool composes the generated flow graphs and converts the result into a PDS. Finally Moped [14] is used to model check the PDS against the global property.

The *post–processor* collects all model checking results and converts these into a user–understandable format. It only returns a positive result if all collected model checking tasks succeed. If one of the local model checking tasks fails, the name of the method that violates its specification is returned. If the global model checking task fails, the counter example provided by Moped, transformed into a program execution, is returned.

Specification Extraction. To reduce the effort needed to write specifications, ProMoVer provides support to extract a specification from a given method implementation, resulting in the (over–approximated) order of method invocations for this method. The user might then want to remove some superfluous dependencies, in order not to be overly restrictive on possible evolution of the code. ProMoVer extracts specifications in the form of modal equation systems (as defined by Larsen [16]). These are equivalent to formulae in modal μ-calculus with boxes and greatest fixed points only, and have the advantage that in cvpp they can serve directly as input for the construction of maximal flow graphs. It is future work to also extract to other specification languages, such as LTL.

Consider again Figure 1. Specification extraction for method `even` results in (where `eps` is ASCII notation for ε, and `ff` denotes *false*):

```
@local_eq_prop: (X0){ X0  =  [odd](X1) /\ [even]ff /\  [eps]X0;
                      X1  =  [odd] ff  /\ [even]ff /\  [eps]X1;}
```

This specifies that method `odd` may be called at most once: initially `X0` holds, and method `odd` may be called or an internal step (labelled `eps`) may be made. After calling `odd`, `X1` should hold and only internal steps are allowed.

As a more involved example, consider method `m` and its extracted specification:

```
@local_eq_prop:
(X0){ X0  =  [m4]ff /\ [m1](X1) /\ [m3]ff /\ [m2]ff /\ [m]ff /\  [eps]X0;
      X1  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2](X2) /\ [m]ff /\  [eps]X1;
      X2  =  [m4](X3) /\ [m1]ff /\ [m3](X4) /\ [m2]ff /\ [m]ff /\ [eps]X2;
      X3  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]ff /\ [m]ff /\  [eps]X3;
      X4  =  [m4]ff /\ [m1]ff /\ [m3]ff /\ [m2]ff /\ [m]ff /\  [eps]X4; }
public void m() { int i = m1(); int j = m2();
                  if (i < j) {m3(); } else { m4(); } }
```

The formula captures that first only `m1` can be called, then only `m2`, and then either `m3` or `m4`, and no further calls can be made. Actually, the order of invoking `m1` and `m2` is immaterial for this program, so a designer may choose to change the equations defining `X0` and `X1` to allow the two methods to be called in any order (whereas the defining equations for `X2` to `X4` remain unchanged):

```
X0  =  [m4]ff /\ [m1](X10) /\ [m3]ff /\ [m2](X11) /\ [m]ff /\ [eps]X0;
X10 =  [m4]ff /\ [m1]ff    /\ [m3]ff /\ [m2](X2)  /\ [m]ff /\ [eps]X10;
X11 =  [m4]ff /\ [m1](X2)  /\ [m3]ff /\ [m2]ff    /\ [m]ff /\ [eps]X11;
```

Private Method Abstraction. Since private methods are used as means of implementation for public methods, at the flow graph level, all calls to private methods can be inlined into the flow graph of the public methods. The resulting method flow graphs thus only describe the public behaviour, and users only have to specify the public methods. For details the reader is referred to [9].

Property Specification Library. ProMoVer's web interface provides a collection of pre–formalised global properties. These describe platform–specific security properties, restricting calls to API methods. Currently, the library contains several Java Card and voting system properties.

Proof Storage and Reuse. All extracted method flow graphs and constructed maximal flow graphs are stored when a program is verified by ProMoVer. If later the implementation of method m changes, a new method flow graph is extracted and checked against m's local specification. If m's local specification ϕ_m changes, the existing flow graph of method m is model checked against ϕ_m. In addition a new maximal flow graph for m is constructed from ϕ_m. This is composed with the other maximal flow graphs (recovered from storage), and the composed flow graph is model checked against the global property.

5 Experimental Results with ProMoVer

We use ProMoVer to verify a standard control flow safety property on a number of Java Card applications. Java Card is one of the leading interoperable platforms for smart cards. Many smart card applications are security–critical.

Table 1. Applications details

Application	#LoC	#Methods (Public)	#Calls (Relevant)
`AccountAccessor`	190	9 (7)	38 (4)
`TransitApplet`	918	18 (5)	106 (5)
`JavaPurse`	884	19 (9)	190 (25)

As mentioned above, for platforms such as Java Card, collections of control flow safety properties exist that programs should adhere to in order to provide minimal security requirements. We focus on such a property of the Java Card transaction mechanism. This mechanism ensures that data remains consistent upon power loss. Safe use of it demands that certain methods are not called within a transaction. We show how this global safety property can be expressed in our setting, and be verified with ProMoVer for several applications, where we apply specification extraction to annotate the public methods of the applications.

The Java Card Transaction Mechanism. Smart cards have two types of writable memory, *persistent memory* (EEPROM or Flash) and *transient memory* (RAM). Transient memory needs constant power supply to store information, while persistent memory can store data without power. Smart cards do not have their own power supply; they depend on the external source that comes from the card reader device. Therefore, a problem known as *card tear* may occur: a power loss when the card is suddenly disconnected from the card reader. If a card tear occurs in the middle of updating data from transient to persistent memory, the data stored in transient memory is lost and may cause the smart card to be in an inconsistent state.

To prevent this, the *transaction mechanism* is provided. It can be used to ensure that several updates are executed as a single *atomic* operation, *i.e.*, either all updates are performed or none. The mechanism is provided through methods `beginTransaction` for beginning a transaction, `commitTransaction` for ending a transaction with performed updates, and `abortTransaction` for ending a transaction with discarded updates [10] – all declared in class `JCSystem` of the Java Card API.

However, the Java Card API also contains some *non–atomic* methods that are better not used when a transaction is in progress. Notably, the class `javacard.framework.Util` that provides functionality to store and update byte arrays, contains methods `arrayCopyNonAtomic` and `arrayFillNonAtomic`. Typical Java Card programming standards, such as the Global Platform specification, state that these methods may not be used within a transaction. We use ProMoVer to verify that applications comply with this *Safe Transaction Policy*.

The Applications. For this experiment we use several public examples of Java Card applications. All are realistic e-commerce applications developed by Sun Microsystems to demonstrate the use of the Java Card environment for developing e-commerce applications. `AccountAccessor` is an application to keep track of account information. It is to be used by a wireless device connected via a

Table 2. Verification Results

Application	PPT	GE	#NEF	LMC	MFC	#NMF	GMC	TT
`AccountAccessor`	1.4	3.8	435	0.5	0.7	20	0.9	8.7
`TransitApplet`	1.4	4.7	897	0.5	0.9	30	0.9	13.2
`JavaPurse`	1.5	6.5	1543	0.5	13.0	48	1.1	22.5

network service. It contains methods to look up and to modify the account balance. `TransitApplet` implements the on-card part of a system that connects to an authenticated terminal and provides account information and operations to modify the account balance. `JavaPurse` is a smart card electronic purse application providing secure money transfers. It contains a balance record denoting the user's current and maximum credits, and methods to initialize, perform and complete a secure transaction. Further, it also contains methods to update information related to a loyalty program, and to validate and update the values of transactions, balance and PIN code.

Table 1 shows information about the size, number of methods (total and public), and number of method invocations (total and relevant for the global property) of these applications.

Specification of Safe Transaction Policy. As discussed above, we want to ensure formally that the non-atomic methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not invoked within a transaction. Hence, applications have to adhere to the following global control flow safety property:

> In every program execution, after a transaction begins, methods `arrayCopyNonAtomic` and `arrayFillNonAtomic` are not called until the transaction ends.

This safety property can be expressed formally with the following LTL formula:

$$\texttt{G}\,(\texttt{beginTransaction} \rightarrow ((\neg\texttt{arrayCopyNonAtomic} \wedge \neg\texttt{arrayFillNonAtomic})\,\texttt{W}\,\texttt{commitTransaction}))$$

Extracting Local Method Specifications. The specification extractor is used to obtain local specifications for every public method. Basically, these describe the order of method invocations. We inspected those for immaterial orderings, and translated the adjusted representations into safety LTL. The intention is that local method specifications capture the allowed sequences of method calls made from within the specified method, but in an abstract way, allowing for possible evolution of the method implementations.

Verification Results. After annotating the applications, they are passed to ProMoVer. The tool extracts the flow graph of the applications, and partitions them into the individual method graphs to verify adherence to the local properties. Further, for each local property a maximal flow graph is constructed, and

Table 3. Proof Reuse Results

	Code Change		Local Specification Change		
Application	New TT	% TT	MFC	New TT	% TT
`AccountAccessor`	6.0	68	0.1	4.6	52
`TransitApplet`	7.2	54	0.1	5.0	37
`JavaPurse`	9.0	40	0.1	5.4	24

their composition is verified *w.r.t.* the global property above. The statistics for these verifications are given summarized in Table 2. The table shows: the time spent by the pre–processor (PPT) and the graph extractor (GE), the number of nodes in the extracted flow graphs (#NEF), the time spent for local model checking (LMC) and for constructing maximal flow graphs (MFC), the number of nodes in the maximal flow graph composition (#NMF), the time spent for global model checking (GMC), and the total time spent for the whole verification task including conversions between formats and post–processing (TT). All times are in seconds, and were obtained on a SUN SPARC machine.

We also experimentally evaluated the advantages of exploiting the proof storage and reuse mechanism. After the first verification, when method and maximal flow graphs are stored, for each application, we once changed the source code and once the local specification of a public method, and used ProMoVer to reverify the application. The result of proof reuse are shown in Table 3. The numbers show that proof reuse can reduce significantly the verification time for larger applications.

6 Conclusion

This paper describes ProMoVer, a tool that supports automatic procedure–modular verification of control flow safety properties of sequences of method invocations. ProMoVer takes as input a Java program annotated with temporal correctness assertions. It essentially implements a particular verification scenario for the CVPP tool set that supports compositional verification of programs with procedures [9].

Modularity is understood here as the relativization of global program correctness properties on the correctness of its components, and is seen as the key to program verification in the presence of static and or dynamic variability due to code evolution, code customization for many users, or as yet unknown or unavailable code such as mobile code. We illustrate two important points: (*i*) temporal safety properties provide a meaningful abstraction for individual methods; and (*ii*) procedure–modular verification of temporal safety properties can be performed automatically. Moreover, ProMoVer implements a mechanism for proof storage and reuse, so that only relevant parts have to be reverified after a system change. This makes the verification method advocated by ProMoVer suitable to be used in a context where systems evolve frequently, as is the case *e.g.*, for software product lines or mobile code. The modularity of the verification allows

an independent evolution of the implementations of the individual methods, only requiring the re–establishment of their local correctness.

We believe that writing properties at the procedure–level is intuitive for a programmer. Still, to decrease the effort of annotating programs, we provide support for specification extraction in the case of post–hoc specification of already implemented methods, an inlining–based private method abstraction that requires only public methods to be specified, and a library of standard global safety properties.

Experiments with realistic Java Card applications show that useful safety properties of such programs can be conveniently expressed in a light–weight notation and verified automatically with ProMoVer.

Still, some issues remain to be resolved in order to increase the utility of ProMoVer. Both for pre– and post–hoc method specification, notations based on automata or process algebra may prove more convenient than LTL, and may also allow more efficient maximal flow graph construction. Ultimately, our goal is that all specifications (local and global) can be written in various temporal logics and notations, or to use patterns to abbreviate common specification idioms. The tool set will provide translations into the underlying uniform logic, which is currently the safety fragment of the modal μ-calculus. However, because of limitations on the currently available PDS model checkers, global properties have at present to be written in LTL.

Many important safety properties require program *data* to be taken into account. As a first step towards handling data, work has begun on extending our verification framework and tool set to Boolean programs. We are also currently investigating how to generalize our method for the program model of Rot *et al.* that models object references in the presence of unbounded object creation [19].

Finally, to investigate the *scalability* of the approach, we plan to perform a significantly larger case study.

Acknowledgments. We are indebted to Wojciech Mostowski and Erik Poll for their help in finding a suitable case study, to Afshin Amighi and Pedro de Carvalho Gomes for helping with the implementation of cvpp and ProMoVer, and to Stefan Schwoon for adapting the input language of Moped to our needs.

References

1. Alur, R., Arenas, M., Barcelo, P., Etessami, K., Immerman, N., Libkin, L.: First-order and temporal logics for nested words. In: Logic in Computer Science (LICS 2007), pp. 151–160. IEEE Computer Society, Washington, DC, USA (2007)
2. Alur, R., Chaudhuri, S.: Temporal reasoning for procedural programs. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 45–60. Springer, Heidelberg (2010)
3. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 467–481. Springer, Heidelberg (2004)

4. Cleaveland, R., Parrow, J., Steffen, B.: A semantics based verification tool for finite state systems. In: International Symposium on Protocol Specification, Testing and Verification, pp. 287–302. North-Holland Publishing Co., Amsterdam (1990)
5. Das, M., Lerner, S., Seigle, M.: ESP: Path–sensitive program verification in polynomial time. In: Programming Language Design and Implementation (PLDI 2002), pp. 57–68. ACM, New York (2002)
6. Doclet overview, http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html
7. Goldman, M., Katz, S.: MAVEN: Modular aspect verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 308–322. Springer, Heidelberg (2007)
8. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 136–150. Springer, Heidelberg (2009)
9. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)
10. Hubbers, E., Poll, E.: Transactions and non-atomic API methods in Java Card: specification ambiguity and strange implementation behaviours. Technical Report NIII-R0438, Radboud University Nijmegen (2004)
11. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static Analysis Workshop for Java. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 92–106. Springer, Heidelberg (2011)
12. Huisman, M., Aktug, I., Gurov, D.: Program models for compositional verification. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 147–166. Springer, Heidelberg (2008)
13. Huisman, M., Gurov, D.: CVPP: A tool set for compositional verification of control–flow safety properties. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 107–121. Springer, Heidelberg (2011)
14. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: Moped - a model-checker for pushdown systems, http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/
15. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
16. Larsen, K.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990)
17. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual, Department of Computer Science, Iowa State University (February 2007), http://www.jmlspecs.org
18. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
19. Rot, J., de Boer, F., Bonsangue, M.: A pushdown system representation for unbounded object creation. In: Informal pre-proceedings of Formal Verification of Object–Oriented Software (FoVeOOS 2010) (2010)
20. Soleimanifard, S., Gurov, D., Huisman, M.: ProMoVer web interface, http://www.csc.kth.se/~siavashs/ProMoVer
21. Soleimanifard, S., Gurov, D., Huisman, M.: Procedure–modular verification of control flow safety properties. In: Workshop on Formal Techniques for Java Programs, FTfJP 2010 (2010)
22. Stirling, C.: Modal and Temporal Logics of Processes. Springer, Heidelberg (2001)

Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques

Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer

Chair of Software Engineering, ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. With formal techniques becoming more and more powerful, the next big challenge is making software verification practical and usable. The Eve verification environment contributes to this goal by seamlessly integrating a static prover and an automatic testing tool into a development environment. The paper discusses the general principles behind the integration of heterogeneous verification tools; the peculiar challenges involved in combining static proofs and dynamic testing techniques; and how the combination, implemented in Eve through a blackboard architecture, can improve the user experience with little overhead over usual development practices. Eve is freely available for download.

1 Verification as a Matter of Course

Even long-standing skeptics must acknowledge the substantial progress of formal methods in the last decades. Established verification techniques, such as those based on axiomatic semantics or abstract interpretation, have matured from the status of merely interesting scientific ideas to being applicable in practice to realistic programs and systems. Novel approaches have extended their domain of applicability beyond their original scope, providing new angles from which to attack the hardest verification challenges; for example, model checking techniques, initially confined to digital hardware verification, are now applied to software or real-time systems. Other techniques, such as testing, have long been part of the standard development process, but only recently have they become first-class citizens of the verification realm, evolving in the case of random-based testing into rigorous, formal, and automatable approaches. Verification requires accurate specifications, and progress in this area has been no less conspicuous, with the development of understandable notations, such as those based on Design by Contract, which integrate seamlessly with the programming language and are amenable to static as well as dynamic analysis techniques. Finally, tool support has tremendously improved in terms of both reliability and performance, as a result of cutting-edge engineering of every component in the verification tool-chain as well as the increased availability of computing power.

With the consolidation of these outstanding achievements [14], the new frontier is to make verification really usable by practitioners [28]: the quest for high reliability to become a standard part of the software development process—"verification as a matter of course". The present paper is a step towards this ambitious goal with two contributions, one general and one specific.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 382–398, 2011.

The general contribution is a development environment that seamlessly integrates formal verification with the standard tools offered by programming environments for object-oriented development (editor, compiler, debugger, ...). The integrated environment is called Eve, built on top of EiffelStudio—the main IDE for Eiffel developers. Section 6 describes the engineering of Eve, showing how it takes into account several of the heterogeneous concerns originating from the goal of improving the usability of formal verification, such as user interaction and management of computational resources.

The implementation of Eve, freely available for download [11], continues to evolve as a result of ongoing efforts to integrate more verification techniques and new verification tools. The currently available implementation, illustrated through an example session in Section 2, focuses on the integration of two well-known techniques: static verification based on Hoare-style proofs, currently implemented in Eve through the AutoProof tool [27,20], and dynamic analysis based on random testing, implemented through AutoTest [19]. Section 4 describes these tools. After a review of the state of the art in Section 3, Sections 5 and 7 illustrate the specific contribution of the paper by discussing the challenges of integrating two very different verification techniques, tests and proofs, and how Eve combines them to improve each one's effectiveness and usability. Section 9 concludes with an analysis of limitations and our current work to overcome them.

2 An Example Session with Eve

Consider the perspective of a user—henceforth called Adam—who is using Eve to develop a collection of data structure implementations. Table 1 shows portions of Adam's code; the code shown is simplified for presentation purposes, but it reflects real features found in versions of EiffelBase, a standard library used in most Eiffel programs.

The ancestor class *COLLECTION* models generic containers with a well-defined interface including, in addition to other features not shown, routines (methods) *extend* that adds its argument to the collection and *is_equal* which tests for object equality. *extend* is annotated with a precondition (**require**) and postcondition (**ensure**) which refer to other features of the class (such as *has*) not shown. *extend* is *deferred* (abstract) as it lacks an implementation; *is_equal*'s body, instead, calls a pre-compiled implementation written in C through the **external** keyword. This encapsulation mechanism prevents correctness proofs of the routine's implementation (whose source is not accessible); in addition, *COLLECTION* cannot be instantiated and tested because it includes deferred routines. This seems an unfortunate situation for verification, but verification with Eve becomes effective in the two descendants of *COLLECTION* shown in Table 1: *ARRAY* and *ARRAYED_LIST*.

Class *ARRAY* redefines the attribute *extendible* to **False** because an array is a container of statically-defined size and cannot accommodate new elements *ad lib*. Correspondingly, the precondition of the inherited feature *extend* becomes unsatisfiable in *ARRAY*. This way of "deactivating" a routine is inconvenient for automatic testing tools such as AutoTest, which tries, in a vain effort, to generate instances of *ARRAY* where the precondition of *extend* holds in order to test it. AutoProof, the static proof component of

Table 1. Classes *CONTAINER*, *ARRAY*, and *ARRAYED_LIST*

```
1  deferred class COLLECTION [G]
2    ...
3
4    extendible: BOOLEAN
5
6    extend (v: G)
7      -- Ensure that structure contains 'v'.
8    require
9      extendible
10     v ≠ Void
11   deferred
12   ensure has (v) end
13
14   has (v: G): BOOLEAN
15     -- Does structure contain 'v'?
16   deferred
17   end
18
19   is_equal (other: COLLECTION [G]): BOOLEAN
20     -- Is 'other' attached to an object equal to '
           Current'?
21   require other ≠ Void
22   external built_in
23   ensure Result = other.is_equal (Current) end
24
25 end -- CONTAINER
26
27 class ARRAY [G]
28   inherit COLLECTION [G]
29           redefine extendible end
30
31   extendible: BOOLEAN = False
32
33   ...
34 end -- ARRAY
```

```
35 class ARRAYED_LIST [G]
36   inherit ARRAY [G]
37           redefine extendible end
38
39   extendible: BOOLEAN = True
40
41   extend (v: G) do ... end
42
43   make_default (n: INTEGER)
44     -- Allocate list with 'n' slots items
45     -- and fill it with default values.
46   require n ≥ 0
47   local l_v: G
48   do
49     Precursor (n)
50     across [1..n] as i loop extend (l_v) end
51   end
52
53   remove_left_cursor (c: CURSOR)
54     -- Remove item to left of 'c' position.
55   require
56     not is_empty
57     c ≠ Void and valid (c)
58     not c.before and not c.is_first
59   do
60     remove (c.index - 1)
61   ensure
62     count = old count - 1
63     c.index = old c.index - 1
64   end
65 end -- ARRAYED_LIST
```

Eve, comes to the rescue in this case: it easily figures out that the precondition of *extend* is unsatisfiable in *ARRAY* (line 10 in Table 1), and hence that *extend* is trivially correct and requires no further analysis. Adam checks that *ARRAY.extend* receives a green light and requires no further attention (Figure 1).

Class *ARRAYED_LIST* switches *extendible* to **True** and provides a working implementation of *extend* available to clients. When Eve tries to test the class, it quickly discovers a fault in the creation procedure (constructor) *make_default*: after the instruction **Precursor** (*n*) calls the creation procedure in the ancestor of *ARRAY*, the loop (**across**...**loop**...**end**) tries to call *extend* with the local *l_v* as argument; this violates *extend*'s precondition clause *v* ≠**Void** because *l_v* is not initialized and hence equals the default value **Void** (*null* in Java or C). Adam sees there is something wrong in Eve's report (Figure 1); he expands the description of the error and understands how to fix the bug by adding an instruction **create** *l_v* before the loop on line 47.

While Adam is busy fixing the error, testing cannot proceed on the same class. Even if the creation procedure were correct, routine *remove_left_cursor* would remain arduous for automated testing techniques because its precondition is relatively complex; a random-based approach to the generation of test cases requires specialized techniques and a long running time to select objects satisfying the clauses in lines 53–55 [30]. Eve circumvents these limitations by running a static proof, which analyzes individual

Fig. 1. Example report of Eve, showing scores of classes and routines. The third column displays the lowest negative score among the routines of each class.

routines and does not need a correct creation procedure. The proof succeeds in establishing that the invocation of *remove* (line 57) is correct and ensures the postcondition of *remove_left_cursor*: the routine is correct and no testing is needed (Figure 1).

Later, as soon as the constructor of *ARRAYED_LIST* is fixed, Eve continues its work and exhaustively tests the implementation of *is_equal* finding no postcondition violations. This is not as good as a correctness proof, but it comforts Adam's confidence in the reliability of *is_equal*, and it is the best result possible for a routine whose implementation can be analyzed only as black-box.

Although it only uses some of Eve's features, this scenario illustrates how Eve can help develop correct applications with little overhead over standard practices:

- Eve is completely automatic and integrated in a full-fledged IDE.
- It supports verification of functional correctness specifications embedded as contracts (pre and postconditions, class invariants, intermediate assertions).
- It transparently manages different verification engines to complement their strengths, supports the full programming language Eiffel, and provides fast feedback to users.
- It only displays such feedback when needed, to encourage focus on the most egregious errors, and to increase the users' confidence in the correctness of an implementation based on the available evidence.

3 Related Work

The following sections explains the Eve machinery that makes usage scenarios such as the above possible. To set these solutions in context, we first examine briefly a few state-of-the-art tools for static and dynamic verification (proofs and tests), with a summary of their distinctive features and a summary of the relatively few approaches that combine both techniques. A broader review of formal techniques is available elsewhere [14,28].

Static Verification. Projects such as ESC/Java [12] and Spec# [2] have made Hoare-style correctness proofs more practical and automatic, at least for simple programs. The Spec# language extends C# with preconditions, postconditions, object invariants, and non-null types; the Spec# environment verifies Spec# programs by translating them into Boogie, also developed within the Spec# project. The success of this approach has shown the importance of using an intermediate language for verification. Spec# works

on interesting examples; however, it is still not applicable to every feature of C# (for example, exceptions and function objects). A design choice that distinguishes Spec# from AutoProof for Eiffel is the approach to deal with some delicate issues, namely the framing problem and managing class invariants. Spec# introduces specialized annotations, which make proofs easier but at the price of a significant additional annotational burden for developers. AutoProof, on the contrary, does not introduce *ad hoc* annotations and correspondingly may fail to verify programs where Spec# is successful. Some of these limitations are mitigated in Eve by supplementing AutoProof with testing.

Separation logic is an extension of Hoare logic designed to handle frame properties; verification environments based on separation logic (e.g., jStar [8] for Java and VeryFast [16] for C and Java) can verify sophisticated features such as usages of the visitor, observer, and factory patterns. Writing separation logic annotations requires considerably more expertise than using contracts embedded in the programming language; this makes separation-logic tools more challenging to use by practitioners.

Other static verification techniques, such as software model-checking [3] and abstract interpretation [6], approximate the semantics of programming languages to make their analysis scalable and to require little annotations. These techniques are currently unsupported in Eve, but they may become as part of future work.

Dynamic Verification. Only recently have dynamic techniques, such as testing, become applicable fully automatically to large programs (e.g., [13,17,4]). In this line of work, DART [13] introduced the concept of dynamic symbolic execution, a combination of dynamic verification with lightweight static techniques. CUTE [23] and EXE [4] follow similar approaches but they are applicable to more complex features (such as pointers and complex data structures) and scale massively. The main high-level difference of AutoTest is that it relies on contracts to verify functional properties; the aforementioned testing tools, instead, work on languages without contracts and therefore are limited in the kinds of errors that they can detect.

In recent years, dynamic techniques have extended their domain of applicability to problems such as contract inference [9,30] and specification mining [1,7] which have traditionally been approachable only by static means. Future versions of Eve will integrate dynamic contract inference as implemented in our AutoInfer tool (sketched in Section 6).

Combined Static/Dynamic Verification. Recently, a few authoritative researchers have pointed out the potential of combining static and dynamic techniques [22,10,24] to make verification more usable; the present paper concurs in this vision.

Some of the aforementioned testing tools [13,23,4] already leverage lightweight static analyses to boost the performance of automated testing. Pex [25] is another scalable automatic testing framework, which relies more heavily on static methods: it exploits a variant of dynamic symbolic execution where an automated theorem prover (Z3) analyzes the symbolic executions to improve code coverage. Pex uses parameterized unit tests [26] as specifications. This makes it possible to test fairly sophisticated properties, but it also requires users to produce specifications in this customized form; contract specifications, however, seem more palatable to practitioners [5].

DASH [22] combines static and dynamic verification with an approach extending the software model-checking paradigm [3]: DASH's algorithm to generate exhaustive tests maintains a sound abstraction of the program, which can be used to construct automatically a correctness argument.

The few recent attempts at combining static and dynamic techniques tend to be specific conservative extensions of basic methods; the approach described in the present paper tries integration at a higher level to avail the complementarity of static and dynamic techniques to a larger extent.

4 The Tools of Eve

The integrated verification techniques currently available in Eve and illustrated in the preceding example session rely on two fundamental tools: AutoProof and AutoTest, which are now presented.

AutoTest. AutoTest [19]—now a standard component of commercial EiffelStudio—is a fully automatic contract-based testing tool. AutoTest generates objects by random calls to creation procedures. Preconditions select valid inputs and postconditions serve as oracles: every test case consists of the execution of a routine on objects satisfying its precondition; if executing the routine violate its postcondition or calls another routine without satisfying its precondition, the routine tested has a fault. A failing test case provides a concrete error report which is useful for debugging.

Like any dynamic technique based on execution, AutoTest handles every feature of the source language (Eiffel). Among its limitations, instead, is that random testing can take several hours to find the most subtle faults, and that complex specifications can exacerbate this problem.

AutoProof. AutoProof [20]—a more recent member of the Eiffel tool-set—is an automatic verification tool that translates Eiffel programs (with contracts) into annotated Boogie programs. AutoProof then uses the Boogie verifier [2] to check whether the Eiffel implementation satisfies its specification.

AutoProof improves on similar environments for static verification (e.g., Spec# [2]) by supporting some advanced language constructs such as function objects (agents in Eiffel terminology). Nonetheless, some features of Eiffel—most notably exceptions and floating point arithmetic—are still unsupported and routines using them are not adequately translated to Boogie. The performance of AutoProof depends on the quality of contracts available; accurate contracts improve the modularity of the analysis which can then also verify partial implementations.

5 The Advantages of Being Static and Dynamic

From a user's perspective, Eve's integration of static and dynamic tools can make verification more effective and agile in a variety of scenarios.

- Static verification is more modular and scales better to large systems made of several classes. It can also verify routines of deferred (abstract) classes which cannot

be tested because they cannot be instantiated. This indirectly improves the performance of testing as well, because the testing effort can focus on routines or classes not proved correct.

Conversely, whenever testing uncovers a faulty routine, the static tool stops trying to verify that routine. This policy may be broken down to individual clauses: for example, if testing finds a run of *remove_left_cursor* (Table 1) violating the postcondition clause *count* = **old** *count* − 1, it may still be worthwhile to try to prove the other clause, *c.index* = **old** *c.index* + 1.

- Dynamic analysis provides concrete reports of errors, which make debugging easier. For example, the following trace documents the error in the creation procedure *make_default* of Table 1:

```
create {ARRAYED_LIST} l.make_default (1)
    -- Inside make_default:
    l_v := Void ; Precursor (1) ; extend (l_v) -- l_v is Void
```

- Classes that have a faulty creation procedure or are deferred cannot be instantiated; testing cannot proceed in this case unless the constructor is fixed or an implementation of every routine is available. Static techniques do not incur these limitations: as illustrated in the example of Section 2, they can verify individual implemented routines even if others in the same class are deferred.
- Many core libraries rely on routines implemented through calls to low-level external routines (typically, in the Eiffel case, C functions); an example was *is_equal* in Table 1. Such routines are inaccessible to static analysis but are still testable. The integrated results of static and dynamic analysis on classes with such external routines reinforce the confidence in the correctness of the overall system.
- The combination of static and dynamic analysis can help detect discrepancies between the runtime behavior of a program and its idealized model. Examples are overflows and out-of-memory errors, which are often not accounted for in an abstract specification assuming perfect arithmetic and infinite memory. Consider, for example, a routine that updates the balance of a bank account as a result of a deposit operation:

```
deposit (amount: INTEGER)
    require amount > 0
    do balance := balance + amount
    ensure balance > old balance
end
```

balance: *INTEGER*

AutoProof, which models the type *INTEGER* as mathematical integers, verifies that the routine is correct. AutoTest, however, still finds a bug which occurs when **old** *balance* + *amount* is greater than the largest integer value representable and *balance* overflows. It is then a matter of general policy whether one should change the postcondition or the implementation. In any case, the comparison of the results of static and dynamic analysis clearly highlights the problem and facilitates the design of the best solution. With default settings, Eve gives a null correctness score

(Section 7) in such situations, which reflects the uncertainty and the need for further analysis.
– Complex contracts considerably slow down automatic testing, both because their runtime evaluation incurs a significant overhead and because random generation takes a long time to build objects that satisfy complex preconditions. Contracts may even in some cases be non-executable because they involve predicates over infinite sets; for example, the invariant of a class modeling a hash function requires that the hash code of every possible object (an infinite set) be a nonnegative integer. Static techniques can help in all such scenarios: it may be easier to prove the correctness of a routine if the precondition is complex, and hence also stronger; complex postconditions boost modular verification.

These observations highlight the usefulness of treating proofs and tests as complementary and convergent techniques (even though they have often been pursued, in the past, by separate research sub-communities in software engineering). There is indeed no contradiction; in particular, with the purpose of tests being entirely defined as attempting to make programs fail [18], a useful (that is, failed) test is a proof that the program is not correct. The approach illustrated by Eve is then to combine tools that can prove a program correct (such as AutoProof) and tools that can prove a program incorrect (AutoTest); as soon as a user has written a new program element, the two will start in parallel, each with its own specific goal, prove or disprove; in favorable situations, one of them will reach its goal fast, providing the user with a fast response of correctness or incorrectness.

6 The Design of an Integrated Verification Environment

The Eve integrated verification environment is built on top of the EiffelStudio IDE and supplements it with functionalities for verification.

Contracts. The choice of Eiffel as programming language ensures that we rely on formal specification elements embedded in the program text as *contracts* (pre and postconditions, class invariants, and other annotations). Since correctness is a relative notion (being dependent on a specification), every verification activity requires *some form* of specification. Empirical evidence suggests that formal specifications in the form of contracts are a good compromise between the rigor required by formal techniques and the kind of effort that practitioners are able, or willing, to provide [5, 21].

Not all contracts must be written by programmers: the architecture of Eve can accommodate components for *specification inference* (see Section 3) to help users write better and stronger contracts. This particular property, however, is not emphasized in the present paper, which focuses on the integration of static and dynamic analysis assuming some contracts are available.

Automation. A defining characteristic of the verification tools in Eve is that they are automatic and can do most of the work without any explicit input from the user, assuming the presence of contracts which Eiffel programmers are already used to provide [5]. In order to decouple the machinery of the individual verification tools and to filter out their output, Eve relies on a blackboard architecture [15] shown in Figure 2.

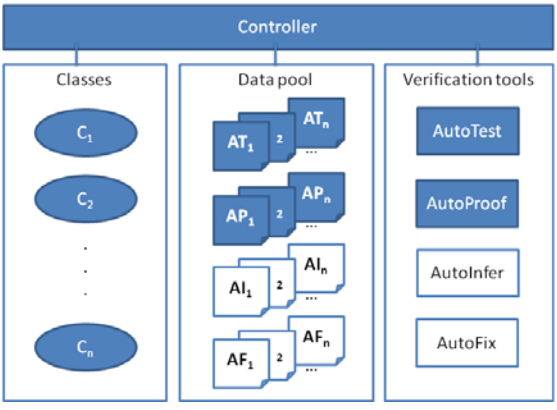


Fig. 2. Eve's blackboard architecture (shadeless boxes currently not implemented)

A *controller* is responsible for triggering the various tools when appropriate, invisibly to the users. The controller bases its decisions on what the user is currently doing, which resources are available, and the results of previous verification attempts. The latter are collected in a *data pool* where every verification tool stores the results of its runs. Users do not directly read the output of individual tools in the data pool. Instead, the controller summarizes the output data and displays individual tool results only upon explicit user request.

A major design decision of Eve was to make the verification mechanisms as unobtrusive as possible. Users can continue using the IDE and their preferred software development process as before; the verification techniques are an additional benefit, available on demand and compatible with the rest of the IDE's tools. In the same way that type checking adds a new level of help on top of the more elementary mechanisms of syntax error checking, Eve provides reports from proofs and tests on top of the simple verification techniques provided by type checking.

Interaction with the User. Users only have a coarse-grained, binary form of control over the verification: enable or disable. Typically, they will enable verification as soon as some code and a few contracts have been written. Even when enabled, verification never interferes with the more traditional development activities: Eve works in the background on the latest compiled version of the system, and displays a summary of the verification results through an interface similar to that used to signal syntax or type errors in standard IDEs (Figure 1). At any time, the user can browse through the *result list*, which links back to the parts of the program relevant for each message, and decide to revise parts of the implementation or specification according to the suggestions provided by Eve.

Every entry in the result list has a *score*: a quantitative estimate of the correctness or incorrectness of the associated entry, based on the evidence gathered so far by running the various tools. The score varies over the real interval $[-1, 1]$ (In the user interface the scale is, for more readability, -100 to $+100$, with rounding to the closest integer). A positive score indicates that the evidence in favor of correctness prevails, whereas

a negative score characterizes evidence against correctness. The absolute value of the score indicates the level confidence: 1 is conclusive evidence of correctness (for example a successful correctness proof), -1 is conclusive evidence of incorrectness (for example a failing test case), and 0 denotes lack of evidence either way. Figure 1 shows an example of report with scores and stripes colored accordingly. Section 7 discusses how the score is computed for the verification tools currently integrated in Eve.

Modularity and Granularity. Object-oriented design emphasizes modularity, from which verification can also benefit. While the level of granularity achievable by an integrated verification environment ultimately depends on the level of granularity provided by the tools it integrates, Eve orients verification at the two basic levels of encapsulation provided by the object-oriented model: *classes* and *routines* within a class. Eve associates *correctness scores* with items at both levels. Additional information may be attached to a correctness score, such as the line where a contract violation occurs in a test run, or the abstract domain used in an abstract interpretation analysis. For large systems, it is also useful to have scores for highest levels of abstraction, such as groups of classes or entire libraries, but in the present discussion we limit ourselves to routine and class levels.

The scores from multiple sources of data at the same level are combined with weighted averages, and define the correctness scores at coarser levels. For example, every tool t tries to verify a routine r in class C and reports a correctness score $s_r^C(t) \in [-1, 1]$. The cumulative score for the routine r is then computed as the normalized weighted average:

$$s_r^C = \frac{1}{\sum_{t \in T} w_r^C(t)} \cdot \sum_{t \in T} w_r^C(t) s_r^C(t) \tag{1}$$

where $w_r^C(t) \in \mathbb{R}_{\geq 0}$ denotes the weight of the tool t on routine r. A similar expression computes the cumulative score s^C for a class C from the scores s_r^C of its routines and their weights w_r^C:

$$s^C = \frac{1}{\sum_{r \in R} w_r^C} \cdot \sum_{r \in R} w_r^C s_r^C \tag{2}$$

The weights take various peculiarities into account:

- A tool may not be able to handle certain constructs: its confidence should be scaled accordingly. For example, a tool unable to handle exceptions appropriately has its score reduced whenever it analyzes a routine which may raise exceptions.
- The results of a tool may be critical for the correctness of a certain routine. For example, a quality standard may require that every public routine be tested for at least one hour without hitting a bug; correspondingly, the weight $w_r^C(t)$ for public routines r would be high for testing tools and low (possibly even zero) for every other tool.
- The correctness of a routine may be critical for a class; then the routine score should have a higher weight in determining the class cumulative score.
- More generally, the weight may reflect suitable *metrics* that estimate the criticality of a routine according to factors such as the complexity of its implementation or specification, whether it is part of the interface of public, and the number of references to it in clients or within the containing class.

– Similar metrics are applicable at other levels of granularity, for example to weigh the criticality of a class within the system.

Eve provides default values for all the weights (Section 7), but users can override them to take relevant domain knowledge into account.

Resource usage. The verification layer must not impede more traditional development activities. This requires in particular a careful usage of the resources to guarantee that the responsiveness of the IDE when verification is enabled does not cause a significant overhead. In Eve, the controller activates the various verification tools only when there are resources to spare. It also takes into account the peculiarities of the various tools: those with short running times are usually run first and re-run often, while those requiring longer sessions are activated later, only if the faster tools did not give conclusive results, and when enough resources are available.

When verification is first activated for a project, all the scores are null, as the system does not have any evidence to assess correctness or incorrectness. Then, the controller runs the least demanding tools first, to provide the user with *some* feedback *as soon as possible*. The same approach is applied modularly, whenever some part of the system changes (and is recompiled). More detailed analysis is postponed to when more time is available, the system is sufficiently stable, or conclusive evidence is still lacking about the correctness of some routines or classes.

Extensibility. The architecture of Eve is *extensible* to include more tools of heterogeneous nature. The user interface will stay the same, with the blackboard controller being responsible for managing the tools optimally and only reporting the results through the summary scores described above. It is our plan to integrate more verification tools, currently available only through explicit invocation.

The architecture can also accommodate tools that, while not targeted to verification in a strict sense, enhance the user experience. For example, tools for assertion inference—such as our own AutoInfer [29]—can complement user-provided contracts and improve the performance of approaches that depend on contracts. The controller can activate assertion inference when the verification machinery performs poorly and when metrics suggest that the code is lacking sufficient specifications. The assertion inference tools themselves may sometimes re-use the results of other tools; for example AutoInfer relies on the test cases generated by AutoTest. Finally, Eve can show the inferred assertions in the form of suggestions, in connection with the results of other verification activities. For example, it could display an inferred loop invariant with the report of a failed proof attempt, and suggest that the invariant can make the correctness proof succeed if added to the specification. The current implementation of Eve does not integrate such suggestions mechanisms yet, but the architecture is designed with these extensions in mind.

7 Correctness Scores for Proofs and Testing

Equation 1 on page 391 gives the correctness score for a routine r of class C; now, consider a set of tools $T = \{p, t\}$, where p denotes AutoProof and t denotes AutoTest.

General principles for scores. We noted earlier that an interesting test, that is to say a failed test, is a proof of incorrectness. This is of course another form of Dijkstra's famous observation about testing—but restated as an argument *for* tests rather than a criticism of the notion of testing. This observation has two direct consequences on the principles for computing correctness scores.

First, it is relatively straightforward to assign scores to the result of a testing tool when it reports errors: assign a score of -1, denoting certain incorrectness, to every routine where testing found an error. In certain special circumstances, the score might be differentiated according to the severity of the fault; for example a bug that occurs only if the program runs for several hours may be less critical than one that occurs earlier, if the system is such that it is reset every 45 minutes. In most circumstances, however, it is better to include such domain information in the specification itself and to treat every reported fault as a routine error. Then, different routines may still receive a different weight in the computation of the score of a class (Equation 2 on page 391)—for example, a higher weight to public routines with many clients.

The second consequence is that it is harder to assign a positive score sensibly to routines passing tests without errors. It is customary to assume that many successful tests increase the confidence of correctness; hence, this could determine a positive correctness score, which increases with the number of tests passed, the diversity of input values selected, or the coverage achieved according to some coverage criteria such as branch or instruction coverage. In any case, the positive score should be normalized so that it never exceeds an upper limit strictly less than 1, which denotes certain correctness and is hence unattainable by testing.

Since static verification tools are typically sound, a successful proof should generally give a score of 1. Certain aspects of the runtime behavior, such as arithmetic and memory overflows as discussed above, may still leak in some unsoundness if the static verifier does not model them explicitly; in such cases the score for a successful proof may be scaled down in routines with a specification that depends on such aspects.

Which score to assign to a static verifier reporting a failed proof attempt depends on the technique's associated guarantee of *completeness*. For a complete tool, a failed proof denotes a certain fault, hence a score of -1. If the tool is incomplete, a failed proof simply means "I don't know"; whether and how this should affect the score depends on the details of the technique. For example, partial proofs may still increase the evidence for correctness and yield a positive score.

Score and weight for AutoTest. If AutoTest reports a fault in a routine r of class C, the correctness score $s_r^C(t)$ becomes -1. This score receives a high weight $w_r^C(t) = 100$ by default; the user can adjust this value to reflect specific knowledge about the criticality of certain routines over others with respect to testing.

When AutoTest tests a routine r of class C without uncovering any fault, the score $s_r^C(t)$ increases proportionally to the length of the testing session and the number of test cases executed, but with an upper limit of 0.9. With the default settings, this maximum is reached after 24 hours of testing and 10^4 test cases executed without revealing any error in r. Users can change these parameters; the default settings try to reflect the specificities of random testing shown in repeated experiments [31]. We decided against using specific coverage criteria such as branch coverage in the calculation of the rou-

tine score, as the experiments suggest that for example the correlation between branch coverage and the number of uncovered faults is weak.

Score and weight for AutoProof. AutoProof implements a sound but incomplete proof technique. The score $s_r^C(p)$ for a routine r of class C is set accordingly: a successful proof yields a score of 1; an out-of-memory error or a timeout are inconclusive and yield a 0; a failed proof with abstract trace may be a faint indication of incorrectness: the abstract trace may not correspond to any concrete trace (showing an actual fault), but it often suggests that a proof might be possible with more accurate assertions. The score is then -0.2 to reflect this heuristic observation.

The weight $w_r^C(p)$ takes into account the few language features that are currently unsupported (floating point numbers and exceptions, see Section 3): if r's body contains such features, $w_r^C(p)$ is conservatively set to zero. In all other cases, $w_r^C(p)$ is 1 by default, but the user can adjust this value.

Routine Weights. Equation 2 (page 391) combines the scores s_r^C of every routine r of class C with weights w_r^C to determine the cumulative score of C. The weights w_r^C should quantify the relevance of routine r for the correctness of class C. This depends in general on the overall system design, which only developers can express appropriately, but which often depends on the visibility of a routine.

Eve supports a simple way to enter this piece of information: every routine has an optional *importance* flag which takes the values *low* and *high*. w_r^C is then

$$w_r^C = v_r^C \cdot i_r^C$$

The visibility of r determines v_r^C, which is 2 if r is public and 1 otherwise. The importance of r determines i_r^C, which is 2 if r has high importance, $1/2$ if it has low importance, and 1 it the developer did not set the importance flag.

8 Usage Scenarios

How serviceable is Eve's score which combines the results of different verification tools, as opposed to considering the tools' outputs individually? This section outlines a few straightforward scenarios that compare the output given by AutoProof or AutoTest in isolation against Eve's combined output; they show the greater confidence supplied by Eve, and the straightforward interpretability of its output. The example[1] models attributes of an individual with a class *PERSON*. Table 2 lists 5 routines of the class to be verified; for each routine, the table reports the score and weight of AutoProof and AutoTest within Eve, and the corresponding combined score.

Routine *set_age* demonstrates a favorable scenario, where each tool can provide strong positive evidence indicating correctness. The overall score is, correspondingly, quite high, but it still falls short of the maximum because testing can never prove the absence of errors with 100% confidence.

[1] The complete source code of the example is available at:
http://se.ethz.ch/people/tschannen/sefm2011_example.zip

Table 2. Individual and combined results for class *PERSON*

Item	Tool	Result	Weight	Score
set_age	AutoProof	Verified successfully	1.0	1.00
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.95**
increase_age	AutoProof	Verified successfully	0.5	1.00
	AutoTest	Overflow detected	100.0	-1.00
	Routine score			**-0.99**
age_difference	AutoProof	Verified successfully	0.5	1.00
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.93**
set_name	AutoProof	Proof failed	0.5	-0.10
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.57**
body_mass_index	AutoProof	Inapplicable	0.0	0.00
	AutoTest	No errors found	1.0	0.90
	Routine score			**0.90**
apply_command	AutoProof	Verified successfully	1.0	1.00
	AutoTest	Inapplicable	0.0	0.00
	Routine score			**1.00**
PERSON	**Class score**			**0.56**

Routine *increase_age* includes integer arithmetic, which might produce overflow. AutoProof can verify the routine, but Eve is aware that the proof scheme models integers as mathematical integers, hence it weights down the value of the successful proof because the abstraction may overlook overflow errors. Indeed, AutoTest reveals an overflow when executing the routine with the maximum integer value. The combined score indicates that there is an error, which AutoTest discovered beyond the limitations of AutoProof. Another routine *age_difference* also uses integer arithmetic but it is correct. Eve still scales down AutoProof's score accordingly; in this case, however, AutoTest does not find any error, hence the overall score grows high: the uncertainties of the two tools compensate each other and the cumulative score indicates confidence.

Routine *set_name* relies on the object comparison semantics, which AutoProof over-approximates. In this case, a failed proof does not necessarily indicate an error in the routine, hence it only accounts for a mildly negative score. When AutoTest does not find any error after thorough testing, the combined score becomes visibly positive, while still leaving a margin of uncertainty given the lack of conclusive evidence either way.

Routines *body_mass_index* and *apply_command* demonstrate how Eve's combination of tools expands the applicability of verification: *body_mass_index* uses floating point arithmetic, unsupported by AutoProof, whereas *apply_command* uses agents, unsupported by AutoTest. Eve relies entirely on the only applicable tool in each case.

The overall class score (last line of Table 2) uses a uniform weight for the routines; the score concisely indicates that considerable effort has been successfully invested in the class's verification, but some non-trivial issues are open.

9 Conclusions

Eve improves the usability of the individual verification tools by integrating them into an environment which features: automation and minimal direct user interaction; modularity at class and routine level; and extensibility with new tools. The current implementation of Eve [11] combines a static verifier for Hoare-style proofs and a dynamic

contract-based testing framework. The present paper has shown how these two techniques can be used in combination to improve the overall productiveness of verification.

Limitations and Future Work. In some situations, the integration of proofs and tests is still ineffective and provides an unsatisfactory user experience:

- When testing is the only technique applicable, it may be difficult to provide users with fast feedback. Automated random testing is very effective at finding delicate and unexpected bugs, but may require long sessions.
- The analysis of correct routines that use certain sophisticated language features—beyond those currently supported by AutoProof—may be inconclusive: testing does not find any error, but this is no substitute for a correctness proof.
- The completeness of contracts strongly affects the performance of verification. Weak contracts are easier to write and to reason about; strong contracts boost modular verification and expose subtler defects.
- Eve integrates multiple verification tools to complement their strengths and weaknesses. Different tools, however, may introduce discrepant models of the same implementation, such as for the class *INTEGER* discussed above. As the number of integrated tools grows, reconciling several contradictory semantics may become a delicate issue.

Future work will address these limitations to perfect the integration of testing and proofs; in particular, the following directions deserve further investigation.

- If AutoProof successfully verifies an assertion clause, the runtime checking of that specific clause can be disabled; this would contribute to speeding up the testing process. This improvement is currently unsupported because it requires a change in the Eiffel runtime to enable and disable the checking of individual assertion clauses.
- Integrating contract inference tools, such as our own AutoInfer [29], will assuage the problem of weak contracts that hold back the full potential of static provers. Another related synergy between static and dynamic techniques is the static verification of dynamically *guessed* contracts.
- A failed proof attempt usually comes with an *abstract* counterexample trace, which is, in general, not directly executable. The abstract trace may, however, provide enough information to suggest a *concrete* trace that is executable and show a real bug, or to conclude that the abstract trace is spurious. A spurious trace can help refine the proof model and sharpen the proof attempt, in a way similar to what done in the CEGAR (Counter-Example Guided Abstraction Refinement) paradigm [3].

The integration of more tools into Eve will improve the overall effectiveness of the various techniques and advance the quest towards the goal of *Verification As A Matter Of Course*.

Acknowledgements. The authors thank Nadia Polikarpova and Yi Wei for suggesting examples discussed in the paper. This work has been partially funded by the SNF grant SATS (200021-117995/1) and by the Hasler foundation on related projects.

References

1. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
4. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12 (2008)
5. Chalin, P.: Are practitioners writing contracts? In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Fault-Tolerant Systems. LNCS, vol. 4157, pp. 100–113. Springer, Heidelberg (2006)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
7. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: ISSTA (July 2010)
8. Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: Proceedings of OOPSLA, pp. 213–226 (2008)
9. Ernst, M.: Dynamically Discovering Likely Program Invariants. PhD thesis, University of Washington, US (2000)
10. Ernst, M.D.: How tests and proofs impede one another: The need for always-on static and dynamic feedback. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 1–2. Springer, Heidelberg (2010)
11. Eve: Eiffel verification environment, http://se.inf.ethz.ch/research/eve/
12. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI, pp. 234–245. ACM, New York (2002)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI, pp. 213–223. ACM, New York (2005)
14. Hoare, C.A.R., Misra, J.: Preface to special issue on software verification. ACM Comput. Surv. 41(4) (2009)
15. Hunt, J.: Blackboard architectures, JayDee Technology Ltd. 27 (2002)
16. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
17. Korel, B.: Automated test data generation for programs with procedures. In: Proceedings of ISSTA, pp. 209–215. ACM, New York (1996)
18. Meyer, B.: Seven principles of software testing. Computer 41, 99–101 (2008)
19. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. IEEE Software 42, 46–55 (2009)
20. Nordio, M., Calcagno, C., Meyer, B., Müller, P., Tschannen, J.: Reasoning about Function Objects. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 79–96. Springer, Heidelberg (2010)
21. Polikarpova, N., Ciupa, I., Meyer, B.: A comparative study of programmer-written and automatically inferred contracts. In: ISSTA, pp. 93–104 (2009)
22. Rajamani, S.K.: Verification, testing and statistics. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 33–40. Springer, Heidelberg (2009)
23. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of ESEC/FSE, pp. 263–272. ACM, New York (2005)
24. International conference on tests and proofs. LNCS. Springer, Heidelberg (2007-2010)

25. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
26. Tillmann, N., Schulte, W.: Parameterized unit tests. In: Proceedings of ESEC/FSE, pp. 253–262. ACM, New York (2005)
27. Tschannen, J., Furia, C.A., Nordio, M., Meyer, B.: Verifying Eiffel programs with Boogie. In: BOOGIE workshop (2011), http://arxiv.org/abs/1106.4700
28. Usable verification workshop (November 2010), http://fm.csl.sri.com/UV10/,
29. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring better contracts. In: Proceedings of ICSE 2011, pp. 191–200 (2011)
30. Wei, Y., Gebhardt, S., Oriol, M., Meyer, B.: Satisfying test preconditions through guided object selection. In: Proceedings of ICST 2010, pp. 303–312 (2010)
31. Wei, Y., Oriol, M., Meyer, B.: Is coverage a good measure of testing effectiveness? Technical Report 674, ETH Zurich (2010)

Efficient Computation of Dominance in Component Systems (Short Paper)*

Jaap Boender

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS,
F-75205 Paris, France

Abstract. To deal with the complexity of modern systems, they are split up into components. These components use metadata to specify whether they need other components to function correctly (dependencies) or whether they cannot function with other components (conflicts).

In previous work, we have presented component relationships that are more useful for analysis than those specified in the metadata: strong dependencies that identify components that are absolutely necessary, and dominance that allows us to gain insight in the structure of a component system by identifying clusters of related packages.

In this paper, we present an efficient way of computing this dominance relationship, by exploiting its similarity to the concept of dominance already known in the field of control flow graphs.

1 Introduction

The use of component-based systems has become a standard way of structuring large collections of software; not only programs, but also software distributions such as the different variations of free and open source (F/OSS) operating systems, or development environments such as Eclipse.

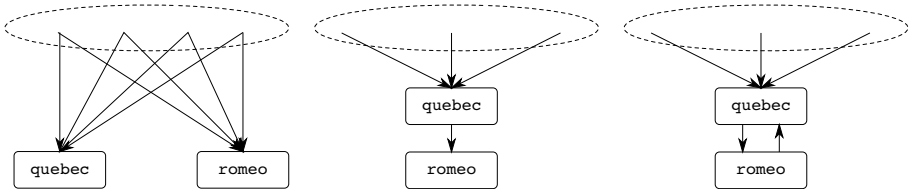
The components in component-based systems do not exist in isolation: one of the most useful properties of most component-based systems is their ability to specify *relations* between their components, such as dependency or conflict requirements.

Such inter-component relationships can be very complex. In the more complicated systems, relationships are expressed as formulae in conjunctive normal form, which makes their resolution equivalent to the problem of boolean satisfiability.

However, these relationships do not tell the whole story. If we look at the dependencies as they are specified in the component, the simple fact that a component is mentioned as a dependency of another only tells us that it *might* be needed: if the dependency is disjunctive, another component might be used to satisfy the dependency. There might also be a conflict that precludes the component from being installed altogether.

* Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project. Work developed at IRILL.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 399–406, 2011.



(a) Coincidence **(b)** Order **(c)** Equivalence

Fig. 1. Examples of strong dependency configurations

For this reason, it is interesting to look at the *semantic* relationships (as opposed to the *syntactic* relationships as mentioned in the component metadata). In [1], we have introduced *strong dependencies*. A component strongly depends on another if it is absolutely impossible to install without also installing the other components. In this way, we get a relationship that is based on both dependency and conflict information, and has a single meaning (as opposed to a simple dependency, which can either be conjunctive or disjunctive).

We can take this concept even further. Consider the diagrams from figure 1, which show some possible configurations of strong dependencies. In diagram 1a, the components that strongly depend on `quebec` and `romeo` are exactly the same, but there is no visible reason for this occurrence.

However, in diagrams 1b and 1c, the fact that the same components strongly depend on `quebec` and `romeo` is caused by the strong dependency of `quebec` on `romeo` (or vice versa). In a way, the strong dependencies of `quebec` *explain* the strong dependencies of `romeo`. We will refer to this concept as *dominance*: in diagram 1b, `quebec` dominates `romeo`, and in diagram 1c, `quebec` and `romeo` dominate each other.

This dominator relationship can be used to identify clusters of related components in a system. As an example, the dominator graph of a typical Linux distribution consists of a large number of independent clusters. This can help in gaining insight in the structure of a component-based systems, as well as in targeting maintenance efforts.

In this paper, we present an efficient method to compute the dominator graph of a distribution. We provide a conversion of the strong dependency graph to a control flow graph, and prove that the notion of dominance in the strong dependency graph is equivalent to the well-known notion of dominance in this control flow graph. In this way, we can use the well-known Lengauer-Tarjan algorithm to compute dominance.

2 Strong Dependencies and Dominators

In this section, we will start with briefly recapitulating some definitions from [1].

We start with the notion of *strong dependency*. Intuitively, a package p strongly depends on another package q if and only if it is absolutely impossible to install p without also installing q. Since installability is always defined with respect to a

repository, the strong dependency relation is defined with respect to a repository as well.

Definition 1 (Strong dependency). *Given a repository R, a package p strongly depends on a package q ($p \Rightarrow_R q$) if and only if:*

- *p is installable w.r.t. R*
- *every installation of p w.r.t. R includes q*

For this definition and the ones that follow, if it is clear from the context which repository R is used, we will omit it for the sake of brevity (and, in the case of strong dependencies, simply note $p \Rightarrow q$).

We insist upon the fact that p must be installable, otherwise a non-installable package would trivially strongly depend on every other package in the distribution.

Note that the strong dependency relation is reflexive and transitive.

We shall note the set of strong dependencies of a package p (i.e. $\{q \in R \mid p \Rightarrow_R q\}$) as $Scons(p, R)$.

This allows us to define the *impact set* of a package:

Definition 2 (Impact set). *Given a repository R, the impact set $Is(p, R)$ of a package p is the set $\{q \in R \mid q \Rightarrow_R p\}$.*

As noted in the introduction, it is possible that the impact set of one package is *explained* by the impact set of another. In other words, the fact that a package p has a large impact set is caused by the fact that it is a dependency of another package q, which has itself a large impact set.

Formally, we shall express this notion (which we call *dominance*, inspired by the notion of dominance in control flow graphs) as follows:

Definition 3 (Dominance). *Given a repository R, a package p dominates another package q ($p \succcurlyeq_R q$) if and only if:*

- *$p \Rightarrow_R q$*
- *$Is(p, R) \supseteq (Is(q, R) \setminus Scons(p))$*

Here, we capture the notion that the impact set of q contains only elements from the impact set of p (hence the impact set of q is explained by the impact set of p). We need to remove the strong dependencies of p, because it is possible that there are packages x such that $p \Rightarrow x$ and $x \Rightarrow p$.

With these definitions, a naive algorithm for the computation of the dominators in a distribution is easy to find: simply compute the strong dependency graph of a distribution (as shown in [1]), and for every two packages p and q such that $p \Rightarrow q$, see if the impact sets satisfy the condition for dominance.

There is, however, a very efficient algorithm, proposed by Lengauer and Tarjan in [6], to compute dominance in control flow graphs. In the next section, we will propose a method for converting the graph of strong dependencies of a distribution into a control flow graph, and prove that our notion of dominance is equivalent to dominance in this control flow graph.

In this way, we can use the Lengauer-Tarjan algorithm to compute the dominance graph in a distribution, which only takes a few minutes.

3 Dominance in Strong Dependency Graphs and Flow Control Graphs

In this section, we will present the notion of strong dependency graphs, a method for converting them into control flow graphs and finally a proof that they are equivalent with respect to dominance (as presented in the previous section for strong dependency graphs; for control flow graphs we use the classical notion of dominance).

This will allow us to use the Lengauer-Tarjan algorithm to efficiently compute dominance on open source distributions.

Definition 4 (Strong dependency graph). *The strong dependency graph $SG(R)$ of a distribution is the directed graph with all packages of R as its vertices, and as edges all pairs (p, q) such that $p \Rightarrow_R q$.*

Since the strong dependency relationship is transitive, $SG(R)$ is a transitive graph as well. This allows us to prove the following lemma:

Lemma 1. *If there is a cycle $v_1 \to v_2 \to \ldots \to v_n \to v_1$ in $SG(R)$, then $\{v_1, v_2, \ldots, v_n\}$ is a clique in $SG(R)$.*

Proof. For any $1 \leq i, j \leq n$, there must be a path from v_i to v_j (both are part of a cycle). Because $SG(R)$ is transitive, there must therefore also be an edge $v_i \to v_j$.

If there is a clique C in $SG(R)$, when considering installability, all the members of C are equivalent. They all strongly depend on each other, and therefore either all members of C will be installed, or none of the members of C will be installed. We will not lose any information (when it comes to installability) by amalgamating all members of C into one vertex.

Definition 5 (Collapse). *Given a directed graph G, we define its collapse $G\downarrow$ as the graph obtained by the following procedure:*

- *Let $G\downarrow$ be equal to G.*
- *For all* maximal *cliques $C = \{v_1, v_2, \ldots, v_n\}$ in G, do:*
  - *Add a vertex v_C to $G\downarrow$.*
  - *Replace all edges $v \to v_i$ $(1 \leq i \leq n)$ in $G\downarrow$ with $v \to v_C$, and all edges $v_i \to v$ with $v_C \to v$.*
  - *Remove all edges $v_C \to v_C$ from $G\downarrow$.*
  - *Remove all vertices in C from $G\downarrow$.*
- *Perform a transitive reduction on $G\downarrow$.*

The function φ_G is defined to map a vertex in G to its equivalent in $G\downarrow$.

We can show that collapsing a transitive graph (such as the strong dependency graph) results in a graph without cycles:

Lemma 2. *If G is a transitive graph, then $G\downarrow$ is acyclic.*

Proof. By lemma 1, every cycle in G is a clique.

Every clique in G is either a maximal clique itself, or completely contained within a maximal clique.

It is not possible for a clique C to be only partially contained within a maximal clique M: in this case, there is at least one vertex v that is within both M and C. Therefore, v is connected to every vertex of M and C. Since G is transitive, every vertex in M is connected to every vertex and C and vice versa. Hence, $M \cup C$ is a clique, which contradicts the assumption that M is a maximal clique.

Hence, if we replace all maximal cliques in G by single vertices in $G\downarrow$, we remove all cliques. Since every cycle is a clique, we also remove all cycles.

We can now construct the flow graph equivalent of the strong dependency graph.

Definition 6 (Flow graph). *Given a strong dependency graph $SG(R)$, the corresponding flow graph is obtained by taking $SG(R)\downarrow$, and adding to it an extra vertex* `start`*, and edges* $(\texttt{start}, p)$ *to every vertex p in $SG(R)\downarrow$ that does not have any predecessors.*

An example of the flow graph transformation is given in figure 2: first, the cycle between `bravo` and `charlie` is removed, and then the transitive reduction is performed (removing the edge from `bravo/charlie` to `foxtrot` and the edges from `alpha` and `delta` to `foxtrot`). Finally, a start vertex is added with edges to `alpha`, `bravo/charlie` and `delta`, all of which have no predecessors.

Every vertex in the flow graph is reachable from the start vertex:

Lemma 3. *For every vertex $v \in FG(R)$, there is a path from* `start` *to v.*

Proof. Any vertex $v \in SG(R)\downarrow$ must either have no predecessors (and therefore be directly reachable in $FG(R)$ from `start`), or have at least one predecessor. Since $SG(R)\downarrow$ is finite and acyclic, it is impossible to have an infinite path of predecessors; at some point we must reach a vertex that has no predecessors, and that therefore in $FG(R)$ is reachable directly from `start`.

We can prove that the flow graph retains the strong dependency relationship, modulo transitivity:

Lemma 4. *A vertex $\varphi(w)$ is reachable from a vertex $\varphi(v)$ in $FG(R)$ if and only if $v \Rightarrow w$.*

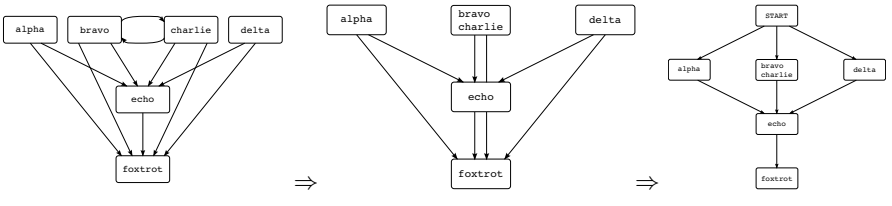


Fig. 2. Example of flow graph transformation

Proof.

($\Leftarrow$) We assume that $v \Rightarrow w$. To be proven is that $\varphi(w)$ is reachable from $\varphi(v)$.
Since $v \Rightarrow w$, there is an edge (v, w) in $SG(R)$. When constructing $SG(R)\downarrow$, either v and w end up in the same clique, which means that $\varphi(v) = \varphi(w)$ and reachability is trivial; or v and w are in different cliques, and the edge (v, w) becomes (through transitive reduction) a path from $\varphi(v)$ to $\varphi(w)$ in $SG(R)\downarrow$ and therefore in $FG(R)$.

($\Rightarrow$) We assume that $\varphi(w)$ is reachable from $\varphi(v)$. To be proven is that $v \Rightarrow w$.
Every vertex in $FG(R)$ represents a clique in $SG(R)\downarrow$, and therefore there is a list of cliques $C_1, C_2, \ldots, C_n$ such that $v \in C_1$, $w \in C_n$ and there are $x_i \in c_i$ such that $v \Rightarrow x_1$, $x_1 \Rightarrow x_2$, ..., $x_n \Rightarrow w$. Thus, $v \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_n \rightarrow w$ is a path from v to w in the transitive reduction of $SG(R)$, which means that there is an edge (v, w) in $SG(R)$, and therefore $v \Rightarrow w$.

Now we can establish the equivalence between the concept of dominance from definition 3 and the concept of dominance from flow graphs.

Theorem 1. *Given a repository R, $v \succcurlyeq_R w$ if and only if every path from* `start` *to $\varphi(w)$ in $FG(R)$ passes through $\varphi(v)$.*

Proof.

($\Leftarrow$) We assume that every path from `start` to $\varphi(w)$ in $FG(R)$ passes through $\varphi(v)$. To be proven is that $v \succcurlyeq_R w$, in other words $v \Rightarrow w$ and $Is(v) \supseteq (Is(w) \setminus Scons(v))$.

- By lemma 3, there is at least one path from `start` to $\varphi(w)$. This path, by the hypothesis, passes through $\varphi(v)$. Therefore, $\varphi(w)$ is reachable from $\varphi(v)$ and by lemma 4, $v \Rightarrow w$.
- Let x be a package in $Is(w, R) \setminus Scons(v)$. There must be a path from `start` to $\varphi(x)$ (lemma 3) and a path from $\varphi(x)$ to $\varphi(w)$ ($x \in Is(w, R)$, therefore $x \Rightarrow w$ and lemma 4). The concatenation of these two paths is a path from `start` to $\varphi(w)$ and therefore must contain $\varphi(v)$. Since $x \notin Scons(v)$, there is no path from $\varphi(v)$ to $\varphi(x)$, so $\varphi(v)$ must be on the path from $\varphi(x)$ to $\varphi(w)$. Therefore, there is a path from $\varphi(x)$ to $\varphi(v)$, which means that $x \Rightarrow v$ and therefore $x \in Is(v, R)$.

($\Rightarrow$) We assume that $v \succcurlyeq_R w$. To be proven is that every path from `start` to $\varphi(w)$ in $FG(R)$ passes through $\varphi(v)$.
In any path p from `start` to $\varphi(w)$ in $FG(R)$, we can find the first vertex $\varphi(x)$ such that $x \in Scons(v)$. This vertex exists, since $\varphi(w) \in p$ and $w \in Scons(v)$.
If $\varphi(x)$ is the direct successor of `start`, then $x = v$, since $x \in Scons(v)$, but x has no predecessors in $SG(R)\downarrow$. Hence, the path from `start` to $\varphi(w)$ passes through $\varphi(v)$ (since this is the same vertex as $\varphi(x)$).
Otherwise, there is a vertex $\varphi(x')$ directly preceding $\varphi(x)$ in p, such that $x' \notin Scons(v)$. However, there is a path from $\varphi(x')$ to $\varphi(w)$, so that

(by lemma 4) $x' \in Is(w)$. Therefore, $x' \in Is(w) \setminus Scons(v)$, and by the hypothesis, this means that $x' \in Is(v)$ and therefore there must be a path from $\varphi(x')$ to $\varphi(v)$ in $FG(R)$. Now there is a path from $\varphi(x')$ to $\varphi(x)$ through $\varphi(v)$ (since $x \in Scons(v)$), but there is also a direct edge from $\varphi(x')$ to $\varphi(x)$, which is in contradiction with the fact that $FG(R)$ is a transitive reduction.

4 Discussion

In the previous section, we have introduced a new method of computing the dominator graph of component-based systems, which is much faster than the naive method. We have proven that this method produces an equivalent result.

With this equivalence, we can use the Lengauer-Tarjan algorithm to compute the dominator graph of a distribution. This results in a significant gain in speed. The naive algorithm runs in $O(mn)$ time, with m the number of edges in the strong dependency graph and n the number of vertices (i.e. the number of packages in the distribution), whereas the most efficient implementation of the Lengauer-Tarjan algorithm runs in $O(m\alpha(m, n))$ time, with α the functional inverse of the Ackermann function. This is almost linear in time.

Here is a comparison of the run times of both algorithms on the latest Debian GNU/Linux distribution. We can see that the Lengauer-Tarjan algorithm is much faster. We do have to spend some time on doing the cycle reduction and the transitive reduction, but this is insignificant compared to the speed-up achieved by using the Lengauer-Tarjan algorithm for the generation of the dominator graph.

Activity	Naive	Lengauer-Tarjan
Generation of strong dependency graph	246s	247s
Cycle reduction	—	221s
Transitive reduction	—	35s
Generation of dominator graph	24415s	74s

The run time of the Lengauer-Tarjan algorithm being only a few minutes on a typical distribution, it is possible to run the algorithm daily on the development version of the distribution, to keep track of changes in its structure. It should even be possible to re-generate the dominance graph every time a new package is submitted, to easily note the impact of the changes on the package structure.

Related Work. There is extensive literature on the formal study of component-based systems. In this section, we will discuss some papers that discuss similar subjects to our own.

In [2], the concept of *predictable assembly* is presented, defined as the properties of assemblies of components. This is similar to our study of component-based systems, most notably F/OSS distributions.

The method presented above could also be adapted to software product lines, expressed as feature diagrams [8]. It has been shown that a significant subset of feature diagrams can be encoded as component problems [3].

We have tested our method on F/OSS distributions, most notably the Debian and Mandriva distributions. It could also be adapted to other component-based systems, such as Eclipse plugins [5] or the OSGi component model.

Some work has been done on the exploration of the structure of F/OSS distributions, as an instance of small-world networks [4,7]. These studies also show that F/OSS distributions are highly clusterised.

Acknowledgements. The author would like to thank Roberto Di Cosmo, Pietro Abate, Yacine Boufkhad, Ralf Treinen and Stefano Zacchiroli for many fruitful discussions.

References

1. Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S.: Strong dependencies between software components. In: ESEM 2009: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 89–99. IEEE Computer Society, Washington, DC, USA (2009)
2. Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K.: Anatomy of a research project in predictable assembly. In: Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering (2002), White paper
3. Di Cosmo, R., Zacchiroli, S.: Feature diagrams as package dependencies. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 476–480. Springer, Heidelberg (2010)
4. LaBelle, N., Wallingford, E.: Inter-package dependency networks in open-source software. CoRR, cs.SE/0411096 (2004)
5. Le Berre, D., Rapicault, P.: Dependency management for the eclipse ecosystem: An update. In: 3rd International Workshop on Logic and Search (Lash 2010) (July 2010)
6. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. 1(1), 121–141 (1979)
7. Maillart, T., Sornette, D., Spaeth, S., von Krogh, G.: Empirical Tests of Zipf's Law Mechanism in Open Source Linux Distribution. Phys. Rev. Lett. 101(21), 218701 (2008)
8. Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature diagrams: A survey and a formal semantics. In: RE 2006, pp. 136–145. IEEE, Los Alamitos (2006)

The Boogie Verification Debugger (Tool Paper)

Claire Le Goues[0], K. Rustan M. Leino[1], and Michał Moskal[1]

[0] University of Virginia, Charlottesville, VA, USA
legoues@cs.virginia.edu
[1] Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Abstract. The Boogie Verification Debugger (BVD) is a tool that lets users explore the potential program errors reported by a deductive program verifier. The user interface is like that of a dynamic debugger, but the debugging happens statically without executing the program. BVD integrates with the program-verification engine Boogie. Just as Boogie supports multiple language front-ends, BVD can work with those front-ends through a plug-in architecture. BVD plug-ins have been implemented for two state-of-the-art verifiers, VCC and Dafny.

0 Introduction

Deductive software verification technology is sufficiently mature that tools can formally verify non-trivial programs written in semantically intricate modern languages. However, these tools remain difficult to use. They require considerable expertise, patience, and trial-and-error, especially to decipher error conditions and verification failures. In sum: our tools can understand our programs, but we cannot understand our tools.

In this paper, we present a *verification debugger*, called BVD (Boogie Verification Debugger), to help users understand the output of a program verifier. Our tool advances the state-of-the-art in program verification by increasing the communication bandwidth between the verifier and the user. Much as a dynamic debugger allows a programmer to explore a failing run-time state, the verification debugger allows her to explore—and, by extension, understand—a failing verification state. For example, a user can inspect the assumed values of local and heap-allocated variables. Constructing such a debugger is challenging because of the disconnect between the theorem prover counterexample model and the programmer's mental model of program execution.

Verification tools vary in their automation levels. At one extreme lies fully automatic verifiers. This class of verifier includes many abstract interpreters or model checkers; to obtain full automation, such tools typically limit their reasoning to certain domains. At the other extreme lies verifiers that accept user input at all proof steps. This class of verifier describes many mechanical proof assistants; to obtain this flexibility, these tools typically expose the user to the internal representation of proof obligations. In this paper, we consider a family of verifiers between the automatic and interactive extremes, which we refer to as *auto-active verification*. An auto-active verifier has two major characteristics: the user can define assertions for the verifier to prove, and all user interaction is done at the level of the program source language. For example, a user may help the verifier along by declaring a precondition or loop invariant in the program, but

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 407–414, 2011.

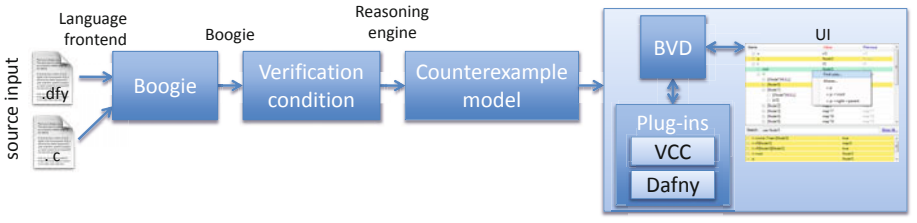


Fig. 0. BVD in the context of a failed verification in the Boogie system. Input source is compiled to Boogie, which Boogie translates into a verification condition (VC). When the reasoning engine cannot discharge the VC, it produces a counterexample model. BVD interacts with a language-specific plug-in to interpret and display the counterexample model to the user.

never interacts directly with the underlying reasoning engine. Examples of auto-active verifiers include extended static checkers [4] and program verifiers like Dafny [8] and VCC [2].

A prevalent implementation technique for auto-active verifiers is to translate the source program and its user annotations into an *intermediate verification language*, like Boogie [1] or Why [5]. This intermediate program is used to generate a verification condition; this condition is passed to a reasoning engine, for example a Satisfiability Modulo Theories (SMT) solver like Z3 [3]. Intermediate verification languages like Boogie confer similar benefits to verification as intermediate program representations do to compilation. In particular, they allow different source-language front-ends to share a verifier back-end. BVD, presented in this paper, works with Boogie and provides a plug-in architecture to support different front-ends. We have built BVD plug-ins for VCC [2] and Dafny [8].[0]

The rest of this short paper is organized as follows. Section 1 situates BVD in the context of the architecture of the Boogie pipeline. Section 2 describes the features, implementation, and use of BVD on indicative examples, and provides screenshots. Section 3 presents related work, while Sect. 4 discusses future work and concludes.

1 High-Level architecture

Fig. 0 provides a high-level overview of the architecture of BVD as it fits in the pipeline of a failed verification in Boogie. The Boogie verification system supports multiple language front-ends. Input source code is transformed by a language-specific front-end into the Boogie intermediate verification language. The Boogie tool then transforms the intermediate program to generate a verification condition (VC) to pass to a reasoning engine. When verification fails, the underlying reasoning engine produces a counterexample model under which the postcondition does not hold. Unfortunately, this counterexample model, consisting of a list of elements and interpretations of functions used in the language encoding, does not map transparently back to the source code. BVD

[0] Boogie, BVD, and Dafny are available as open-source projects at `http://boogie.codeplex.com/`. VCC is available with source at `http://vcc.codeplex.com/`.

interacts with a language plug-in to interpret the counterexample model and display it in a way meaningful to the end-user, and not only the verifier author.

This paper is primarily concerned with the boxed region of the diagram, which corresponds to the BVD tool. BVD support requires minor modification to the front-end compiler, described below; such modifications are typically negligible in practice.

2 BVD

BVD is geared towards the debugging of failed verifications of imperative languages, characterized by sequential execution states. Typically, regular debuggers display one execution state at a time, through which one may step forward, and in some cases backward (e.g., the OCaml debugger or IntelliTrace in VS2010). Similarly, an SMT counterexample model consists of *states* leading to a failed assertion. Each state ultimately corresponds to a location in the source code, and encodes relevant memory contents at those source locations. BVD translates the SMT state sequences into "execution" states and memory values that the user can understand, and displays them in an informative way. This section is concerned with the details of this translation process.

2.0 Insert example

To make the discussion more concrete, consider the following Dafny program implementing a linked list with insertion. The programmer has annotated the method to provide information to the verifier (line 2) and to tell the verifier what it should prove (line 4, which asserts that **n** has been successfully inserted):

```
0  class ListNode { var data: int; var next: ListNode; }
1  static method Insert(hd: ListNode, n: ListNode)
2    requires hd != null ∧ n != null;
3    modifies hd, n;
4    ensures n.next == old(hd.next);
5  { n.next := hd.next; hd.next := n; }
```

This method fails to verify because it is possible for **hd** and **n** to be aliases on entrance to **Insert**, leading both **hd->next** and **n->next** to point to **n** after line 5. This violates the postcondition. This can be addressed by adding a precondition asserting **hd != n** on method entrance.

However, the verifier provides very little information to help the programmer understand this failure. Its error message states that the postcondition is violated, and that line 5 is implicated. The underlying reasoning engine's counterexample model is similarly unhelpful. A Z3 error model, for example, represents program states as named model elements, such as ***0 -> true, *37 null** and functions interpretations on elements, such as **MapType1Select -> *40 *41 *17 -> *56; else -> #unspecified**

We will use this example to clarify the discussion of our tool, and will demonstrate the tool on it and other examples.

2.1 Computing Memory Contents

The model produced by the reasoning engine contains values for a number of memory locations in a number of states. Regular debuggers allow the user to inspect values of variables as well as the values of fields of objects pointed to by the variables. BVD follows this model by translating the SMT's model of memory contents into more familiar object trees that the user can explore.

BVD names memory locations using *access paths*—usually source-level expressions that access a given memory location. Example access paths in the `Insert` example include `hd`, `n.next`, and `hd.next.data`. After the execution of `Insert()`, `n.data` and `hd.next.data` name the same memory location. In addition to local variables, BVD can use Skolem constants as roots, allowing the user to explore verification-specific states as necessary.

Values of memory locations are expanded recursively via communication between BVD and the language plug-in. BVD supplies the program points corresponding to counterexample states to the language-specific BVD plug-in. The latter returns a list of root paths—typically local and global variables—available at the static program points. BVD then supplies values of these root paths according to the model, which the plug-in "expands" to yield additional interesting access paths. The expansion may contain source-level fields (when the value is a pointer to an object), indexes into arrays, maps, sets, or applications of functions. The process repeats recursively, in breadth-first order, stopping whenever an already visited node is reached.

2.2 Displaying States and Access-Path Values

Canonical names. Some values, like integers, Booleans, or the special `null` pointer, are easy to display. In contrast, regular pointers in the SMT model do not have user-meaningful values; they are assigned place-holder values. BVD asks the language plug-in to provide canonical (*i.e.*, state-independent) names for them. The VCC and Dafny plug-ins use the type of the object and a sequential number (*e.g.*, `ListNode'0`).

Stable tree view. BVD displays values for both the currently and previously selected execution states, in blue and red respectively, allowing side-by-side comparison. Some paths may be available only in some states, *e.g.*, `hd.next.next` is unavailable in state 1 because `hd.next` is `null`, but is available in state 2. To avoid abrupt changes to the access-path tree while browsing the states, the left pane shows the union of path trees for all states. A path's name is greyed out if its value is unavailable in the selected state.

Fig. 1 shows BVD on the counterexample produced by Dafny on the `Insert` program. Paths `hd` and `n` are roots; `hd.next` was generated by the Dafny BVD plug-in as the expansion of `hd`. The screenshot displays `hd` and `n` in state 1 (Value) as compared to state 2 (Previous). The right-hand pane displays the value of `hd.next` in all states. The use of canonical names illustrates that `hd` and `n` are aliased: they have the same value (`ListNode'0`). The current state shows the value of `old(hd.next)`: (`null`). If we advance to the final state (corresponding to the execution of the last source code line) we will see that `n.next` points to `ListNode'0`, not `null`.

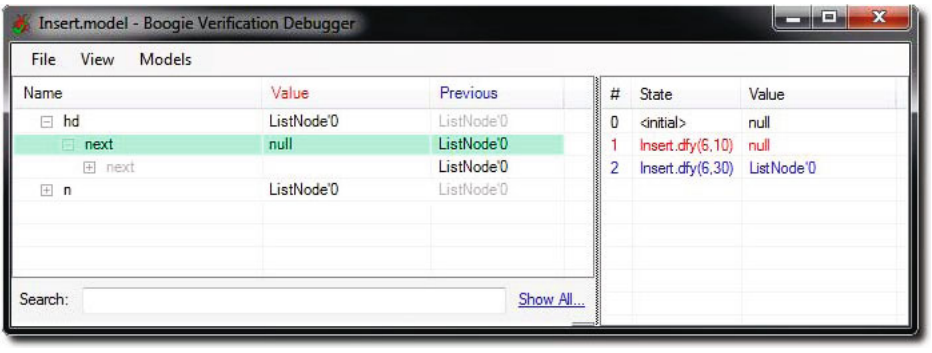


Fig. 1. BVD on the Dafny **`Insert`** example. The right pane provides navigation through execution states, and shows the value of the currently selected access path throughout execution. For example, **`hd.next`** is **`null`** in the first two states, and **`ListNode'0`** in the third. The left pane allows browsing the access path tree and inspecting values in the current (in red) and previously selected (in blue) state, allowing for comparison. Canonical names are provided by the language plug-in for model values that are not otherwise user-meaningful (such as **`ListNode'0`**).

2.3 Complex Data Types and Search

Skolem constants. The screen shots in Figs. 2 and 3 are of a failed VCC verification of a recursive red-black tree implementation. Fig. 2 illustrates the use of canonical names to display complex data type values. The **`m@Red..(79,15)`** is a Skolem constant (**`m`**) for which a quantified invariant at line 79 column 15 fails to hold.

Sparse data structures. Fig. 2 also shows a ghost field **`t->owns<Tree>`**, introduced by the VCC verification methodology and containing the set of objects owned by **`t`**. The set is displayed using canonical names: it contains **`Node'0`**, does not contain **`Node'9`**, and so on. The set representation in the SMT model is *sparse*—it does not say anything about possible **`Node'42`**, because it does not need to: the value of **`Node'42`** is irrelevant to the verification failure. We use a similar display format for maps, arrays, and specification functions (either methodology-supplied like **`$inv2(...)`**, or user-defined).

Search. Canonical names are useful for spotting aliases and understanding changes of variables across states. The user may search for aliases or uses of a given access path value. In a given state, a value is used by an access path if either the location pointed to by the access path contains the value, or the access path itself mentions the value. Fig. 3 shows screenshots of some of BVD's search capabilities. To determine correspondence between canonical names and variables in the current state, the user right-clicks on an access path (left screen shot) or uses the search facility (right screen shot).

Both these features are desirable and possible due to the size of the models: the models are in the range of hundreds (up to a few thousands) of elements—too large for the human being to look at as a whole, but much smaller than the gigabytes of heap that a dynamic debugger may need to consider to provide such features.

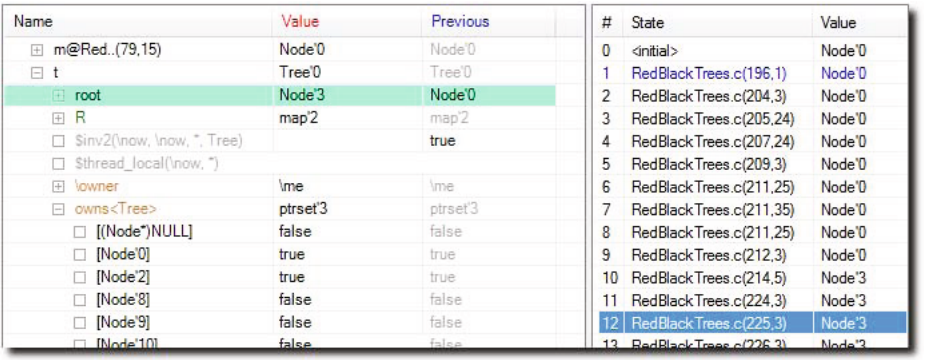


Fig. 2. BVD on a failed VCC verification of a recursive red-black tree implementation. Skolem constants receive canonical names (such as **`m@Red..(79,15)`**) as with regular variables. This example demonstrates BVD's treatment of sparse data structures: only values relevant to the counterexample—those with values in the model—are displayed. **`Node'42`**, for example, is not displayed in the ghost field **`t->owns<Tree>`**, because it is not relevant to the failure.

2.4 Plug-in Programmer Interface

The translator from source language to Boogie needs to insert special assumption statements at program points that capture source code locations and variable values, which are included in the counter-example model and can then be used to decode states displayed in BVD. The source-language BVD plug-in is responsible for enumerating access paths. BVD provides the plug-in with a mapping from Boogie variables to model elements in each marked program point, as well as complete enumeration of model elements and associated function interpretations.

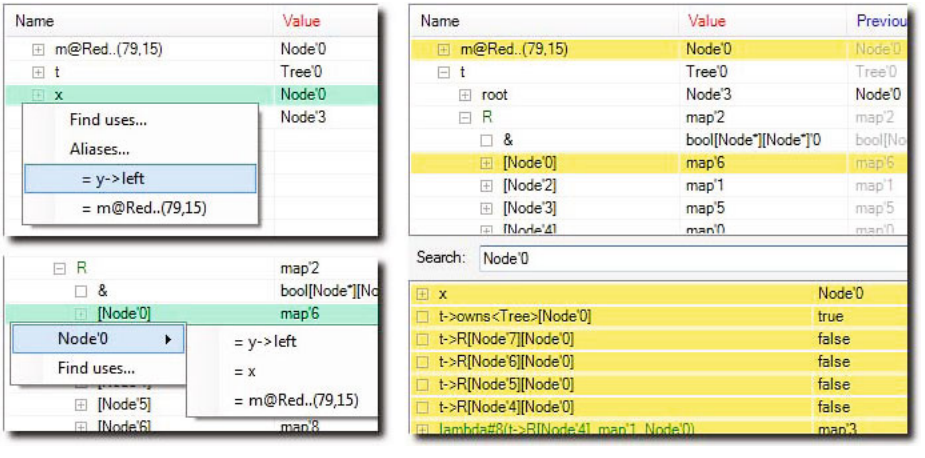


Fig. 3. Search on the red-black tree example. Right-clicking a node produces commands to jump to other access paths containing the current value (aliases). The pop-up menu at the bottom shows three locations containing **`Node'0`** in the current state; the menu above it shows two other locations besides **`x`** (currently selected). The menu also allows populating the full-text search box.

Writing language plug-ins is not difficult. The VCC plug-in is about 1000 lines of C# code, while Dafny is only about 400 (mostly due to Dafny being a leaner language). The modifications needed in the C and Dafny to Boogie translators are negligible.

3 Related Work

The idea of adding instrumentation to verification conditions for the purpose of generating usable error messages is old. For example, ESC/Modula-3 labeled sub-formulas in verification conditions and used the labels returned by the SMT solver to determine the source location of the potential program error [4]. ESC/Java extended the mechanism to allow the reconstruction of an execution trace leading to the error [9]. The Spec# verifier extended Boogie with a mechanism to retrieve the values of certain pre-determined expressions in the error state; for example, this lets the verifier report the value used as an index in an array bounds error.[1]

The forerunner to our work was the VCC Model Viewer[2], which provides a debugger-like, interactive user interface to explore the verification state [2]. BVD integrates into Boogie rather than building on top of it, permitting a simpler encoding. In addition, BVD's architecture supports plug-ins for multiple languages, and through its use of canonical names, permits more advanced features like stable tree views and search.

Müller and Ruskiewicz have implemented a Visual Studio dynamic debugger plug-in for Spec#, with the same purpose [10] as our tool. The debugged program is a variation of the original program that uses values from the SMT model to construct concrete data structures; these are used according to the verification semantics of the program. For example, instead of iterating a loop, the verification semantics immediately jumps to the final loop iteration, where the values of variables are constrained only by the loop invariant, which is how the program verifier views the execution. Their tool can identify some spurious error reports. Our approach is simpler, in that it avoids the many problems associated with constructing concrete data structures from a mathematical model. Furthermore, our approach makes explicit the partial information considered by the SMT solver, which lets us sparsely represent arrays and maps and show functions.

There has also been work to improve the user experience with software model checkers. The typical output of a model checker is a full execution trace leading to an error. There has been work to prune these enormous traces by comparing successful executions with failing executions, seeking to determine the cause of the error [0, 6].

The auto-active verifier VeriFast is based on symbolic execution and works with both C and Java programs [7]. Its IDE displays, at each program point, both the heap structure and the constraints on the values of variables and heap locations. It does not currently display concrete values for variables, though it could in principle.

4 Conclusions and Future Work

We have presented BVD, a multi-language verification debugger that helps programmers decipher and diagnose program verifier output. We have built BVD plug-ins for

[1] This `-enhancedErrorMessages` mode was implemented by Ralf Sasse.

[2] Developed by Markus Dahlweid and Lieven Desmet.

VCC and Dafny and found the verification debugger to be useful in practice. For example, it has elucidated why, in the `SiftUp` and `SiftDown` methods of a priority-queue heap data structure, it is necessary to include a loop invariant that establishes a connection between the parent and children of the node being updated.[3] We believe that tools like BVD are necessary to move verification into the hands of non-experts.

As future work, we wish to conduct user experiments with verification non-experts, perhaps in a teaching setting. We wish to add features that further help narrow the cause of an error (not just debug the symptoms) or identify spurious error reports (see Sect. 3). Adding to the textual tree views provided in BVD, we would like to see complementary visualization (*e.g.*, nodes and arrows) of the data structures in error states. Finally, we would like to see an even tighter integration of aids like a verification debugger into IDEs, so that it can become standard practice to have verification and diagnostic information available to the programmer immediately as code is being designed.

References

0. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. In: POPL 2003, pp. 97–105. ACM, New York (2003)
1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. 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.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
3. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center (December 1998)
5. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
6. Groce, A., Kröning, D., Lerda, F.: Understanding Counterexamples with `explain`. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004)
7. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
8. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
9. Leino, K.R.M., Millstein, T.D., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Sci. Comput. Program. 55(1-3), 209–226 (2005)
10. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 73–87. Springer, Heidelberg (2011)

[3] See `Test/dafny1/PriorityQueue.dfy` at `boogie.codeplex.com`.

Object-Oriented Formal Modeling and Analysis of Interacting Hybrid Systems in HI-Maude

Muhammad Fadlisyah[1], Peter Csaba Ölveczky[1], and Erika Ábrahám[2]

[1] Department of Informatics, University of Oslo, Norway
[2] Computer Science Department, RWTH Aachen University, Germany

Abstract. This paper introduces the HI-Maude tool that supports the formal modeling, simulation, and model checking of *interacting hybrid systems* in rewriting logic. Interacting hybrid systems exhibit both discrete and continuous behaviors, and are composed of components that influence each other's continuous dynamics. HI-Maude supports the *compositional* modeling of such systems, where the user only needs to describe the dynamics of *single* components and interactions, instead of having to explicitly define the continuous dynamics of the entire system. HI-Maude provides an intuitive, expressive, object-oriented, and algebraic modeling language, as well as simulation and LTL model checking with reasonably precise approximations of continuous behaviors for interacting hybrid systems. We introduce the tool and its formal analysis features, define its formal semantics in Real-Time Maude, and exemplify its use on the human thermoregulatory system.

1 Introduction

Modelers typically have to choose between *simulation tools* that provide intuitive and expressive modeling languages but only support system simulation, and *model checking* tools that can do powerful formal analyses but only provide quite restrictive modeling formalisms to ensure that key properties are decidable. For *real-time systems*, the rewriting-logic-based *Real-Time Maude* tool [14] tries to bridge this gap by providing an intuitive and expressive object-oriented modeling formalism as well as both simulation, reachability, and LTL and TCTL model checking. Although the price of this expressiveness is that properties are in general undecidable, useful formal analyses can often be performed on very complex systems; furthermore, Real-Time Maude model checking is sound and complete model checking for many systems encountered in practice [13]. Real-Time Maude has proved to be useful for analyzing a wide range of advanced applications that are beyond the scope of timed automata (see, e.g., [15]).

This paper introduces the *HI-Maude* tool that extends Real-Time Maude to support the modeling, simulation, and model checking of *hybrid systems* with combined discrete and continuous behaviors. We target complex hybrid systems in which multiple physical entities interact and influence each other's continuous behavior. For example, a hot cup of coffee in a room interacts with the room

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 415–430, 2011.

through different kinds of heat transfer, leading to a decrease of the coffee's temperature and to a slight increase of the room's temperature. One distinguishing feature of HI-Maude is the *modularity* and *compositionality* of the specification of the system's continuous dynamics. Non-compositional specification of the *whole* system is very hard, as it involves combining the ordinary differential equations (ODEs) that specify the dynamics of its components; it also requires redefining the system's continuous dynamics for each new configuration of interacting physical components. To achieve the desired modularity and compositionality, HI-Maude offers an object-oriented modeling methodology [5], based on the *effort/flow approach* [18], that allows us to specify the continuous dynamics of *single physical entities* (such as the cup of coffee and the room) and of *single physical interactions* (such as thermal conduction and convection). To analyze the system, HI-Maude uses adaptations of different numerical methods (the Euler method and the Runge-Kutta method of different orders) to give fairly precise approximate solutions to coupled ordinary differential equations.

Several simulation tools for hybrid systems, such as MATLAB/Simulink [17], HyVisual [12], and CHARON [4], are based on numerical methods. In contrast to these tools, HI-Maude also supports reachability analysis and temporal logic model checking. Of course, the results of such model checking must be seen in light of the approximation inaccuracies of the continuous behaviors. Our approach also differs from model checkers for hybrid systems, such as CheckMate [1], PHAVer [8], d/dt [3], and HYPERTECH [10], in that we do not use abstraction or over-approximation. Whereas other formal tools use hybrid automata, chart or block models, or formulas for modeling, a main advantage of HI-Maude is that it is based on the intuitive yet expressive *rewriting logic* formalism as the underlying modeling formalism.

To summarize, HI-Maude provides:

1. a modeling framework for hybrid systems that is (i) intuitive and expressive, (ii) object-oriented, with support for features such as inheritance and dynamic creation and deletion of objects, (iii) algebraic, and that (iv) makes it easy to specify the continuous dynamics in a simple and compositional way;
2. simulation, reachability analysis, and LTL model checking for such models based on fairly precise approximations of the system's continuous behavior.

We exemplify the use of HI-Maude with a small example of the coffee in the room, as well as with the formal modeling and analysis of the human thermoregulatory system. Both these systems are outside the decidable fragment of hybrid automata, in part due to their complex non-linear continuous dynamics.

The paper is structured as follows: Section 2 gives an overview of Real-Time Maude. Section 3 briefly explains our effort/flow-based method proposed in [5] for modeling hybrid systems in rewriting logic. Section 5 describes the HI-Maude tool, Section 6 briefly outlines its semantics, and Section 7 gives an overview of the modeling and analysis of the human thermoregulatory system in HI-Maude. Finally, some concluding remarks are given in Section 8.

The HI-Maude tool, together with some examples and a longer technical report, is available at `http://folk.uio.no/mohamf/HI-Maude`.

2 Real-Time Maude

A Real-Time Maude [14] *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [2] theory with Σ a signature[1] and E a set of *confluent and terminating conditional equations*. (Σ, E) specifies the system's state space as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.
- IR is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written with syntax `rl [`l`] :` t `=>` t', where l is a *label*. Such a rule specifies a *one-step transition* from an instance of t to the corresponding instance of t'. The rules are applied *modulo* the equations E.[2]
- TR is a set of (usually conditional) *tick rewrite rules*, written with syntax `crl [`l`] : {`u`} => {`v`} in time` τ `if` *cond*, that model time elapse. `{_}` encloses the global state, and the term τ denotes the *duration* of the rewrite.

The Real-Time Maude syntax is fairly intuitive. A function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a subterm $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied. We refer to [2] for more details on the syntax of Real-Time Maude.

In *object-oriented* Real-Time Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a state is represented as a term `<` O `:` C `|` $att_1 : val_1, \ldots, att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a *concurrent* object-oriented system, the state is a term of sort `Configuration`. It has the structure of a *multiset* of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is *multiset rewriting* supported in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] :  < O : C | a1 : 0, a2 : y, a3 : w, a4 : z >  =>
          < O : C | a1 : T, a2 : y, a3 : y + w, a4 : z >
```

[1] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*

[2] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

defines a parametrized family of transitions which can be applied whenever the attribute `a1` of an object `O` of class `C` has the value `0`, with the effect of altering the attributes `a1` and `a3` of the object. "Irrelevant" attributes (such as `a4`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

Formal Analysis. A Real-Time Maude specification is *executable*, and the tool offers a range of formal analysis methods. The *rewrite* command simulates *one* behavior of the system *up to a certain duration*. The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system, by checking whether states matching a *pattern* and satisfying a *condition* can be reached from the initial state. The command searching for n such states reachable within time τ has syntax `(tsearch [`n`]` t `=>*` *pattern* `such that` *cond* `in time <=` τ `.)`.

Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a temporal logic formula. *State propositions*, possibly parametrized, can be predicates characterizing properties of the state and/or properties of the global time of the system. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `W` ("weak until").

3 Effort/Flow Modeling of Interacting Hybrid Systems

Our tool is based on the modeling methodology in [5], which adapts the *effort/flow* method [18] to model a physical system as a network of *physical entities* and *physical interactions* between the entities.[3] This makes the models *modular* and *compositional*, in the sense that it is sufficient to define the continuous dynamics for each component to define the dynamics of the entire system.

As shown in Fig. 1, a *physical entity* is described by a real-valued *effort*, a set of *attribute* values, and the entity's *continuous dynamics*. The attribute values describe discrete properties, e.g., the mass or the phase of a material, that can only be changed by discrete events. The effort variable represents a dynamic physical quantity, such as temperature, that evolves over time as given by the continuous dynamics in the form of an ordinary differential equation (ODE).

A *physical interaction* between two physical entities is described by a real-valued *flow*, a set of *attribute* values, and a *continuous dynamics*. The flow value describes the dynamic interaction between two entities, whose evolution over time is specified by the continuous dynamics. The values of the effort variables of the two physical entities are used in the definition of the continuous dynamics of the interaction. In a *flow source* we abstract from one of the entities and model a constant flow to a single entity.

[3] The approach using effort/flow variables is applicable to different areas of physical systems. In mechanical translation systems, the pair of effort and flow variables are force and velocity; in mechanical rotation systems, torque and angular velocity; in electrical systems, voltage and current; in fluidic systems, pressure and volume flow rate; in thermal systems, temperature and heat flow rate.

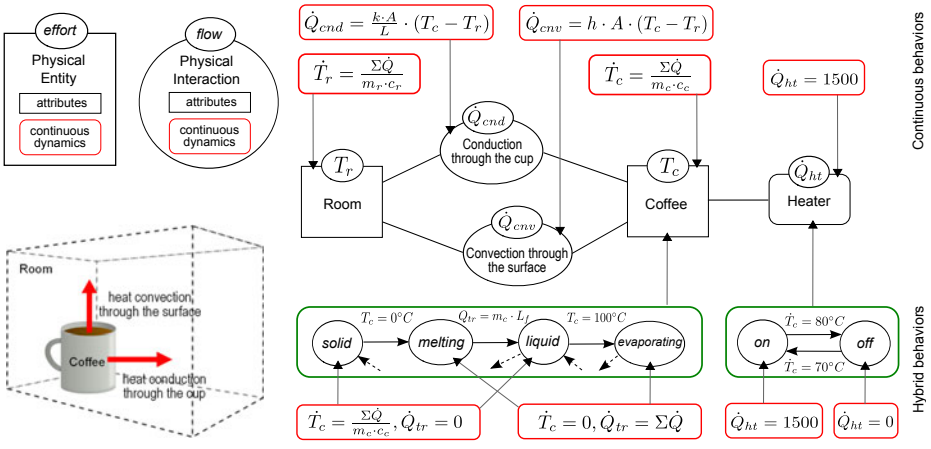


Fig. 1. Physical system components and their interaction in a simple thermal system

Finally, the system may also exhibit discrete transitions, because of phase changes, explicit control, communication, or other factors.

Figure 1 illustrates our modeling methodology on a *thermal* system consisting of a cup of coffee in a room. Heat flows from a hot cup of coffee to the room through both *heat convection* and *heat conduction*, where the flow variable ($\dot{Q}$) denotes the heat flow rate of the physical interaction. The effort variables of the two physical entities (coffee and room) are the temperature T_r of the room and the temperature T_c of the coffee. The values of the other attributes are parameters such as the mass and surface area of the cup. We also have a (flow-source) heater that adds a constant flow of heat to the cup of coffee. Finally, we have discrete behaviors, since the phase of the coffee could change instantaneously from, e.g., liquid to evaporating, and since the heater could be automatically turned off and on to keep the temperature of the coffee between 70 and 80 degrees.

4 Executing Interacting Hybrid Systems

The continuous dynamics of a physical *entity* is typically defined as an ordinary differential equation (ODE), where the time derivative of its effort is a function of both the entity's attribute values *and* the flows of connected interactions. The continuous dynamics of a physical *interaction* is an equation with the flow variable on the left-hand side and an expression possibly referring to the interaction's attributes and the efforts of the connected entities on the right-hand side. This way the direct coupling of the ODEs of physical entities can be avoided.

As we do not require *linear* ODEs, the continuous dynamics of a system is in general not analytically solvable. We therefore use numerical techniques to *approximate* the continuous behaviors by advancing time in small discrete time increments, and approximating the values of the continuous variables at each "visited" point in time. We have adapted the following numerical methods to

our effort/flow framework: the *Euler* [5], the *Runge-Kutta 2nd order* (RK2), and the *Runge-Kutta 4th order* (RK4) methods [7] for fixed time increments. The papers [5,7] explain in detail how we have adapted these numerical methods, and how they have been defined in Real-Time Maude. Furthermore, those papers also show the execution times and the relative errors for the different numerical methods on an example that does have an analytical solution.

5 The HI-Maude Tool

The HI-Maude tool integrates the modeling techniques in Section 3 and the Real-Time Maude implementations of the numerical approximation algorithms Euler, RK2, and RK4 to support the rewriting-logic-based object-oriented formal modeling and simulation, reachability, and LTL model checking analysis of hybrid systems containing interacting physical components. In particular, the HI-Maude tool makes it easy to define the continuous dynamics of the effort and flow variables of single physical entities and physical interactions, respectively. Once the dynamics of the single physical components have been defined, the tool

1. automatically defines the continuous dynamics of the entire systems, and
2. provides the usual Real-Time Maude formal analysis commands, but where the desired built-in approximation algorithm and the desired time increments used by the approximations are additional parameters of the commands.

Furthermore, the tool provides infrastructure to define that instantaneous transitions (modeled as instantaneous rewrite rules) are applied in a timely manner.

HI-Maude is implemented in Maude as an extension of the Real-Time Maude tool, and is available at `http://folk.uio.no/mohamf/HI-Maude/`.

5.1 Representing Continuous Values

Maude provides a built-in data type for the *unbounded* rational numbers, and we first used these rationals for the effort and flow values. However, it quickly became apparent that it is inconvenient to use the rationals, because: (i) it is hard to read large rational numbers; and (ii) the *size* of the rational numbers gets very large (both the numerator and the denominator are large numbers), which slows down the execution. HI-Maude therefore uses the Maude's built-in IEEE-754 floating-point numbers to represent the effort and flow values.

5.2 Modeling

This section explains how hybrid systems can be specified in HI-Maude as a multiset of objects representing physical entities, physical interactions, other flow components, as well as other objects, representing, e.g., controllers. We first show how the physical entities and interactions and their continuous dynamics should be defined, and then discuss instantaneous discrete transitions.

Modeling Physical Entities and Interactions. HI-Maude provides the following built-in classes for specifying physical entities and interactions, as well as flow-source components. Concrete physical entities (interactions, ...) must then be defined as object instances of user-defined subclasses of these built-in classes:

```
class PhysicalEntity | effort : Float .
class PhysicalInteraction | flow : Float, entity1 : Oid, entity2 : Oid .
class FlowSource | flow : Float, entity : Oid .
```

The `effort` attribute denotes the effort value of the entity. The `flow` attribute of a `PhysicalInteraction` denotes the flow between the physical entities given by the `entity1` and the `entity2` attributes. In case of `FlowSource`, the `entity` attribute denotes the name of physical entity which receives the given flow.

For example, to define *thermal* systems such as the simple coffee example in Fig. 1, we can define a new class `ThermalEntity` (with attributes denoting the heat capacity and the mass of the entity), whose objects model thermal physical entities, such as the cup of coffee and the room, as a subclass of `PhysicalEntity`:

```
class ThermalEntity | heatCap : Float, mass : Float .
subclass ThermalEntity < PhysicalEntity .
```

The `effort` attribute of the superclass denotes the temperature of the entity. Likewise, as shown in Fig. 1, the heat flow by *convection* through the surface of the coffee is characterized by the temperatures of the entities as well as of the *area* of the surface (A) and the convection coefficient h:

```
class Convection | area : Float, convCoeff : Float .
subclass Convection < PhysicalInteraction .
```

Thermal *conduction* and *radiation* can be defined similarly (see [5]). Finally, we can define a heater as a constant heat flow source (where this flow is 0 when the `status` of the heater is `off` and is 1500 when the `status` is `on`; we also monitor the temperature of the coffee, but abstract from the details of that sensing):

```
class Heater | status : OnOff, monitoredTemp : Float .
subclass Heater < FlowSource .
```

An initial state (in which the system has not yet computed the first values of the flows) of the coffee system could then be

```
{< coffee : ThermalEntity | effort : 70.0, heatCap : 4.2, mass : 0.4 >
 < room : ThermalEntity | effort : 20.0,  heatCap : 10.5, mass : 80.0 >
 < c-rCond : Conduction | flow : 0.0, entity1 : coffee, entity2 : room, ... >
 < c-rConv : Convection | flow : 0.0, entity1 : coffee, entity2 : room,
                          convCoeff : 0.02, area : 0.05 >
 < heater : Heater | flow : 1500.0, entity : coffee, status : on, monitoredTemp : 70.0 >}
```

Modeling the Continuous Dynamics. HI-Maude provides infrastructure that only requires the user to define the continuous dynamics of single physical components. As indicated in Fig. 1, the time derivative of the *effort* of an entity is

a function of *the sum of the flows* to/from the entity as well as of other attribute values of the entity (e.g., $\dot{T}_r = \frac{\sum \dot{Q}}{m_r \cdot c_r}$). Likewise, the *flow* of an interaction is a function of the (previous) values of the efforts of the connecting entities and other attribute values of the interaction (e.g., $\dot{Q}_{cnv} = h \cdot A \cdot (T_c - T_r)$). The flows of the `FlowSource` components may depend on attribute (and effort) values of both the component and the connecting entity/entities (although in the coffee example this flow is simply defined as $\dot{Q}_{ht} =$ if `status` == `on` then 1500.0 else 0.0).

For each such physical component, the user must define the corresponding of the following functions:

```
op effortDyn : Float Object -> Float .
op flowDyn : Float Float Object -> Float .
op flowSourceDyn : Configuration -> Float .
```

The first argument of `effortDyn` is the sum of the values of the flows to/from the entity; the other argument is the entity object itself. `effortDyn`$(\sum \dot{Q}, \mathit{entity})$ therefore defines the time derivative of the effort variable of the *entity* object. In our coffee example, this is defined in the same way for both the coffee and the room ($\dot{T}_r = \frac{\sum \dot{Q}}{m_r \cdot c_r}$ and $\dot{T}_c = \frac{\sum \dot{Q}}{m_c \cdot c_c}$, for the different $\sum \dot{Q}$s) and hence the user must define `effortDyn` as follows:

```
eq effortDyn(X, < O : ThermalEntity | mass : M, heatCap : C >) = X / (M * C) .
```

We follow here the convention that variables are written with (only) capital letters, and do not show the variable declarations.

To define the flow of an interaction, we define `flowDyn(E1,E2,`*interaction*`)`, where `E`*i* denotes the (previous) value of the effort of the object referred to by the `entity`*i* attribute of the *interaction* object. For example, to define the heat flow through *convection* between the room and the coffee, a HI-Maude user defines

```
eq flowDyn(E1, E2, < O : Convection | convCoeff : CC, area : A >) = CC * A * (E1 - E2) .
```

The function `flowSourceDyn` defines the flow of the flow-source components, and its argument is the entire multiset of objects in the system. In our coffee example, the heat flow from the heater only depends on the state of the heater:

```
eq flowSourceDyn(< O : Heater | status : S > REST) = if S == on then 1500.0 else 0.0 fi.
```

Discrete Transitions. Discrete transitions are modeled as instantaneous rewrite rules. In general, a rule may be applied whenever it is enabled, but nothing forces a rule to be taken in a timely manner. For example, a discrete transition that turns off the heater when it perceives that the temperature of the coffee has reached 80 degrees can be modeled by the following conditional rewrite rule:

```
crl [turnOffHeater] :
    < H : Heater | status : on, monitoredTemp : T >
   =>
    < H : Heater | status : off > if T >= 80.0 .
```

To force the application of this rule as soon as the temperature has reached 80 degrees, the user can define the function `timeCanAdvance` on heater objects. When `timeCanAdvance` of an object is `false`, time advance in a system stops, forcing the application of a suitable instantaneous rule. In the coffee example, the only discrete transitions that must be performed in a timely manner are those turning on and off the heater. Therefore, the user can achieve this by only letting time advance when the heater can stay in a given state:

```
eq timeCanAdvance(< H : Heater | status : S, monitoredTemp : T >)
 = if S == on then T < 80.0 else T > 70.0 fi .
```

5.3 Formal Analysis in HI-Maude

HI-Maude extends Real-Time Maude's analysis commands by allowing the user to select (i) the numerical approximation technique used to approximate the continuous behaviors, and (ii) the time increment used in the approximation.

For example, HI-Maude's hybrid *rewrite* command is used to simulate one behavior of system from a given initial state up to a certain duration and, possibly, up to a certain number of rewrites. Its syntax is:

`(hrew` [`[`*n*`]`] *initState* `in time` $\sim$ *timeLimit* `using` *numMethod* `stepsize` *stepSize* `.)`

where the '`[`*n*`]`' part is optional. The number n denotes the upper bound on the number of rewrite steps to perform; *initState* is the initial state; $\sim$ is either '`<=`' or '`<`'; *timeLimit* denotes an upper bound on the total duration of the rewrite sequence; *numMethod* $\in \{$`euler`, `rk2`, `rk4`$\}$ is the numerical method used to approximate the continuous behaviors; and *stepSize* is the time increment used in the approximation of the continuous behaviors.

Real-Time Maude's timed search command—which searches for states that are matched by a search pattern with a substitution that satisfies an (optional) condition and that can be reached from an initial state within a given time interval—has been extended to the hybrid setting in the same way:

```
(hsearch [[n]] initState =>* searchPattern [such that cond] in time ~ timeLimit
                  using numMethod stepsize stepSize .)
```

where $\sim \in \{$`<`, `<=`, `>`, `>=`$\}$, and *cond* is a condition on the variables appearing in the search pattern. The arrow '`=>!`' is used to search for states which cannot be further rewritten. We can also search without time bounds by writing '`with no time limit`' instead of '`in time` $\sim$ *timeLimit*'.

The following command finds the shortest time needed to reach a state:

```
(hfind earliest init =>* pattern [such that cond] using numMethod stepsize sSize.)
```

Finally, HI-Maude's model checker extends Real-Time Maude's explicit-state time-bounded linear temporal logic model checker in the same way. The time-bounded hybrid model checking command is written with syntax

```
(hmc initState |=t formula in time ~ timeLimit using numMethod stepsize sSize .)
```

where *formula* is an LTL formula and $\sim$ is either '`<`' or '`<=`'. The model checker returns '`true`' if the property holds, and returns a counterexample otherwise.

5.4 Soundness and Completeness of HI-Maude Analyses

There is a trade-off between expressiveness and analytic power for hybrid systems. Model checkers and reachability analysis tools deal with very restricted fragments of hybrid systems, such as initialized rectangular hybrid automata (see [9] for a survey on decidable fragments of hybrid automata), to ensure that key properties are decidable. On the other hand, simulation tools have much more expressive modeling languages, but only provide simulation capabilities.

HI-Maude is as expressive as simulation tools, yet provides reachability and LTL model checking analysis in addition to simulation. The price to pay is that reachability and satisfaction of LTL properties are in general no longer decidable.

HI-Maude only analyzes those behaviors that are possible with the selected time increment and numerical method used to approximate the continuous behaviors. Therefore, the results of search and model checking in HI-Maude may not be correct. If a counterexample is found in LTL model checking, or a desired state is found in a search, these are indeed valid counterexamples *up to the approximation errors due to the use of numerical approximations and round-off errors due to the use of floating-point numbers*. However, since only a *subset* of all possible behaviors are analyzed, the fact that a state is not found during a search or that LTL model checking returns `true` does not necessarily imply that the state cannot be reached or that the LTL property holds.

6 The Real-Time Maude Semantics of HI-Maude

This section gives an overview of the Real-Time Maude semantics of HI-Maude.

A HI-Maude *hybrid module* automatically imports a library of numerical approximations methods, the built-in classes described above, other functions, etc. For any HI-Maude command, the tool adds a *system manager* object

```
< sysMan : SysMan | numMethod : numerical method, stepSize : step size >
```

to the initial state and transforms a HI-Maude analysis command to the corresponding Real-Time Maude command. For example, the hybrid search command

```
(hsearch {init} =>* pattern in time <= 100 using rk4 stepsize 2 .)
```

is executed by executing the Real-Time Maude timed search command

```
(tsearch {init < sysMan : SysMan | numMethod : rk4, stepSize : 2 >} =>* pattern
         in time <= 100 .)
```

The following tick rule is added to any hybrid module. It advances time by the time increment given in the HI-Maude command, and updates the continuous effort and flow variables according to the numerical method used:

```
crl [tick] :
    {< SM : SysMan | stepSize : SS > REST}
   =>
    {computeEF(< SM : SysMan | > REST)}  in time SS if timeCanAdvance(REST) .
```

The function `timeCanAdvance` is used to allow the user to specify that time cannot advance in certain states, to ensure timeliness of discrete transitions. It distributes over each object in the state, and an '`owise`' equation ensures that time advance is not impeded by those objects for which the user has *not* defined a `timeCanAdvance`-equation. The function `computeEF` computes the new values of the effort and flow values. We refer to the executable specification of HI-Maude for its precise definition, and to [5,7] for an explanation of how to implement the numerical approximation algorithms in Real-Time Maude.

7 Case Study: The Human Thermoregulatory System

We have also used HI-Maude and its effort/flow-based modeling methodology on a more ambitious case study modeling and analyzing the human thermoregulatory system. Since the model is fairly large, we can only present a brief overview of our modeling and analysis efforts in this paper, and refer to our longer technical report at `http://folk.uio.no/mohamf/HI-Maude/` for a thorough exposition.

7.1 The Human Thermoregulatory System

Human thermoregulation regulates the heat production within the body and the heat exchange between the body and the environment in order to maintain an internal body temperature of around $37°C$. Heat is produced within the body by the metabolism process, while the interaction with the environment causes heat loss or gain through physical processes such as radiation, evaporation, and convection [11]. *Hyperthermia* and *hypothermia* occur when the body temperature increases, respectively decreases, significantly beyond normal [16].

The thermoregulatory system is controlled by the *hypothalamus*, which enables the following mechanisms to support heat loss from the body when the body temperature is increasing above normal: increasing the diameter of blood vessels to let more blood flow underneath the skin (*vasodilation*), which promotes heat loss by radiation, convection, and conduction; and increasing sweat production, which promotes heat loss by evaporation. When the body temperature is decreasing, the hypothalamus enables the following mechanisms to reduce heat loss and increase heat production: decreasing the diameter of blood vessels to let less blood flow underneath the skin (*vasoconstriction*), and stimulating the skeletal muscles to cause shivering, which increases heat production.

7.2 Effort/Flow Modeling of the Human Thermoregulatory System

To reason about the thermoregulatory system, we can think of a person in a room as a *thermal* system, where the body core, the body skin, and the room are *thermal entities*, and where the heat flow between the body core and the skin and between the skin and the room are *thermal interactions*, as shown in Fig. 2. Heat flows between the body core and the skin through blood vessels, and between the skin and the room through radiation and convection. The *effort* variable of

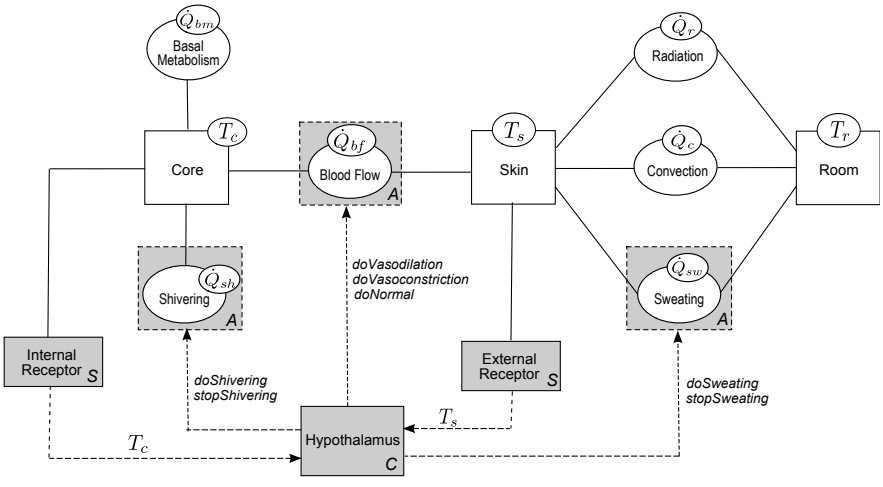


Fig. 2. Effort/flow model of the human thermoregulatory system

a thermal entity denotes its *temperature*, and the *flow* variable of the thermal interaction is the *heat flow rate*. The heat production inside the body through basal metabolism and shivering are represented as *flow-source* components. The heat loss from the skin by sweating is represented as a physical interaction, since the consequence of this process is heat gain in the room.

Modeling Human Thermal Entities. A thermal entity is defined by extending the built-in class `PhysicalEntity` with the entity's heat capacity and mass:

```
class ThermalEntity | mass : Float, heatCap : Float .
subclass ThermalEntity < PhysicalEntity .
```

The body core, the skin, and the room share the same thermal dynamics, where the temperature change ΔT is derived from $\Delta Q = m \cdot c \cdot \Delta T$, where ΔQ is the amount of heat transferred per time unit. We therefore define `effortDyn`, specifying the dynamics of an entity's effort value as in the coffee example:

```
eq effortDyn(SF, < TE : ThermalEntity | mass : MASS, heatCap : HC >) = SF / (MASS * HC) .
```

where the sum `SF` of the heat flows of the entity is computed by the tool.

To model the temperature-related states of the body core (*normal*, *moderate* and *severe hyperthermia*, *moderate* and *severe hypothermia*, and *dead*), we define a new subclass of `ThermalEntity` with a new attribute `coreState`:

```
sort CoreStateType .
ops normal modHyperthermia sevHyperthermia modHypothermia sevHypothermia
    dead : -> CoreStateType [ctor] .
class CoreHumanBody | coreState : CoreStateType .
subclass CoreHumanBody < ThermalEntity .
```

For example, the following rewrite rule models the body core state change from *normal* to *moderate hyperthermia* when the temperature exceeds $38°C$:

```
crl [normal-to-modhyperthermia] :
    < CORE : CoreHumanBody | effort : TEMP, coreState : normal >
   =>
    < CORE : CoreHumanBody | coreState : modHyperthermia >  if TEMP > 38.0 .
```

To ensure that the above rules are applied as soon as they are enabled, we use the built-in `timeCanAdvance` function to define, for each core state, when time can advance *without* a rule having to be taken, e.g.:

```
eq timeCanAdvance(< CORE : CoreHumanBody | effort : TEMP, coreState : normal >)
 = TEMP > moderateHypothermiaPoint and TEMP <= moderateHyperthermiaPoint .
```

Modeling Human Thermal Interactions. The thermal interactions are *radiation*, *convection*, and *blood flow*. We show how to define radiation, whose dynamics represents the rate of heat radiation given by $\dot{Q} = \varepsilon \cdot \sigma \cdot A \cdot (T_1^4 - T_2^4)$, where ε is the emissivity of the surface, σ is the Stefan-Boltzmann constant, and A is the surface area through which radiation takes place. The class defining radiation interactions therefore adds attributes for emissivity and area to the built-in class `PhysicalInteraction`, and its continuous dynamics is specified using the built-in function `flowDyn`:

```
class Radiation | area : Float, emissiv : Float .
subclass Radiation < PhysicalInteraction .
eq flowDyn(TEMP1, TEMP2, < TI : Radiation | area : AREA, emissiv : EMISSIV >)
 = EMISSIV * stefBoltzConst * AREA * ((TEMP1 ^ 4.0) - (TEMP2 ^ 4.0)) .
```

Modeling the heat flow between the body core and the skin through the blood flow is more challenging, since the heat flow rate depends on the blood flow rate, which again depends on the diameter of the blood vessels, which can be changed by vasodilation and vasoconstriction (see our technical report for details).

Modeling the Hypothalamus. In our model, the hypothalamus is modeled by an object that senses the core and skin temperatures, and manages the activation and deactivation of the shivering, the sweating, and the blood flow.

7.3 Formal Analysis

We model the human body as a vertical tube with height $1.7m$, diameter $0.3m$, body mass $63.5kg$, and body heat capacity $3.47kJ/kg°C$. The person is resting, with basal metabolic rate $0.08kW$. The room is $5m$ long, $5m$ wide, and $3m$ high, and is filled with air with heat capacity $1.005kJ/kg°C$ and density $1.2kg/m^3$. Moderate hyperthermia starts at the core temperature $38°C$, the severe hyperthermia at $41°C$, and death occurs when the core temperature exceeds $45°C$.

We consider three persons: a healthy person, a person with high fever, and a person with brain damage in the hypothalamus. We analyze the effect of these

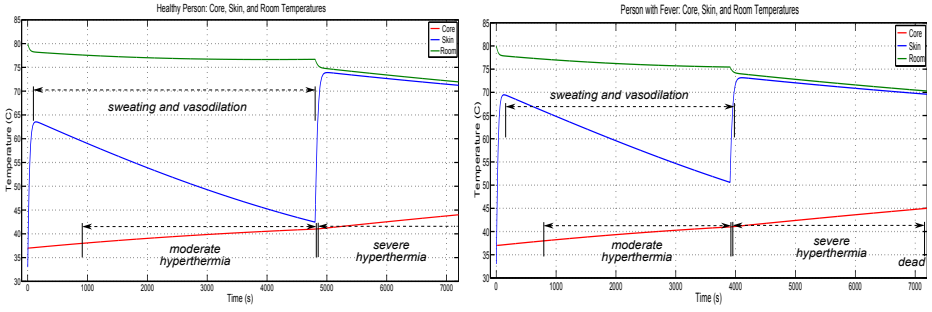


Fig. 3. The simulation results for the healthy and fever persons

conditions to survive an extreme condition, and set the initial temperature of the body, the skin, and the room at $37°C$, $33°C$, and $80°C$, respectively. The experiments are performed on an Intel Pentium 4 CPU 3.00GHz with 3GB RAM.

The following initial state `cs1` defines a system with a healthy person:

```
op cs1 : -> GlobalSystem
eq cs1 =
   {< core : CoreHumanBody | effort : 37.0, mass : 0.85 * 63.5, heatCap : 0.85 * 3.47, coreState : normal >
    < skin : SkinHumanBody | effort : 33.0, mass : 0.15 * 63.5, heatCap : 0.85 * 3.47 >
    < room : ThermalEntity | effort : 80.0, mass : 90.0, heatCap : 1.005 >
    < bloodf : BloodFlow | flow : 0.0, entity1 : core, entity2 : skin, area : bodyArea,
                           skinBloodFlowRate : 0.0315, bloodHeatCap : 3.85, thermalCond : 0.00021,
                           controller : hypothal, state : normal, ... >
    < rad : Radiation | flow : 0.0, entity1 : skin, entity2 : room, area : bodyArea, emissiv : 0.97 >
    < convec : Convection | flow : 0.0, entity1 : skin, entity2 : room, area : bodyArea, coeff : 0.0026 >
    < metabol : BasalMetabol | entity : core, flow : bodyMet, status : on >
    < shivering : Shivering | entity : core, flow : 0.0, status : off, controller : hypothal >
    < sweating : Sweating | entity1 : skin, entity2 : room, flow : 0.0, status : off, ... >
    < hypothal : Hypothalamus | status : idle, hotSetPointCore : 37.5, coldSetPointCore : 36.5,
                                hotSetPointSkin : 34.5, coldSetPointSkin : 12.0, ... >
    < intrecept : InternalReceptor | entity : core, controller : hypothal, ... >
    < extrecept : ExternalReceptor | entity : skin, controller : hypothal, ... >} .
```

The state `cs2` adds an object which records key values in each step of the simulatio to `cs1`. We use hybrid rewriting to simulate the system for two hours:

```
Maude> (hrew cs2 in time 7200 using euler stepsize 1.0 .)
```

Fig. 3 shows the simulation results—the temperature of the body core, the skin, and the room temperatures as time advances—for a healthy person and for a person with fever. For the person with fever the sweating starts later since the shifting of the set point in the hypothalamus causing it sense the danger late.

We next use the *find earliest* command to analyze how long a person can stay in the sauna before he dies:

```
Maude> (hfind earliest
          cs1 =>* {C:Configuration  < personCore : CoreHumanBody | coreState : dead >}
            using euler stepsize 1.0 .)
```

The following table shows how it long takes for each person to reach the various stages of discomfort (as well as the CPU time of the command execution):

	moderate hyperthermia	severe hyperthermia	death
healthy	912 sec CPU: 14 sec	4802 sec CPU: 87 sec	8051 sec CPU: 251 sec
fever	803 sec CPU: 13 sec	3906 sec CPU: 90 sec	7197 sec CPU: 356 sec
brain damage	703 sec CPU: 2 sec	2880 sec CPU: 17 sec	6214 sec CPU: 83 sec

Many complex properties cannot be formulated as reachability problems, but may instead be defined as linear temporal logic (LTL) model checking problems. The following command checks whether a moderately hyperthermic person will sweat and experience vasodilation until (s)he becomes severely hyperthermic:

```
Maude> (hmc cs1 |=t [] (modHyper -> ((sweating /\ vasodilation-active) W sevHyper))
                       in time <= 7200 using euler stepsize 1.0 .)
```

where `modHyper`, `sweating`, `vasodilation-active`, and `sevHyper` are atomic propositions. For each person, the model checking returned the expected result (in 110 seconds of CPU time).

8 Concluding Remarks

We have introduced the HI-Maude tool that supports the formal modeling, simulation, and model checking of complex interacting hybrid systems in rewriting logic. The tool supports the compositional modeling of a complex system based our adaptation of the effort/flow approach developed in [5], and integrates the numerical approximation methods formalized in Real-Time Maude in [5,7]. We have illustrated the use of the tool on the challenging human thermoregulatory system that features a set of interacting physical subsystems with complex continuous dynamics. Being based on rewriting logic, HI-Maude provides an intuitive yet expressive modeling language with support for concurrent objects, user-definable data types, different communication models, etc.

As usual much work remains. We should integrate techniques that dynamically adjust the step size used in the approximations to (i) make the analysis *more precise* by making the time step smaller when needed either to come close to a time instant when a discrete transition must be taken or to maintain a desired precision of the approximation, and (ii) make the analysis *more efficient* by increasing the step size whenever possible. In particular, adaptive step sizes gives the user to possibility to define his/her own *error tolerance* to balance between desired precision and computational efficiency. These features have been formalized in Real-Time Maude [6,7] and must be integrated into HI-Maude.

Acknowledgments. This work was supported by The Research Council of Norway (RCN) through the Rhytm project, and by RCN and The German Academic Exchange Service through the DAADppp project HySmart.

References

1. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., Theobald, M.: Verification of hybrid systems based on counterexample-guided abstraction refinement. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 192–207. Springer, Heidelberg (2003)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Bevilacqua, V., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
3. Dang, T.: Verification and Synthesis of Hybrid Systems. Ph.D. thesis, INPG (2000)
4. Esposito, J., Kumar, V., Pappas, G.: Accurate event detection for simulating hybrid systems. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 204–217. Springer, Heidelberg (2001)
5. Fadlisyah, M., Ábrahám, E., Lepri, D., Ölveczky, P.C.: A rewriting-logic-based technique for modeling thermal systems. In: Proc. RTRTS 2010. Electronic Proceedings in Theoretical Computer Science, vol. 36 (2010)
6. Fadlisyah, M., Ábrahám, E., Ölveczky, P.C.: Adaptive-step-size numerical methods in rewriting-logic-based formal analysis of interacting hybrid systems. In: Proc. TTSS 2010 (2010), to appear in ENTCS
7. Fadlisyah, M., Ölveczky, P.C., Ábrahám, E.: Formal modeling and analysis of hybrid systems in rewriting logic using higher order numerical methods and discrete-event detection. In: Proc. CSSE 2011. IEEE, Los Alamitos (2011)
8. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
9. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. 57(1), 94–124 (1998)
10. Henzinger, T.A., Horowitz, B., Majumdar, R., Wong-toi, H.: Beyond HyTech: Hybrid systems analysis using interval numerical methods. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 130–144. Springer, Heidelberg (2000)
11. Herman, I.: Physics of the Human Body. Springer, Heidelberg (2007)
12. Lee, E., Zheng, H.: HyVisual: A hybrid system modeling framework based on Ptolemy II. In: IFAC Conference on Analysis and Design of Hybrid Systems (2006)
13. Ölveczky, P.C., Meseguer, J.: Abstraction and completeness for Real-Time Maude. ENTCS 176(4), 5–27 (2007)
14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
15. Ölveczky, P.C., Bevilacqua, V.: The Real-Time Maude tool. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 332–336. Springer, Heidelberg (2008)
16. Plantadosi, C.: The Biology of Human Survival: Life and Death in Extreme Environments. Oxford University Press, Oxford (2003)
17. Simulink home page, http://www.mathworks.com/products/simulink
18. Wellstead, P.E.: Introduction to physical system modelling. Academic Press, London (1979)

Towards an Agent-Based Methodology for Developing Agro-Ecosystem Simulations

Jorge Corral and Daniel Calegari

Instituto de Computación, Facultad de Ingeniería, Universidad de la República,
J. Herrera y Reissig 565, Montevideo, Uruguay
{corral,dcalegar}@fing.edu.uy

Abstract. Agro-ecosystems are ecological systems subject to human interaction whose simulation is of interest to several disciplines (e.g. agronomy, ecology and sociology). The agent-based modeling approach appears as a suitable tool for modeling this kind of complex system, along with a corresponding agent-oriented software engineering (AOSE) methodology for the construction of the simulation. Nevertheless, existing AOSE methodologies are general-purpose, have not yet accomplished widespread use, and clear examples of applications to agro-ecosystems are hard to find. This article sets the ground for an AOSE methodology devised specifically for developing agro-ecosystem simulations. The methodology framework is based upon other general-purpose AOSE methodologies, and it relies on the Unified Modeling Language for an easy uptake from interdisciplinary teams. As a first proof of concept, it is applied to a real case study: the evolution of the strategies followed by cattle producers of the basalt-region of north Uruguay against severe draughts.

Keywords: Agro-Ecosystem, Agent-Based Modeling, Simulation, Agent-Oriented Software Engineering, Unified Modeling Language.

1 Introduction

Many of the current challenges and opportunities (e.g. globalization, sustainability, epidemics or climate change) can be seen as complex systems [13]. Understanding the components, behavior and interactions in these systems is the first step to whatever analysis is needed about them. Models, as simplifications of certain reality or problems, are fundamental tools to this aim. Moreover, the possibility of direct experimentation over these systems is rare if not impossible, so the need for simulation becomes imperative. Even though not at global scale, agro-ecosystems are complex systems [5]. An agro-ecosystem is the human manipulation and alteration of ecosystems for the purpose of establishing agricultural production [8]. In this context, modeling and simulation allow for understanding the system as well as for exploring future scenarios.

Several approaches can be used for modeling an agro-ecosystem, like system dynamics or mathematical approaches. In particular, the agent-based modeling (ABM) approach appears as a specially suitable tool for this aim [13] since it enables the simulation of heterogeneous populations of interacting individuals (called agents) within an environment, along with passive objects (a.k.a. resources) and including

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 431–446, 2011.

human decision-making, all of these in a constructive way and using representations and metaphors that are meaningful for the stakeholders. On the other hand, the other approaches use more formal representations (e.g. equations) that fall short when trying to capture human and natural diversity; they do not appear as intuitive for non-technical stakeholders (so they are hard to validate); they do not scale well to medium-size problems because their mathematical complexity becomes overwhelming; and they do not explicitly model the environment.

Simulating generally means the development of a software system representing the ABM, which requires the use of some agent-oriented software engineering (AOSE) methodology.

Several AOSE methodologies are currently available for guiding a programmer in order to develop software following the ABM approach [9]. However, they are general-purpose agent-oriented methodologies, not focused on simulation, so we are faced with a trade-off between using an already existing one and not leveraging the specificities of agro-ecosystems, or to follow an ad-hoc methodology that pays detail to those features. Up to our knowledge, there are no specific AOSE methodologies for this purpose. Nevertheless, there are some related work [12] that address the simulation of agro-ecosystems using an ABM approach, but without a methodological framework. We have used this proposal in several projects [3, 14] from which an ad-hoc methodology has emerged. In particular, this ad-hoc methodology addresses an important issue which is the need to facilitate the communication between farmers, agronomists, and programmers.

In this article, we propose an AOSE methodological framework for simulating agro-ecosystems that is based upon other general-purpose AOSE methodologies [9], in the already-mentioned proposal [12], and in our own experience on developing agro-ecosystem simulations. It also relies on the Unified Modeling Language (UML, [16]) with the purpose of an easy uptake from both, the interdisciplinary team which models the agro-ecosystem, and the programmers who will develop the simulation. The methodological framework is not a methodology itself. It focuses on identifying relatively general steps and artifacts (produced by the steps) that could be ensemble for developing such a simulation, whilst other methodological aspects, as defining a strict order among steps and identifying roles for that steps, are not yet considered.

The paper is structured as follows. In Section 2 we briefly present the background of agent-based simulation of agro-ecosystems. In Section 3 we describe the methodological framework for the development of agro-ecosystem simulations, whilst in Section 4 we describe how agro-ecosystem features are supported in the proposed methodological framework. Then, in Section 5 we present a case study which stands as a proof of concepts. Finally, in Section 6 we present the conclusions and an outline for further work.

2 Agent-Based Simulations of Agro-Ecosystems

Agro-ecosystems result from the interplay between endogenous biological and environmental features of the agricultural fields and exogenous social and economic factors, and it is delimited by arbitrarily chosen boundaries, as schematized in Figure 1. Concerning the resources commonly found in agro-ecosystems, Norman [15] suggests

the following classification: **1) Natural resources**: the given elements of land, water, climate and natural vegetation that are exploited by the farmer; **2) Human resources**: the people who live and work within the farm and use its resources for agricultural production; **3) Capital resources**: the goods and services created, purchased, or borrowed by the people associated with the farm to facilitate their exploitation of natural resources for agricultural production; and **4) Production resources**: the agricultural output of the farm such as crops and livestock.

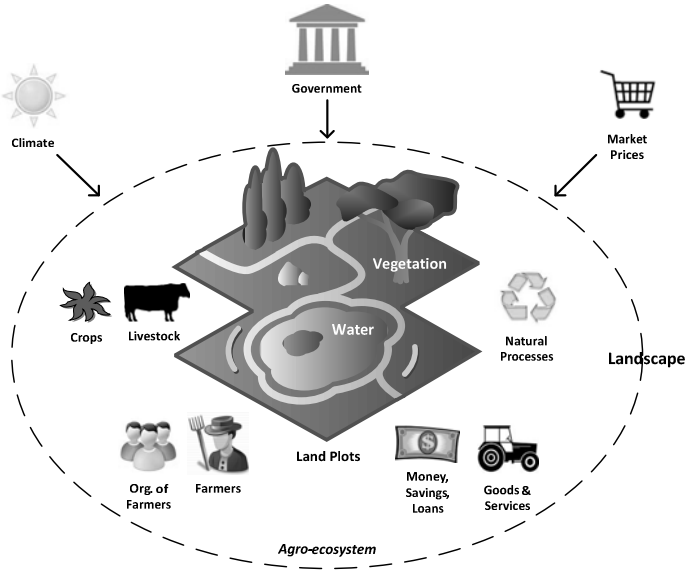


Fig. 1. Schematic representation of relevant agro-ecosystem features

Any ecosystem can be considered as a case of complex adaptive system [5]: systems composed of a large number of interacting elements such that some behavior could not be anticipated from the knowledge of the parts of the system alone (this process is called "emergent"), and there is no external controller or planner engineering the appearance of the emergent features; they appear spontaneously (as "self-organized"). The interactions between the social and natural subsystems inside an agro-ecosystem, such as farmers' practices affecting natural resources, are examples of interactions at a small scale which may produce feedbacks affecting in turn the decision-making of social actors. Repeating these interactions on a daily basis can produce emergent properties and new organizations across the agro-ecosystem in the long-term. Since these coupled human-natural systems cannot be manipulated and tested as other systems, due to scale and resource difficulties, the possibility of simulating them is crucial. One of the main objectives of simulating this kind of systems is to prospect scenarios were the interest is not in the 'fortune-teller' features of the simulation but on discovering possible outcomes under certain conditions and exploring the consequences of manipulating certain system.

In this context, ABM approach appears as a suitable tool for the simulation of agro-ecosystems [13]. The ABM approach enables the simulation of heterogeneous populations of interacting individuals (called agents) within an environment, which can also contain passive objects (a.k.a. resources). A well-known definition of an agent that supports the computer science focus presented in this article is proposed by Wooldridge [19]: "*An agent is a computer system that is capable of independent action on behalf of its user or owner, figuring out what needs to be done to satisfy design objectives, rather than constantly being told [what to do]*". In order to let agents react (or take the initiative) according to changes in the environment, agents must perceive their environment and have some way to act upon it, after reasoning what to do. This leads agent to the so called Perceive/Reason/Act cycle. Since there are multiple ways to achieve this cycle, there are accordingly different agent architectures that represent and implement this concept in different ways and to different degrees, from reactive architectures where the agent has no previous knowledge and simply reacts based on a set of rules, to deliberative ones that manage explicit representations of beliefs, desires, and intentions to make decisions [17].

There are numerous arguments that support that the ABM approach is suitable for modeling agro-ecosystems, like those presented in [13], but nevertheless it is worth analyzing how the ABM approach specifically addresses the following issues, which characterize an agro-ecosystem: **a) Emergence**: ABMs allow to define the low-level behavior of each individual agent in order to let them interact to see whether some emergent property arises or not, and if it does, under which circumstances; **b) Self-Organization**: ABMs do not have any kind of central intelligence that governs all agents. On the contrary, the sole interaction among agents along with their feedbacks is what ultimately 'controls' the system; **c) Human-Natural Systems**: ABMs allow considering together both, social organizations with their human decision-making and social communications, and biophysical processes and natural resources. This conjunction of subsystems enables ABMs to explore the interrelations between them, allowing analyzing the consequences of one over the other; and **d) Spatially Explicit**: the feature of ABMs of being able to spatially represent an agent or a resource is of particular interest when communications and interactions among neighbors is a key issue. This feature is of special interest in the case of agro-ecosystems.

The simulation can be performed by a multi-agent system (MAS). According to Wooldridge [20] "*A Multi-Agent System consists of a number of agents which interact with one-another. […] To successfully interact, they will require the ability to cooperate, coordinate, and negotiate with each other.*" In MAS there is no central control and all information and control is distributed among the various agents.

In order to analyze, design, implement and every other step involved in developing such systems, certain agent-oriented software engineering (AOSE) methodology must be used. There are several methodologies for developing agent-oriented software systems, most of them presented in [9], e.g. GAIA, Tropos, MAS-Common CADS, Prometheus, Passi, Adelfe, Mase, Rap, Message, and Ingenias. From these ten AOSE methodologies, Tran and Low [18] summarize 16 general steps, going from the identification of systems functionality to the deployment of agent instances. It is reasonable to think that any AOSE methodology will be strongly related with these 16 general AOSE steps. Nevertheless, these steps are not tailored for MAS *simulations*, neither for developing *agro-ecosystem* simulations. Up to our knowledge there are no

methodologies for these purposes. However, Le Page and Bommel [12] present in a very general way how some UML diagrams could be used to implement a CORMAS [2] simulation, without providing further methodological aspects on how a software methodology could be followed (e.g. steps and artifacts). There is also no mapping given on how the most relevant features present in agro-ecosystems could be successfully represented using those diagrams, and no details on how to model the simulation aspects (such as initial configuration, input and output parameters or visualization).

These 16 general AOSE steps and the proposal of Le Page and Bommel [12] are indeed the drivers of the methodological framework which is proposed in the next section.

3 The Methodological Framework

In what follows we propose an AOSE methodological framework for modeling and simulating agro-ecosystems. We based this methodological framework upon other general-purpose AOSE methodologies, in the proposal of Le Page and Bommel, and in our own experience on developing agro-ecosystem simulations. Moreover, we assume that the simulation will be finally developed within an existing simulation framework software package (like CORMAS). However, we will not make any assumption about what simulation paradigm it uses, i.e. continuous, time-step or discrete-events. This will be indeed matter of discussion.

We present next the methodological framework by identifying relatively general steps and artifacts (produced by the steps) that could be ensemble for modeling and simulating agro-ecosystems. Further methodological aspects needed in the whole methodology are not yet considered.

Each one of the 16 general AOSE steps summarized by Tran and Low [18] has been analyzed in order to determine its convenience for the methodological framework, and a thorough study has been conducted for each step [4]. Table 1 summarizes the results.

Since the steps are not tailored for dealing with a simulation, they do not specifically cover simulation capabilities. For this reason, three more steps were added:

- The **Simulation Configuration** step aims at defining those fundamental elements that will enable the simulation to be run (Initial Configuration, Time Definition, Task Scheduling, Input Parameters, Output and Visualization).
- The **Implementation** step involves coding the agent-based simulation using an already existing simulation platform. The programmer should have enough elements (as well as enough understanding of the problem) to start coding. In this sense, the programmer should be based on the information provided by the artifacts developed in the previous steps.
- The **Simulation Run & Sensitivity Analysis** step involves gathering all the necessary real-world data (including possible historical data) for allowing the simulation to be run and to perform explorations about how the output of the simulation is affected when certain elements are changed.

Table 1. Summary of the steps selected from the 16 general AOSE steps of [18]

	AOSE Step	Selected?
Problem domain analysis steps	Identify system functionality	Yes: Identify System Purpose
	Identify roles	Yes: Identify Roles and Agent Types
	Identify agent classes	
	Model domain concepts	Yes: Model Domain Concepts
Agent int. design steps	Specify acquaintances between agent classes	No, either included in *Identify Roles and Agent Types* or in *Model Domain Conceptualization*
	Define interaction protocols	Yes: Define Agent Interaction
	Content of ex-changed messages	
Agent internal design steps	Specify agent architecture	Yes: Agent Architecture and Design
	Define agent mental attitudes	No, included in step *Agent Architecture and Design*
	Define agent behavioural interface	No
Overall system design steps	Specify system architecture	No
	Specify org. structure/inter-agent social relationships	No
	Model MAS environment	Yes: Model Env. & Resources
	Specify agent-env. interaction mechanism	No
	Instantiate agent classes	No
	Specify agent deployment	No

As we stated before, we rely on a standard graphical modeling language in order to support the development process by facilitating the interchange between domain experts and programmers. For now we use plain UML [16]. However, the methodology will be best supported by a more specific language for representing software systems based on software agent concepts, e.g. AML [1] and those diagrams promoted by the Foundation for Intelligent Physical Agents (FIPA, [6]). Since these languages are not tailored for agro-ecosystems nor for simulations, it must also be further adapted to our methodological framework. Table 2 summarizes the steps of the proposed methodological framework, including its aim and artifacts.

Regarding the proposal of Le Page and Bommel [12], our methodological framework shares several aspects with it, like the use of UML and particularly the use of Class Diagrams for representing structural aspects and Activity, Sequence and State-Transition Diagrams for representing behavioral aspects.

Nevertheless, our methodological framework adds numerous elements, like defining concrete and easy-to-follow steps and artifacts for developing such simulations, separating the representation of agents, resources and environment, considering simulation features like time, initial configuration, scheduling, visualization and input/output parameters, and finally, by especially tackling those features that characterize an agro-ecosystem, as Table 3 shows.

Table 2. Summary of proposed steps that compose the methodological framework

Prop. Step	Aim of the Step	Artifacts
Id. System Purpose	Define the purpose of the simulation, including its objective and questions to be answered.	Text Document including the purpose of the simulation, an overview of the context in which it will be developed, including why it will be developed and what will be expected from it.
Model Domain Concepts	Depict the structure of the problem, including entities and relationships.	UML Class Diagram for modeling main concepts, including those in *Identify Roles & Agent Types* and *Model Environment & Resources*.
Id. Roles & Agent Types	Identify agent types and roles, especially agent behavior.	Text Documents for role's identification and description and for identifying agent types and their relation to roles; UML Activity Diagrams for agent type's behavior specification.
Define Agent Interaction	Determine when, how and what the different agents will communicate.	UML Sequence Diagrams for modeling interactions.
Agent Architecture and Design	Define internal agent design (structure and behavior) in order to fulfill its perceive/reason/act cycle, within a simulation platform.	UML Class Diagram for designing the internal structure of each agent type, as well as any other UML Structure Diagram (like Package and Component Diagram); and UML Communication Diagrams for designing the internal behavior of each agent type, as well as other UML Behavioral Diagrams (like Activity and State-Transition Diagrams).
Model Env. & Resources	Determine behavioral aspects (evolution) of resources and environment, and completing structural aspects.	Text Document for complementing other diagrams; UML Structure Diagrams for further modeling structural aspects of the environment and resources; and UML Behavioral Diagrams for modeling functional (behavioral) aspects of the environment and resources.
Simulation Conf.	Define those fundamental elements that will enable a simulation to be run.	Text Documents for documenting the configuration; UML Object Diagram for the initial configuration; and UML Sequence Diagram for tasks scheduling.
Implem.	Codify the simulation.	Code.
Sim. Run & Sensit. Anal.	Answer the simulation's objectives and questions.	Text Documents with graphics and statistics, and the conclusions.

Table 3. Representation of agro-ecosystem's features within the methodological framework

Feature	Representation
Land Plots	Plots can be thought of as land components of the environment, giving more flexibility by decomposing the land in several elements.
Water	As a special kind of land plot or as a resource within a land plot.
Climate	As a parameters of the simulation.
Vegetation	As an attribute of the land plot or a resource within a land plot.
Farmers' Types	As agent types. Since the behavior of a farmer can evolve over time, different agent roles can be defined.
Organization of Farmers	As a new agent type that is related to its members (which are other agent types). If belonging to an organization implies certain behavior in its members, then new roles can be defined for them and agents should be able to change their behavior when playing this new role.
Goods & Services	These are generally of two kinds: those which are used as inputs for production and those which are used as family consumption. The former has to do with resources (and in some cases represented as an attribute of a farmer) while the latter with farmers' livelihood, needs and expectations (considered within the behavior of farmers).
Money, Savings & Loans	The concepts here involved may be divided in two: the activities that led to an increase/decrease (e.g. buying or selling) and the quantities (e.g. current amount of money the farmer has or how much debt he/she owes). The former can be modeled in the behavior of the farmer, and the latter as an attribute of it.
Crops & Livestock	As resources which can either be associated to a certain land plot (as in the case of crops) or not (livestock). If a resource presents certain dynamics, these may be expressed, much like the behavior of agents, but applied to resources.
Natural Processes	Since natural processes are not resources themselves, but are closely related to them, it is useful to conceive both at the same time within the environment. As the Crops & Livestock feature, resources may present certain dynamics to be run during the simulation, so it should be clearly defined whether these dynamics actually represent natural processes or if they are only concerned with the resource itself. This implies that natural processes may be explicitly modeled as another feature of the agro-ecosystem, or implicitly considered when modeling resources.
Landscape	As an aggregation of land plots. The initial landscape can be determined by the modeler in the initial configuration, e.g. by determining how many plots are used for agriculture, and may also be instantiated by a GIS map.
Market Prices and Evolution	Since it is generally the case that the price over which production resources are sold is not controlled nor determined by the system under study, prices are considered as an external input to the simulation, and therefore it is considered as an input parameter.
Government Policies	This feature is not directly modeled into the simulation since the government is outside the boundaries of the system. The interest is to compare the evolution of the simulation with and without the introduction of certain government policies. This requires modifying the simulation in order to take them into account.

4 Case Study

The aim of this section is to give an overview of how the methodological framework was successfully applied to a real-world case study, serving as a first on-field validation. Details on the exact application of the methodological framework to this case study, along with all detailed artifacts omitted here for space reasons, can be found in [4].

As discussed, the steps of the methodological framework does not need to be followed in the order in which they were presented. Nevertheless, this section shows a specific and simple order in which the steps were followed for the case study along with which artifacts were considered for each step. This is presented in Figure 2. However, not all artifacts need to be used for successfully implementing this particular case study.

The case study presented in this section is a simplified version of the simulation developed for a research project entitled "*Development, validation and evaluation of a modeling and simulation participatory methodology that contributes in the understanding and communication of the draught phenomena, and improves the adaptation capacity of livestock farmers in the basalt*" [7] (basalt soils are much more affected by draught than other types of soils). This was a two-year project financed by Instituto Nacional de Investigación Agropecuaria [10] (governmental institution that develops and fosters agricultural research nation-wide) and executed by Instituto Plan Agropecuario [11] (institution focused on agricultural extension mainly to small and mid-sized livestock farmers).

The research project was motivated due to severe draughts that affected the region (North Uruguay) in the last century, namely in 1916/17, 1942/43, 1964/65, 1988/89, 2005/06, among others.

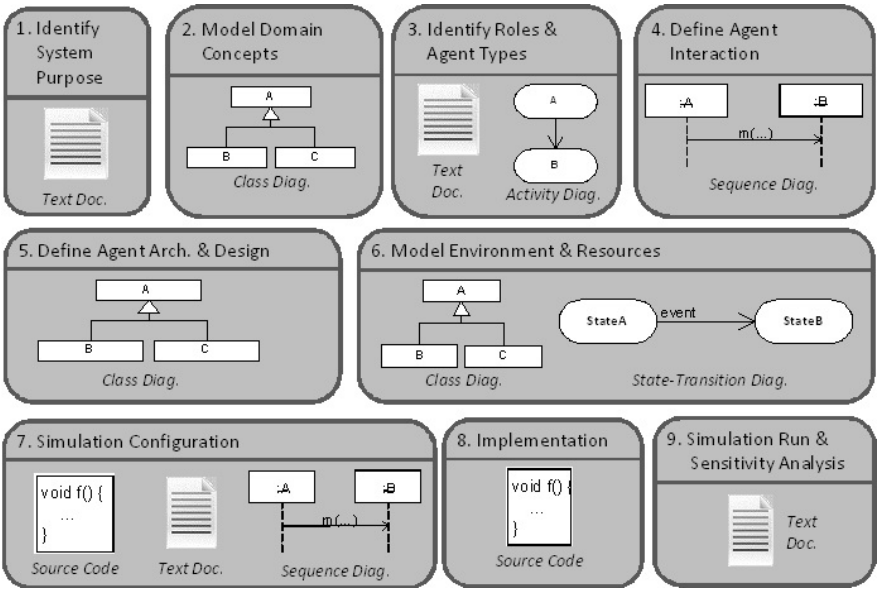


Fig. 2. Steps and artifacts as followed for this case study

The severity of these jeopardizes farm sustainability in all of its three dimensions: economically (because of the loss of income and competitiveness), ecologically (because of cattle mortality and possible loss of grass variety), and socially (because of bankruptcy, emigration to cities and even suicides).

Even though the Instituto Plan Agropecuario (IPA) team knew that there was certain knowledge among farmers on adaptation strategies to these extreme situations, it was unclear about how they worked exactly, and which strategy was better in the long-run. This also evidenced the need for new methodological tools for the team to work with, which would also facilitate the communication of these strategies among farmers and also between agronomists (and extensionists) and farmers.

The purpose of the simulation, as stated in step "*Identify System Purpose*", was to simulate the evolution of farmers (e.g. by looking their income over time) under different draught strategies, and to build prospective scenarios, under the assumption that future conditions (climate, prices) will be similar to previous ones (since the input data that was used for those prospections corresponded to the 2000-2009 decade). This purpose was defined by the IPA team before the project started, and even though represents a first approximation to the problem, it helps defining the most relevant elements and concepts that should be taken into account throughout the simulation development. Nevertheless, the purpose does not necessarily define all the concepts that are to be used, but it should give guidance for finding these.

From now on, all steps except the implementation were based on several workshops within the interdisciplinary research project team, including producers.

The second step was the "*Model Domain Concepts*" which detected the main concepts of the problem related to agents, resources and environment (see Figure 3). While the purpose referred to producers, this step concluded that two kinds of producers were to be considered. These two kinds of producers (or agent types), each with its corresponding draught strategy are: Reactive Producers (those who focus on cattle health or corporal condition score in order to make draught-related decisions) and Proactive Producers (those who focus on grass availability and climate in order to make draught-related decisions).

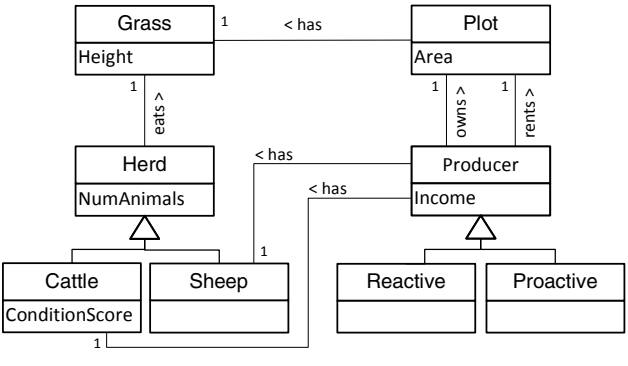


Fig. 3. Domain model (simplified) for the case study

Their behaviors were modeled as stated in step "*Identify Roles and Agent Types*", and Figure 4 shows the UML Activity Diagram that models the Proactive Producer's behavior in winter. It is worth noting that the time-step definition should be defined here since agent's behaviors depend on it (the actions and decisions are not the same if an agent is to be called once a year than if it is every day). Moreover, the time-step could even be defined or suggested in the purpose (e.g. study certain evolution per season would imply to define –at least- a seasonal time-step).

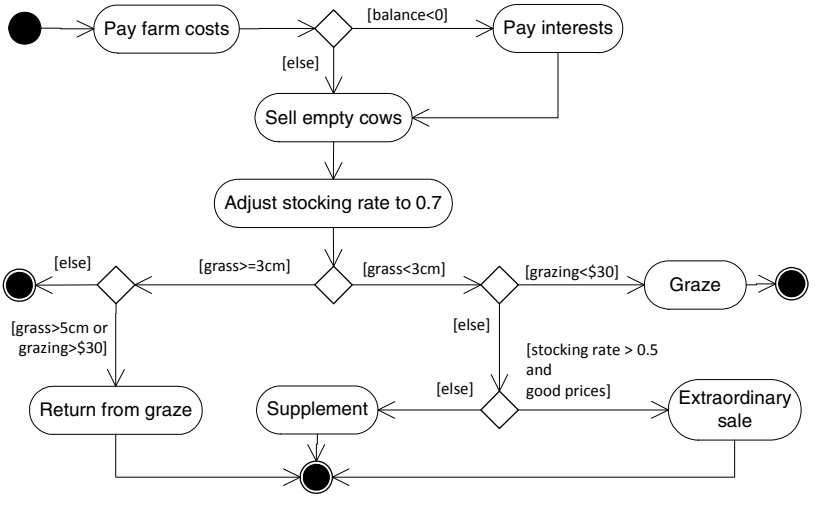


Fig. 4. Winter strategy for a Proactive Producer (CS stands for Condition Score)

For the purpose of the case study, the step "*Define Agent Interaction*" was not considered since there is no interaction between producers in the research project.

Regarding the step "*Define Agent Architecture and Design*", a reactive architecture was chosen for both types of agents, so there will be no explicit representations of beliefs, desires or intentions, but rather each agent will act upon a set of predefined rules, which were captured in the UML Activity Diagrams. The decision of what agent architecture to use heavily depends on whether the simulation platform that will be used (in this case CORMAS) supports other kinds of architectures or not, and also on the purpose of the simulation. Even though it is a technical decision, it impacts the information that needs to be gathered during the modeling. For the design part of this step, a new UML Class Diagram was constructed based upon the concepts of the Domain Model but extending certain CORMAS classes in order to inherit their behavior, which is shown in Figure 5.

The next step was "*Model Environment and Resources*" whose aim was to define behavioral aspects of the environment and resources as well as complete structural ones if needed. For this case study, the former involved defining the cattle lifecycle (see Figure 6) showing each possible cattle state and its transitions, and the latter involved completing the structural aspects of the environment. The same as for agents, the discovery of these new dynamics came from the simulation purpose and the domain model. Therefore, this step can be viewed as a refinement of the domain model, at least from a structural point of view.

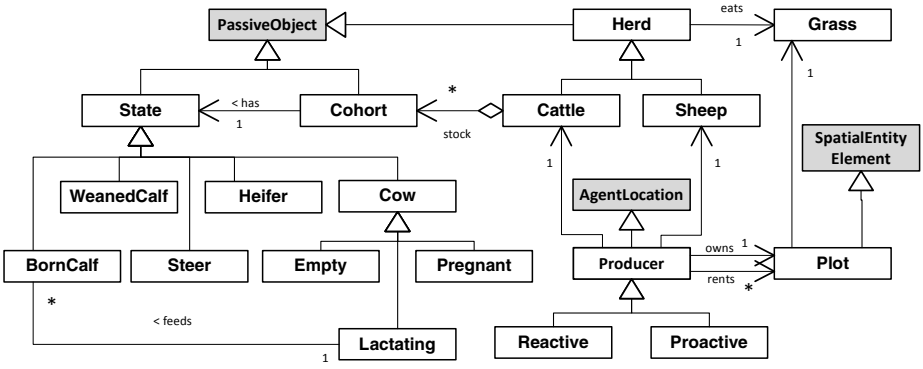


Fig. 5. UML Class Diagram for the case study showing extended CORMAS classes (shown in color) and omitting attributes and operations

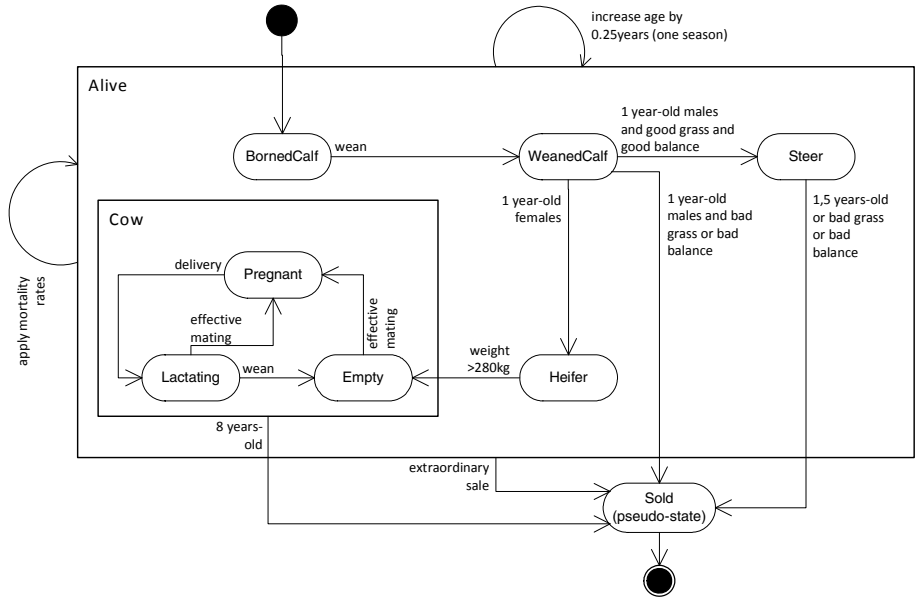


Fig. 6. UML Sate-Transition Diagram representing cattle lifecycle

Up to this point, all steps except "*Define Agent Architecture and Design*" do not require technical (computer science) knowledge, apart from being able to interpret and understand UML diagrams. This allowed for a truly interdisciplinary work and for a white-box approach to modeling and simulation.

The "*Simulation Configuration*" step involved defining the initial configuration of the simulation (i.e. assigning initial values for the previously defined agents, resources and environment), its time-step (for this case study a seasonal time step was defined when identifying agent's behavior), task scheduling order (i.e. in which order the behaviors of agents and resources are called upon at every time step, see Figure 7) and input parameters (e.g. climate data and international prices).

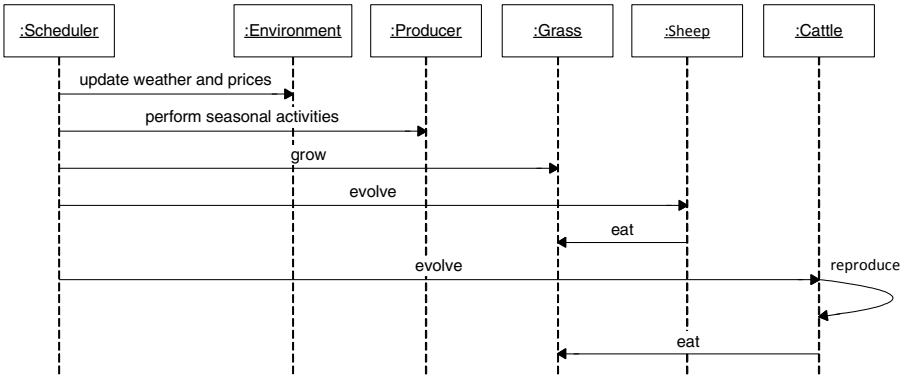


Fig. 7. UML Sequence Diagram representing task scheduling

Particularly, input parameters should be defined earlier in the process since they depend on where the system boundaries are defined. For example, if international markets and prices are not considered within the system at hand but outside its boundaries (as it was the case here) but they affect in some way the system (e.g. by allowing producers to sell their cows when prices rise) then this situation should be early identified and consider these elements as input parameters.

Concerning the output of the simulation, several output parameters were defined, like producer's income and cattle condition score, along with the way in which these should be visualized (mainly in graphs). These outputs, which were already depicted in the domain model, should be consistent with the simulation purpose since they allow users to see the system evolution and analyze their result, which is the aim of the last step: "*Simulation Run and Sensitivity Analysis*".

The "*Implementation*" step for this case study was done in three versions or iterations: first a "grass only model" was implemented that had the objective of validating the grass growth depending on climate; the second "wild model" introduced cattle and aimed at the grass-animal interaction; and finally the "final model" was done which included producers and their different behaviors (draught strategies).

It is worth noting that implementation is straightforward from all the previous models, and especially using such a simulation platform as CORMAS that provides native constructions for such elements as agents, resources and environment.

4.1 Case Study Conclusions

Overall, the methodological framework successfully supported the development of the case study. Knowing what steps should be taken and what artifacts should be constructed provides confidence, saves time, and keeps people focused. Having a mapping between the most common features found in an agro-ecosystem and how they can be represented using the methodological framework was shown to be of high value.

Although the steps were presented in a sequential order, an iterative & incremental approach may well be adopted, benefiting from early user's feedback, in the same way current mainstream software development methodologies do. This would also

imply that a single step would be visited more than once during the development process. Also some of the steps may change if other type of simulation is used, like event-based simulation.

In a truly interdisciplinary team, like the one that developed the case study, to share a basic vocabulary and to have a common understanding of the basic concepts is crucial. To this end, the Domain Model served as a glossary of the most important concepts and provided this common understanding.

Another interesting conclusion was reached in the use of UML diagrams. At the beginning of the project, all team members were convinced that they all shared the same knowledge about basalt producers. However, the use of UML, particularly the Activity Diagrams in this case, demanded experts to explicit their knowledge and assumptions in order to unambiguously build the diagrams. From this process, several contradictory opinions emerged between the different experts, so several discussions took place to arrive at the final versions of these diagrams.

Regarding the producers, along the several workshops of the research project, they felt they were being accurately represented by the UML Activity Diagrams, and even when other variants were proposed (e.g. like considering a third kind of producer as the result of a mix of the other two) the great majority did coincide that the two types of agents (Reactive and Proactive) indeed represented them. This enforces the value of UML as a formal but also intuitive tool for modeling the problem, discussing the domain with experts as well as producers, and afterwards starts the software development bases upon these diagrams.

5 Conclusions

This article introduced a new methodological framework for developing agent-based simulations. The framework is based on already existing general purpose AOSE methodologies, but unlike these, it is tailored to tackle specific agro-ecosystem features. It also relies on the UML with the purpose of an easy uptake from both, the interdisciplinary team which models the agro-ecosystem, and the programmers which will develop the simulation. The methodological framework is not a methodology itself. It focuses on identifying relatively general steps and artifacts (produced by the steps) that could be ensemble for developing such a simulation, whilst other methodological aspects, as defining a strict order among steps and identifying roles for that steps, are not yet considered.

Even though it is not (yet) a complete and fully comprehensive software development methodology, it has already been successfully applied by our team in some research projects.

This methodological framework sets apart from other related work since: **a)** already-existing AOSE methodologies are general purpose agent-methodologies, not focusing on simulation or in agro-ecosystems, both with its own specificities; **b)** even though there are previous related-work like [12] they do not place attention in trying to define a software methodology with its steps and artifacts; and **c)** because this methodological framework summarizes the experience of our group in participating in several research projects and following ad-hoc procedures that are now started to formalized.

Further work is ongoing in different topics with the overall aim of achieving a methodology that is of easy uptake for programmers. From our experience we believe this is a very important issue since human resources for the programming tasks are usually (very) scarce, and the more specific knowledge we require the programmers to have, the more scarce the resources becomes. This is why it is important to keep the methodology simple, using tools and techniques already familiar to programmers (like object-oriented programming, UML and iterative and incremental development).

This further work includes: **a)** the completeness of the methodology, including aspects such as which roles (in the development process) do what, when and how; **b)** evaluate the use of specific graphical modeling languages such as AML and particularly analyze the benefits of including it against the need for programmers to know it; **c)** develop the necessary software tools to assist in the development process (like plug-ins to certain integrated development environment); **d)** propose a semi-automatic construction process from models to simulation code, supported by the model-driven approach of the methodology; **e)** continue to use and refine the methodology and get programmer's feedback about its chances to get real uptake.

The main contributions of this methodological framework are: **a)** the only methodological framework for developing agro-ecosystem simulations using the agent-based modeling approach. Up to our knowledge, there is no other methodology (nor methodological framework) that copes with this problem; **b)** the only case in which the ABM approach was applied to a Uruguayan agro-ecosystem problem, in this case the draught phenomena in the basalt region and a study on how it affects cattle breeders and their draught strategies; **c)** the possibility to get regular OOP+UML software developers to quickly start writing agro-ecosystem simulation software, without previous knowledge of MAS. This becomes increasingly important because it reduces the time and skills required to get into a multidisciplinary team, especially when software developers are hard to find; and **d)** the possibility to formalize a methodology that allows developers (as stated above) as well as experts (e.g. agronomists) and producers to jointly participate in the development of a simulation, achieving a truly interdisciplinary work.

References

1. Cervenka, R., Trencansky, I.: The Agent Modeling Language - AML: A Comprehensive Approach to Modeling Multi-Agent Systems, 1st edn. (2007)
2. CORMAS (n.d.), http://cormas.cirad.fr
3. Corral, J., Arbeletche, P., Morales, H., Burges, J., Continanza, G., Courdin, V.: Multi-Agent Systems applied to land use and social changes in Rio de la Plata basin (South America). In: 8th European International Farming Systems Association, France (2008)
4. Corral, J.: Agent-based methodology for developing agroecosystem simulations. MSc thesis. Centro de Posgrados y Actualización Profesional, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay (2011), http://www.fing.edu.uy/inco/pedeciba/bibliote/cpap/tesis-corral.pdf
5. CSIRO. Complex or just complicated: what is a complex system? CSIRO fact sheet (2008), http://www.csiro.au/resources/AboutComplexSystems.html
6. FIPA, http://www.fipa.org

7. FPTA Project, http://www.inia.org.uy/busqueda/proy_detalle.phtml?id=186&origen=1
8. Gliessman, S.R.: Agroecology: ecological processes in sustainable agriculture, Ann Arbor Press (1997)
9. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-oriented Methodologies. Idea Group, Hershey (2005)
10. INIA, http://www.inia.org.uy
11. IPA, http://www.planagropecuario.com.uy
12. Le Page, C., Bommel, P.: A methodology for building agent-based simulations of common-pool resources management: from a conceptual model designed with UML to its implementation in CORMAS. In: Bousquet, F., Trébuil, G., Hardy, B. (eds.) Companion Modeling and Multi-Agent Systems for Integrated Natural Resource Management in Asia, pp. 327–349. IRRI, Metro Manila (2005)
13. Miller, J.H., Page, S.E.: Complex adaptive systems: an introduction to computational models of social life. Princeton University Press, Princeton (2007)
14. Morales, H., Arbeletche, P., Bommel, P., Burges, J.C., Champredonde, M., Corral, J., Tourrand, J.F.: Modéliser le changement dans la gestion des terres de parcours en Uruguay. Cahiers Agricultures 19(2), 112–117 (2010)
15. Norman, M.: Annual Cropping Systems in the Tropics. University Press of Florida, Gainnesville (1979)
16. OMG. The Unified Modeling Language Specification v2.0 (2005), http://www.uml.org
17. Rao, A., Georgeff, M.: Modeling Rational Agents within a BDI-Architecture. In: Allen, J., Fikes, R., Sandewall, E. (eds.) Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR 1991), pp. 473–484. Morgan Kaufmann, San Francisco (1991)
18. Tran, Q.N., Low, G.: MOBMAS: A methodology for ontology-based multi-agent systems development. Information and Software Technology 50, 697–722 (2008)
19. Wooldridge, M.: Lecture Notes on Introduction to Multiagent Systems Course (2008), http://www.csc.liv.ac.uk/~mjw/pubs/imas/teaching.html
20. Wooldridge, M.: An Introduction to Multiagent Systems. Wiley & Sons, Chichester (2002)

Development Policy Analysis in Mali: Sustainable Growth Prospects*

Matteo Pedercini

Millennium Institute
Washington D.C., U.S.A.
mp@millennium-institute.org

Abstract. In the context of the implementation of the second-generation poverty reduction strategy (CSLP II) in Mali, we investigate the country's development potential, within existing resource constraints. We apply an integrated, resource-based approach to growth, implemented through a system-dynamics-based national development planning model. Scenario analysis indicates that the policy orientation of the CSLP II might foster growth in the long run, but, even in our most optimistic scenario, the government's stated growth and development goals are unlikely to be achieved. Our results highlight the importance of endogenous growth mechanisms for sustainable development, and the significance for economic performance of the major delays involved in the accumulation of the resources that are necessary for growth. We believe that our approach contributes to the most commonly used tools for medium-long term planning by providing a dynamic perspective on the key resources for growth and on the constraints to the country's development.

Keywords: Growth potential, sustainable development, system dynamics, resource constraints, Mali.

1 Introduction

Economic statistics about Mali give a rather clear picture of one of the poorest countries in the world. Most of the 12 millions Malians live below the official poverty line (59.2% in 2005) [1]. Despite the good economic growth observed over the last decade (on the average about 5% per year) the resources available to the Government are insufficient to provide the growing population with broad access to basic social services. In terms of Human Development Index[1] (HDI), Mali ranks 175 of 177 [2]. In addition, Mali development is complicated by structural constraints in terms of natural resources, among others: water is scarce in large parts of the country; forest cover is limited to 10% of the country and rapidly diminishing; mineral resources are limited and heavily exploited.

* An earlier version of this paper was presented as part of the Doctoral Thesis "Modeling Resource-Based Growth for Development Policy Analysis", Matteo Pedercini, University of Bergen, Norway, 2009.

[1] The Human Development Index is a composite index including literacy, life expectancy and income.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 447–463, 2011.

As several other countries in Africa, Mali developed a first medium-term strategic plan for poverty reduction in the early 2000 (the so called "Poverty Reduction Strategy Paper", PRSP, or CSLP in French). The CSLP-I set very ambitious goals for the country – above all a target economic growth rate of 6.7% per year. While setting an ambitious goals can encourage development [3], when goals are set at unachievable levels, performances are invariably perceived as a failure and disillusion arises. Mali is not an exception to the general tendency observed for Sub/Saharan African countries [4] of stating overoptimistic goals [5]. The projections of growth contained in the country's first CSLP, were unrealistic and have not been fulfilled [6]. This failure highlighted the need for strengthening the CSLP process, and for implementing a more realistic and effective second generation CSLP (CSLP-II).

In this context, we address two fundamental issues: (1) Under current policy regime, existing resource constraints, and external conditions, would Mali be able to achieve its stated development goals? (2) Under a policy regime reflecting CSLP II indications and positive external conditions, would Mali be able to achieve its stated development goals?

The analysis of strategies for sustainable development, intended as "development which meets the needs of the present without compromising the ability of future generations to meet their own needs"[2], requires studying the dynamic interrelations between development and the country's own fundamental resources. In order to fulfill such requirementwe develop a quantitative planning model based on the Threshold-21 framework [7], and use it to elaborate sustainable development scenarios in the context of the implementation of the CSLP-II. These scenarios are developed involving a broad range of stakeholders, to increase participation in the planning process, as well as optimism and proactive behavior towards the future [8].

The following section describes our approach and method, while Section 3 describes the validation process the model underwent and describes the base run of the model. Section 4 illustrates the results from a set of alternative scenarios. Finally, Section 5 summarizes the results of the study and provides reflections on the usefulness of the approach to support the development of poverty reduction strategies.

2 An Integrated Resource-Based Approach

2.1 Approach and Method

In order to provide an integrated and dynamic perspective on growth, we adopt a resource-based approach [9-12] to development policy analysis. Although the relevance of the allocation of resources for economic growth is intrinsic in the definition of economics itself [13], the resource-based approach has hardly diffused into growth research. Recently, Warren has introduced a quantitative and dynamic framework for resource-based analysis, focusing in particular on accumulation and feedback processes [14], which is suitable to support growth research for policy analysis.

Over the last two decades, a variety of quantitative modeling methods have been developed to support national policy analysis. The most popular of such methods

[2] According to the definition of the UN "Report of the World Commission on Environment and Development: Our Common Future", also known as Brundtland Report, 1987.

include Computable General Equilibrium (CGE) modeling [15], Macro-Econometric (ME) modeling, Disaggregated Consistency (DC) modeling [16], and System Dynamics. The first three methods have proven useful for various kinds of policy analyses, but are only to a limited extent applicable to medium-long term planning, since they do not support the analysis of environmental dynamics and can only partially represent development of social factors. We thus chose to implement our approach by way of the System Dynamics method.

System Dynamics has recently stood out as a technique to analyze a variety of development issues [17-19], including national policy analysis [20]. The SD method has been developed to analyze the relationship between structure and behavior of complex, dynamic systems [21]. In SD models, causal relationships are formalized into models of differential equations, and their behavior is simulated and analyzed via simulation software. Regular simulations are performed through a recursive computational sequence, while the most powerful software also allow for optimization and sensitivity analysis (through Montecarlo simulation)[3]. With respect to visualization of the model's structure, the method uses a stock and flow representation of systems that allows to maintain a certain degree of transparency. The flexibility, the capability to represent dynamic complexity, and the transparency that characterize SD make it well suited to implement an integrated, dynamic resource-based approach to development policy analysis.

2.2 The Model

In order to develop our model, we used as starting framework the Threshold21 (T21) model developed by the Millennium Institute [7]. T21 is an integrated scenario-analysis tool designed to support national development planning, and it has been applied in various countries [20, 22]. We chose T21 as starting framework since this is the most widely used model that integrates into one framework the most essential social, economic and environmental aspects of development, allowing for a comprehensive long term policy analysis. T21's structure is appropriate for this type of analysis, and adopting such framework is cost-effective: rebuilding from scratch a similar framework would require several months. T21 offers a solid and well validated basis upon which we developed a T21-Mali model, embedding an integrated resource-based approach to development policy analysis.

The T21-Mali model, while keeping the broad and integrated approach that characterizes the generic T21 framework, is specifically developed with an emphasis on representing and analyzing the dynamics of the resources that are central in growth. With respect to the T21 framework initially used, the model includes several additional sectors, and a variety of relationships have been reformulated to better match reality in the country. The model has been conceived, in particular, with two functions in mind: to provide exploratory strategic and external scenarios [23] for the CSLP-II; and to project these scenarios way beyond the time horizon of the CSLP-II (5 years) to analyze their long-term implications. Based on these objectives, and on the limited availability of historical data, we set the time horizon for the analysis to the period 1990-2025.

[3] The most broadly used SD software inlcude Vensim, Powersim, and Ithink: www.vensim.com, www.powersim.com, www.iseesystems.com

The development of T21-Mali has been carried out in close collaboration with Malian Committee for Planning and Forecasting (CPM)[4]. Specifically, identification of central resources for development and estimation of key relationships, that were initially based on existing literature, have been improved and validated by CPM. Such collaborative model development process led not only to a better specified model structure, but also to building confidence of final users in the model and underlying data sets.

Figure 1 provides a high level representation of the structure of the T21-Mali model[5]. The structure is composed of three spheres (economy, society, and environment), each including six sectors. All sectors are dynamically interacting with each other, within the same sphere, as well as across different spheres. Social resources, natural resources, and economic resources, all contribute to economic production and are affected by it, providing an endogenous perspective on growth and development.

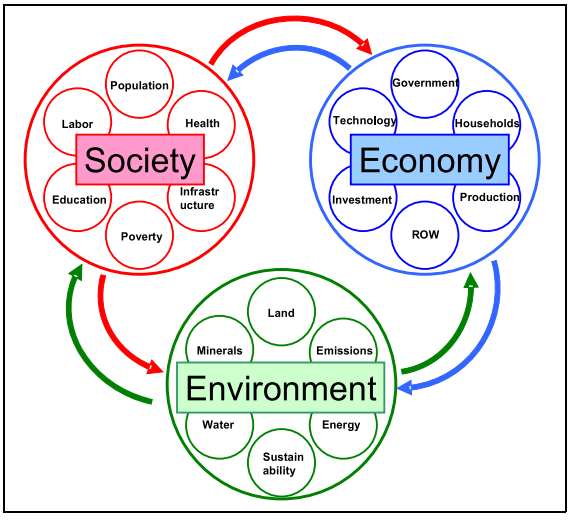


Fig. 1. High level representation of the T21-Mali model

The key sector in the economy is production, where resources of various kinds – economic, social, and environmental – converge to give rise to economic production. We adopt a Cobb-Douglas production function [24] with an endogenous treatment of total factor productivity. We identified physical capital, human capital, and infrastructure as the key resources for the country's development [25, 26]. In addition to these, we also consider the effect of environmental resources, such as agriculture land and gold reserves in the analysis [27]. The resources generated through economic

[4] The CPM is composed of a panel of technicians and policy makers from various Ministries, with a preponderant participation of officials form the Ministry of Planning, and provides advices to other governmental institutions for the preparation of mid and long-term plans, including the CSLP.

[5] The T21Mali model and its complete documentation (English and French) are available at www.millennium-institute.org/projects/africa/mali.html

production are allocated between consumption and investment. Investment in physical capital mostly takes place in the private sector, enhanced by foreign direct investment and remittances from abroad. Investment in human capital, via better education and health, fundamentally takes place as public spending in infrastructure and staff in the education and health sectors. Key public infrastructure (such as transportation and telecommunication infrastructure) also develops as a result of public investment.

In the society, the population sector represents the key mechanisms underlying demographic development. Fertility is determined based on income and education [28], and mortality based on income and access to health care [29, 30]. The labor sector accounts for labor supply/demand balances, distinguishing between skilled and unskilled labor. The health and education sectors determine respectively the level of access to basic health care and the adult literacy rate, based on the level of public service offered. The poverty sector determines the level of monetary poverty using a Lorenz curve approach [31, 32], and the infrastructure sector represents specifically roads and irrigation infrastructure.

In the environment, land, water, minerals and other natural resources are used to sustain production and to cover basic needs, and are regenerated based on their natural cycles. Some of these resources, such as gold, are not renewable; others such as water are renewable, but available only in limited quantity and locations [33]; and others, such as land are fixed, but can be shifted to a limited extent among different uses. Energy is generated using both internal resources (i.e. hydropower) and external resources (imported fossil fuels) [34]. Long term sustainability is assessed using the ecological footprint [35].

3 Base Run

3.1 Data and Validation

As a mean of validation, we simulate the model starting in 1990 and compare the simulation results with the historical data available for a set of relevant indicators. In order to do so, we developed a comprehensive data base, including records for more than 200 variables, over the period 1990-2005[6]. The data base is based on data from Mali's National Statistical Office (DNSI) [36], supplemented with data from internationally accredited sources [1, 34, 37-41].

The set of indicators reported in this study have been selected in order to cover the most relevant aspects of the country's broad socio-economic development. The list covers the key aspects of the Millennium Development Goals (MDG) [42] and of the Human Development Index, including indicators for poverty, education, health and environmental sustainability. The same list of indicators is used to present the results of the reference scenario in section 3.2 and of the alternative scenarios in section 4.

The results obtained for the major indicators for the period 1990 to 2005 are well in line with historical data as summarized in Table 1. Table 1 reports the root mean square percent error (RMSPE), the Theil's inequality statistics, and the coefficient of determination (R^2) resulting from the comparison of the base run with historical data for the selected indicators. The RMSPE is an appropriate indicator of the goodness of

[6] At the time the analysis was completed, latest year for which data was available was 2005.

fit of system dynamics models [43], and its decomposition through Theil's inequality statistics indicate the nature of the discrepancy between model results and data. In particular, Theil's inequality statistics decompose the RMSPE into three components: its bias component (U^M); its unequal variation component (U^S); and its unequal covariation component (U^C).

Table 1. Summary statistics for key indicators (comparison period: 1990-2005)

Variable	RMSPE	Inequality Statistics			R^2	Data Points
		U^M	U^S	U^C		
real Gross Domestic Product (GDP)	0.018	0.348	0.028	0.623	0.996	16
agriculture production	0.045	0.156	0.001	0.842	0.934	16
industry production	0.038	0.009	0.001	0.990	0.991	16
services production	0.024	0.296	0.002	0.702	0.988	16
public deficit as share of GDP	0.172	0.006	0.002	0.992	0.862	16
total population	0.009	0.731	0.108	0.161	0.998	16
average life expectancy	0.044	0.911	0.004	0.085	0.871	4
average adult literacy rate	0.012	0.492	0.000	0.508	0.992	10
proportion of population within 5 km from health center	0.052	0.367	0.148	0.485	0.975	8
road km per 1000 people	0.028	0.546	0.155	0.298	0.814	10
proportion of population connected to electricity network	0.040	0.001	0.076	0.923	0.958	5
proportion of population connected to water network	0.032	0.667	0.238	0.096	0.997	5
proportion of population below poverty line	0.049	0.262	0.460	0.278	0.816	4
Human Development Index	0.077	0.641	0.318	0.041	0.983	4

The high R^2 values obtained for the selected indicators highlight that the model can explain a large portion of the recorded historical data. Although in some cases, e.g. total population, life expectancy or HDI, most of the error is of systematic nature (i.e. either bias or unequal covariation), the size of the error itself is relatively small, and does not diminish our confidence in the model's ability to capture the long-term trend in the data. Of major concern, however, is the limited amount of data available for a few of the variables. Data on poverty and life expectancy in particular, are limited to four data points: Although we regard this as the most reliable data available, a longer data series would have provided a more solid basis upon which we could test our model. Data scarcity partially affects also our estimation of the HDI, which is a composite index including life expectancy, literacy rate, and income. In addition, we have a limited number of data points for indicators related to infrastructure (health centers, roads, electricity and water distribution). Nevertheless, the nature of these variables is such that they can be more directly estimated, and thus such data tends to have smaller measurement errors. Overall, we consider the model's ability to replicate historical data acceptable for the purpose of this analysis.

In the process of model development, behavioral validation of the model was coupled with a structural validation process [44] which included, in particular, a verification of the assumptions and parameters' values – initially estimated based on available data and information – with local experts from the various sectors portrayed in the model.

3.2 Business as Usual Scenario

In order to establish a foundation for the development of alternative scenarios for the CSLP-II analysis, we first generate a reference scenario, which we call "Business as Usual Scenario" (BAU). In this scenario, we simulate the model until 2025 assuming that the future policy orientations and external conditions correspond to those that characterize the CSLP-I period. In the following paragraphs, we provide a summary of the results obtained in this scenario. The results for the period 2005 – 2025 for the selected indicators are reported in Table 2.

Table 2. Summary results for the Business As Usual scenario

Time (Year)	**2005**	**2010**	**2015**	**2020**	**2025**
real Gross Domestic Product (GDP) growth rate	5.3%	3.4%	3.2%	3.2%	3.5%
agriculture production growth rate	5.1%	2.5%	1.8%	1.6%	1.7%
industry production growth rate	4.8%	3.5%	3.4%	3.6%	3.8%
services production growth rate	4.5%	4.9%	4.6%	4.6%	4.7%
public deficit as share of GDP	4.8%	3.8%	3.3%	2.7%	2.0%
total population growth rate	2.5%	2.5%	2.4%	2.3%	2.2%
average life expectancy	52.2	55.5	58.8	61.5	63.9
average adult literacy rate	30.5%	36.3%	42.0%	47.2%	51.6%
proportion of population within 5 km from health center	48.7%	54.1%	59.8%	65.8%	70.6%
road km per 1000 people	1.61	1.70	1.78	1.85	1.93
proportion of population with access to electricity	15.6%	21.5%	30.4%	36.3%	40.3%
proportion of population with access to potable water	63.5%	68.4%	74.4%	80.3%	86.0%
proportion of population below poverty line	64.7%	62.4%	60.3%	58.2%	55.6%
HDI	38.5%	43.5%	47.9%	51.4%	54.5%

This scenario portrays a discouraging picture of Mali's development over the next two decades. While the population is growing at sustained rates, the GDP growth rate is decreasing and then flattening out at about 3.5%, and the resulting growth in per capita GDP is small. As a consequence of the modest economic growth, poverty rates (defined here as the proportion of population below the official poverty line) are declining of less than 10 percentage points over 20 years, from 64.7% to 55.6%. This performance can be considered insufficient by any national or international standard: The internationally agreed MDG indicate a target poverty level in Mali of about 22% by 2015. Education is also progressing slowly, resulting in a literacy rate of only about 50% of in 2025. Life expectancy increases modestly up to about 64 years in 2025. The combination of these results, leads to an HDI growing slowly from 38.9% to 54.5%, which would bring Mali in 2025 at about the level of today's Pakistan.

The results obtained in this scenario are incompatible with the MDG and with the basic targets for the CSLP-II [3]. The goals for the CSLP-II, in particular, include achieving and sustaining an average 7% GDP growth rate per year over the next five years. The reason of such poor performance, in spite of the country's efforts and plans for poverty reduction, is deeply-rooted in the nature of the mechanisms that drive growth in the country.

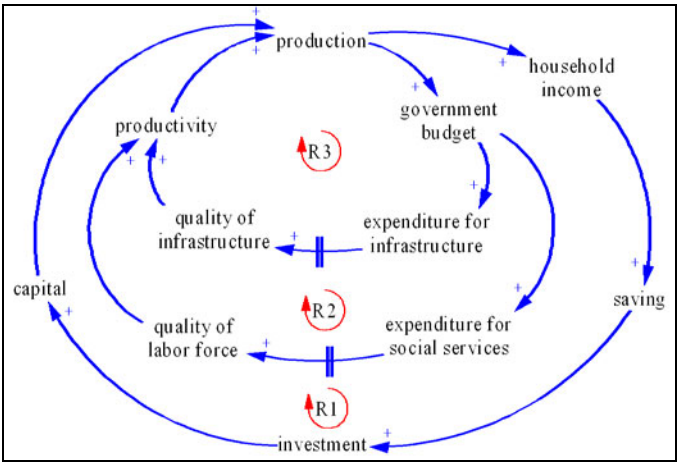


Fig. 2. High level representation of the mechanisms underlying the socio-economic development of Mali

The Causal Loop Diagram (CLD) in Figure 2 provides a high-level interpretation of these mechanisms. CLD are qualitative representations of the causal relationships in a system of interest, and are used to highlight feedback loops relevant to the issues being analyzed [45]. In CLD, arrows with positive polarity indicate a unidirectional relationship between two variables (i.e. an increase in the independent variable causes the dependent variable to take on a value over and above what the value would otherwise be). All the arrows in Figure 2 have positive polarity, and they describe three positive, or reinforcing, feedback loops [46].

The reinforcing feedback loops in Figure 2 are all sources of economic growth: An increase in production leads to an increase in the government budget and households' income, which leads to a larger investment in physical capital, infrastructure, and basic social services, such as education and health care. Capital, infrastructure and a healthy and educated labor force are essential resources for development in Mali [3]. As these resources are built over time via private and public investment, productivity and production further increase, thus closing the loop.

The three reinforcing mechanisms described above are at the heart of Mali's development. However, these mechanisms might work only slowly, for two principal reasons: First, major delays are involved in the accumulation of human resources and in the creation of proper infrastructure (indicated by the vertical bars on the arrows in Figure 2); second, when the initial base of resources is small, the resulting low income level allows for limited saving and thus little accumulation of resources.

Rapid accumulation of physical capital is a first prerequisite for growth (see feedback loop R1 in Figure 2). However, over the last decades (going back as far as economic data has been collected in Mali) per capita disposable income has been especially low in Mali, even compared to the average for Sub-Saharan countries [1]. This low level of income implies a high propensity to consume, and thus little funds are allocated to saving and investment. Such a dysfunctional capital accumulation

mechanism is well known in many developing countries, and is often labeled a "poverty trap" [47].

A second mechanism of great importance for development in Mali is represented by the R2 loop in Figure 2. A low level of production implies little resources available for the government to invest in the provision of basic social services that are indispensable to the population. Social services such as education and health care are vital to the social development of the country, and a healthy and educated labor force is also a key factor in the increase and maintenance of labor productivity [48, 49], production, and eventually government revenues. Unfortunately, the government does not have large amounts of resources to invest in education and health, and, even in the case financial resources were readily available, accumulation of human resources would require a long time.

The Government of Mali also plays a fundamental role in the country as provider of basic infrastructure, in particular for transportation and agriculture (R3 in Figure 2). Historically, as the resources available to the government have been limited, investment in these sectors has been scarce. Creating efficient infrastructure is also particularly time and resources consuming. It takes a long time to establish effective infrastructure, and that infrastructure is often exposed to natural depletion, and requires a continuous maintenance effort. The low level of infrastructure implies high production and delivery costs for goods and services, and thus harms productivity [26, 50]. This, in turn, implies a slow development of the economy and little growth in government revenues.

For the reasons discussed above, the three positive feedback loops illustrated in Figure 2 are not sufficiently effective in Mali. Given the initial low level of resources available, and the long delays involved in their accumulation processes, it would have been difficult for the country to perform better than it did over the last 10 years (GDP growth rate oscillating around 5%). Undoubtedly, over the past decades several developing countries have experienced two-digit growth rates for extended periods. This is fostering new hopes for developing economies. However, growth accelerations in these countries seemed to be triggered by idiosyncratic small scale events [51], indicating that the fundamental resources for production were readily available: As a few crucial bottlenecks were removed, growth spurred. This is not the case of Mali

The economic development of Mali is in addition harmed by the limited possibilities for improvement in the exploitation of two key environmental resources. Gold mining, an important part of industrial production, has shown signs of decline over the last few years. Due to the depletion of reserves, the growth in gold extraction over the last five years is close to zero, and it is not expected to change substantially in the near future [52]. Moreover, the limited water availability is an important environmental constraint to Mali's development. Water supply is concentrated in the southern region, and only a limited share of Mali's surface can thus be used for agriculture [53]. The possibility for growth in agriculture production through an increase in the irrigated area exists therefore only in some areas of the country.

These environmental factors that constrain growth in Mali are structural characteristics of the country. Even if improvements can be made in the management of these resources, little can be done to sustainably increase their amount. Our investigation focuses mostly on the mechanisms that can bring about further development through

the expansion of human resources, physical capital and infrastructure, within the limits posed by the scarce availability of natural resources.

4 Alternative Scenarios

In order to assess whether under a new policy regime the country can accelerate its development and achieve its growth targets, we simulate four alternative scenarios, developed with experts from CPM. The first alternative scenario includes four policy changes directed at increasing productivity in the private sector, this being the first major objective of the strategy developed in the CSLP-II. The other three scenarios gradually introduce a number of realistic – albeit highly optimistic – assumptions with regard to external conditions.

The major changes in policies and external conditions introduced in the four alternative scenarios with respect to the BAU scenario are described in Table 3. Changes to the model indicated as policies (P) reflect actual interventions that the government could implement (e.g. an increase in public expenditure for transport and equipment); changes indicated as assumptions (A) reflect developments in relevant variables over which the government has no direct control. Changes are implemented cumulatively, so that each new scenario contains the same changes introduced in the previous one, plus one or more additional changes. The column to the right-hand side of the table indicates in which alternative scenario a specific policy or assumption is used.

Table 3. Summary of changes in policy and assumptions for the alternative scenarios

Type	Description	Scenario
P	Increase of 33% in expenditure for transport and equipment by 2025	1,2,3,4
P	Increase of 25% in expenditure for rural economic activities by 2025	1,2,3,4
P	Tripling of expenditure for industry, mining and water as share of govt. budget by 2025	1,2,3,4
P	Lower fiscal pressure in the first 5 years, and subsequent compensating increase	1,2,3,4
A	More than doubling of foreign direct investment as share of GDP by 2025	2,3,4
A	Increase of gold discoveries by 20% by 2025	3,4
A	Rapid increase in gold extraction by 30% by 2011	3,4
A	Shift of land use from pasture to crops (average increase of 20% in arable land)	4
A	Gradual increase in value added per unit in the cotton sector (75% increase by 2025)	4
A	Reallocation of agriculture land among crops to increase efficiency in the sector	4
A	Improvements in the weather conditions for agriculture production (from a minimum of 10% improvement for rice, to a maximum of 100% for maize)	4
A	Increase of fish industry production by 60% in 2025	4
A	Increase in tons of meat produced per head by 50% in 2025	4
A	Decrease in Gini[7] coefficient from .525 to .45	4

[7] The Gini coefficient is an aggregate index of inequality in income distribution. A Gini coefficient of 1 indicates that the entire national income is earned by one only individual (or household), while a Gini coefficient of 0 indicates a perfectly uniform distribution.

The first alternative scenario analyzed, only includes the four policy changes listed at the beginning of Table 3. These policies aim mainly at fostering the development of essential infrastructure, while lowering fiscal pressure and thus facilitating private investment in physical capital. The result is a gradual, steady increase in GDP growth rate compared to the BAU scenario, reaching about 4.4% in 2025. Figure 3 illustrates the GDP growth rate obtained in this first alternative scenario (line 1) compared to the BAU scenario (line 0), and to the other scenarios that are discussed in the following paragraphs. The primary reason of the slow response of GDP growth to the implemented policies is to be found in the significant delays involved in infrastructure development: Only towards 2010 a significant part of the new infrastructure is completed and starts positively affecting productivity. In addition, although the reduction in fiscal pressure allows for more savings, these are absorbed by the government borrowing on the domestic market, which is necessary to pay for the new infrastructure. In other words, the government borrowing is crowding out private investment. Overall, therefore, in this first alternative scenario the government shifts more of the country's financial resources towards the construction of infrastructure, with a positive impact on growth in the long run. These policy interventions alone are not sufficient to rapidly increase the growth rate and bring it close to the CSLP II target.

In the second alternative scenario, we assume a gradual increase in foreign direct investment (FDI), which reaches up to 8% of GDP in 2025. This substantial increase in FDI leads to a steeper increase in GDP growth rate (line 2 in Figure 3) than in the previous scenario. Eventually, the growth rate reaches about 6.5% in 2025. These results are achieved through a more rapid accumulation of physical capital, that is, through stimulating the R1 loop in Figure 2. This stimulus is fundamentally driven by an exogenous inflow of financial resources, although in the long run, as income in the country increases, domestic investment also takes off. This scenario highlights how the action of the R1 loop, now stronger, brings about a growth rate close to the target of 7% only in the long run.

The first and second alternative scenarios illustrated above highlight how economic growth can be accelerated more directly by stimulating private investment and thus strengthening the action of the R1 loop than by investing in infrastructure. Improvements in productivity and performance through the loop R2 require longer time, and even longer delays are involved in the loop R3. The faster response of the system that we obtain when strengthening the action of the R1 loop should not lead to the conclusion that an optimal policy for sustained growth should focus primarily on stimulating the accumulation of physical capital. Growth through the development of proper human resources and infrastructure is desirable, not only for the intrinsic benefits that it brings in terms of human development, but also for its essential role to sustain growth in the long run. For example, among the assumptions introduced in the second scenario, we included a strong increase in foreign direct investment (FDI): It is more likely that such growth in FDI takes place if human resources and infrastructure are in abundance in the country [54, 55]. Moreover, resources that take the longer time to accumulate are also those that bring about the larger and more sustainable competitive advantage [56]. Human resources and infrastructure are thus aspects of development that should thus not be neglected for growth to be sustainable, although they might exhibit important dynamics only in the long run.

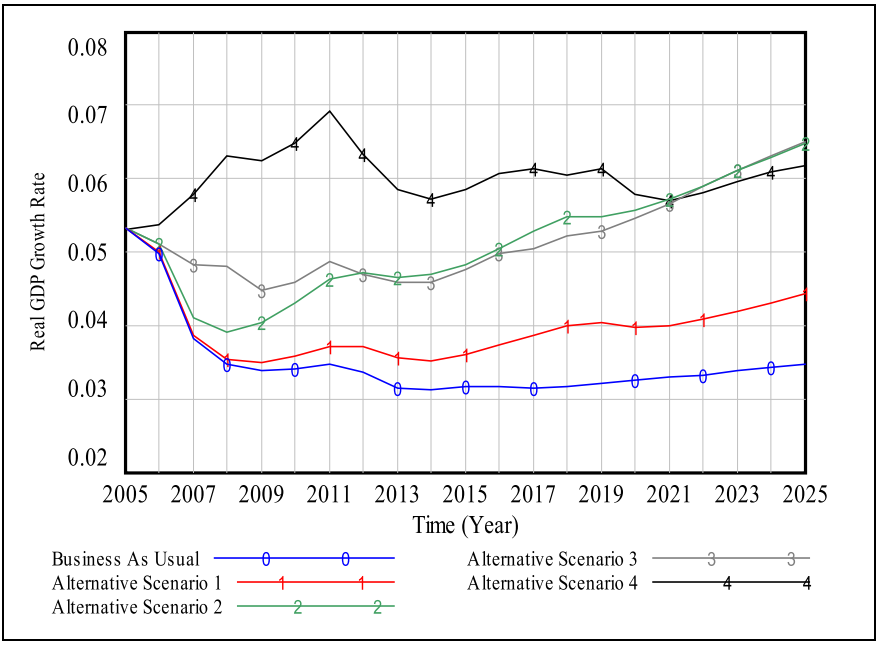


Fig. 3. Mali real GDP growth rate: model projection for the "Business as Usual" scenario (line 0), and the alternative scenarios (lines 1-4)

Under the assumptions and policy framework discussed so far, the reinforcing loops illustrated in Figure 2 cannot bring about a growth rate close to 7% in the short run. In our third scenario, we assume an increase of gold discovery and extraction, as illustrated in Table 3. Such assumptions cause an immediate increase in GDP growth rate, as new gold reserves are discovered and exploited (line 3 in Figure 3). GDP growth rate in the period 2007-2011 is on average about half a percentage point higher than in the previous scenario. The impact of the growth in the gold sector is however short-lived, as the eventual depletion of gold reserves makes extraction more expensive and thus reduces productivity. In the long run, GDP growth rate is in line with that obtained in the previous scenario.

In our fourth and last scenario, we introduce a set of optimistic assumptions regarding the agriculture sector. More specifically, we assume higher productivity for crops, animal husbandry, and fish production, deriving both from an improvement in climatic conditions, and from exogenous increases in firms' efficiency, as illustrated in Table 3. In addition, we assume a significant shift in agriculture land use, form pasture to arable land, leading to an average increase of about 20% in arable land. The combined effect of these optimistic assumptions leads, within 2-3 years from their introduction, to an increase in GDP growth rate above of 6% (line 4 in Figure 3).

The set of policies and assumptions introduced in our fourth scenario results not only in faster economic growth, but also in better performance for nearly all relevant socio-economic indicators, as illustrated in Table 4. The picture of Mali's development over the next two decades that this scenario portrays is more in line with the

country's targets than the one derived from the BAU scenario. Although the population is growing at nearly the same rate as those observed in the BAU scenario, the GDP growth rate is substantially higher, averaging 6.3% for the period 2007 – 2011.

Table 4. Summary results for the fourth, most optimistic scenario

Time (Year)	**2005**	**2010**	**2015**	**2020**	**2025**
real GDP growth rate	5.3%	6.5%	5.9%	5.8%	6.2%
agriculture production growth rate	5.0%	7.7%	4.3%	3.8%	4.7%
industry production growth rate	4.8%	5.3%	5.6%	6.0%	6.4%
services production growth rate	4.5%	6.3%	8.2%	7.9%	7.3%
Public deficit as share of GDP	4.9%	3.9%	3.1%	3.2%	3.3%
total population growth rate	2.5%	2.5%	2.5%	2.5%	2.4%
average life expectancy	52.2	56.6	63.0	68.6	72.1
average adult literacy rate	30.5%	36.3%	42.1%	47.5%	52.3%
fraction of population within 5 km from health center	48.7%	54.2%	61.6%	70.3%	73.3%
road km per 1000 people	1.6	1.7	1.9	2.2	2.5
proportion of population with access to electricity	15.6%	23.1%	41.3%	58.9%	63.9%
proportion of population with access to potable water	63.5%	70.8%	90.3%	100%	100%
average share of population below poverty line	64.6%	58.2%	50.5%	42.1%	33.9%
HDI	38.5%	44.2%	50.8%	56.6%	61.1%

The higher growth in GDP brings about a related growth in per capita GDP. Due to the growth in per capita GDP and the assumed reduction in inequality, the proportion of population living below the poverty line is projected to decrease by 2025 to 33.9%, nearly half the level observed in 2005 (64.6%). Education is progressing slightly faster than in the BAU case, resulting in a literacy rate of about 53.2% in 2025. Life expectancy shows a significant increase up to about 72 years in 2025. The combination of these results leads to an HDI growing slowly from 38.9% to about 61.1% in 2025, about the level of today's India.

5 Conclusions

This paper illustrates how an integrated resource-based approach is applied to analyze alternative sustainable development scenarios in the context of the preparation of the second-generation five-year strategic plan in Mali. Results from our "Business as Usual" scenario indicate that a continuation of the strategy implemented during the period of the first CSLP would lead to a performance well below the stated targets for poverty reduction and development. The results from our most optimistic scenario illustrates that, when introducing policies directed at increasing productivity in line with the strategic indications of the CSLP-II and under optimistic assumptions, better results are achieved overall. But the target GDP growth rate of 7% still cannot be achieved. Although some increase in productivity can be obtained in the short run, the

building up of key resources for growth – infrastructure and human resources in particular – within the existing environmental constraints, requires a few decades.

Our results stress the importance of properly considering the time lags involved in improving productivity through the accumulation of human resources and infrastructure. Awareness of these delays is important when formulating adequate development goals, and to prevent discarding valid development strategies in view of sluggish initial performance. Instead, the focus should not be turned away from the resources that take longer to accumulate, as they represent a potential solid foundation for sustainable development. Also, once the major delays in the growth mechanisms have been identified, strategies and technologies to reduce them can be developed, assessed, and implemented.

Another outcome of the study is that the resource-based approach allowed us to identify a clear set of policies and assumptions that should drive the country towards its goals. This helps to monitor and evaluate policies over time, a key success factor for the implementation of poverty reduction strategies [57]. The differences between actual and expected results can be traced back to specific assumptions, and corrective actions can be taken. At the moment this document is going to press, growth estimations for Mali are available for the years 2006-2009: the average growth rate for the last four years averaged 4.7%, indicating an encouraging acceleration towards the country's development goals with respect to the Business as Usual scenario.

In summary, we believe that our approach complements well the existing approaches to sustainable development policy analysis, enhancing the awareness of policy makers of the constraints to the country's development, and supporting them in defining better strategies and realistic goals within those constraints. Further research is required in Mali and other countries in the region to improve this approach and test it in other settings.

Acknowledgements. We would like to thank the Carter Center, and in particular Elaine Geyer Allely for providing financial and technical support to this study. We also would like to thank the technical team of the CPM team in Bamako, and in particular Siaka Sanogo and Karounga Camara, for their collaboration.

References

1. WB, World Development Indicators 2007. World Bank: Washington D.C. (2007)
2. UNDP, Human Development Report 2006. United Nations Development Program, New York (2006)
3. Senge, P.: The Fifth Discipline: The art and practice of the learning organization. Doubleday, New York (1990)
4. WB-IMF, Poverty Reduction Strategy Papers—Progress in Implementation. In: Nankani, G., Allen, M. (eds.) World Bank and International Monetary Fund, Washington D.C. (2004)
5. GoM, Cadre Stratégique pour la Croissance et la Réduction de la Pauvreté. Government of Mali, Ministère de l'Economie et des Finances (2006)
6. GoM, Poverty Reduction Strategy Paper. Government of Mali, Ministry of Economy and Finance (2002)

7. Barney, G.O.: The Global 2000 Report to the President and the Threshold 21 model: influences of Dana Meadows and system dynamics. System Dynamics Review 18(2), 123–136 (2002)
8. Tonn, B., Hemrick, A., Conrad, F.: Cognitive representations of the future: Survey results. Futures 38, 810–829 (2006)
9. Penrose, E.: The Theory of the Growth of the Firm. Oxford University Press, Oxford (1959)
10. Wernerfelt, B.: A Resource-Based View of the Firm. Strategic Management Journal 5(2), 171–180 (1984)
11. Barney, J.: Firm Resources and Sustained Competitive Advantage. Journal of Management 17(1), 99–120 (1991)
12. Peteraf, M.A.: The Cornerstones of Competitive Advantage: A Resource-Based View. Strategic Management Journal 14(3), 179–191 (1993)
13. Warren, K.: Competitive Strategy Dynamics. John Wiley & Sons Ltd., Chichester (2002)
14. Black, J.: A Dictionary of Economics. Oxford University Press, Oxford (2002)
15. Robinson, S., Yunez-Naude, A., Hinojosa-Ojeda, R.: From stylized to applied models: Building multisector CGE models for policy analysis. The North American Journal of Economics and Finance 10(1), 5–38 (1999)
16. Pedercini, M.: Potential Contribution of Existing Computer-Based Models to Comparative Assessment of Development Options. Working papers in System Dynamics. University of Bergen, Bergen (2003)
17. Arango, S.: Simulation of alternative regulations in the Colombian electricity markets. Socio-Economic Planning Sciences 41(4), 305–319 (2007)
18. Parayno, P.P., Saeed, K.: The Dynamics of Indebtedness in the Developing Countries: The Case of the Philippines. Socio-Economic Planning Sciences 27(4), 239–255 (1993)
19. Saeed, K.: A Re-evaluation of the Effort to Alleviate Poverty and Hunger. Socio Economic Planning Sciences 21(5), 291–304 (1987)
20. Qureshi, M.A.: Challenging Trickle-down Approach: Modelling and Simulation of Public Expenditure and Human Development - The Case of Pakistan. International Journal of Social Economics 35(4), 269–282 (2008)
21. Forrester, J.W.: Industrial Dynamics. Productivity Press, Cambridge (1961)
22. Pedercini, M., Barney, G.O.: Dynamic Analysis of Millennium Development Goals (MDG) Interventions: The Ghana Case Study (forthcoming)
23. Börjeson, L., et al.: Scenario types and techniques: Towards a user's guide. Futures 38, 723–739 (2006)
24. Cobb, C.W., Douglas, P.H.: A Theory of Production. The American Economic Review 18(1), 139–165 (1928)
25. Sacerdoti, E., Brunschwig, S., Tang, J.: The Impact of Human Capital on Growth: Evidence from West Africa. IMF Working Paper. International Monetary Fund, Washington D.C. (1998)
26. Calderón, C., Servén, L.: The Effects of Infrastructure Development on Growth and Income Distribution. World Bank Working Paper. World Bank, Washington D.C. (2004)
27. Jul-Larsen, E., et al.: Socio-Economic Effects of Gold Mining in Mali: A Study of the Sadiola and Morila Mining Operations. CMI Report, Chr. Michelsen Institute (2006)
28. Birdsall, N.: Economic Approaches to Population Growth. In: Chenery, T.N.S.H. (ed.) Handbook of Development Economics, pp. 478–542. Elsevier Science Publishers B.V., Amsterdam (1988)
29. Rodgers, G.B.: Income and Inequality as Determinants of Mortality: An International Cross-Section Analysis. Population Studies 33(2), 343–351 (1979)

30. Coale, A.J., Demeny, P.: Regional Model Life Tables and Stable Population, 2nd edn. Academic Press, New York (1983)
31. Essama-Nssah, B.: The Poverty and Distributional Impact of Macroeconomic Shocks and Policies: A Review of Modeling Approaches. World Bank Policy Research Working Paper. World Bank, Washington D.C. (2005)
32. Qu, W., Barney, G.O.: A Model for Evaluating the Policy Impact on Poverty. In: Proceedings of the 20th International Conference of the System Dynamics Society. The System Dynamics Society, Palermo (2002)
33. FAO, AQUASTAT, Food and Agriculture Organization of the United Nations
34. EIA, Energy Information Administration
35. Monfreda, C., Wackernagel, M., Deumling, D.: Establishing national natural capital accounts based on detailed Ecological Footprint and biological capacity assessments. Land Use Policy 21, 231–246 (2004)
36. DNSI-DNPD, Rapport sur la Situation Economique et Sociale du Mali en 2005 et Perspectives pour 2006. Direction Nationale de la Statistique et de l'Informatique - Direction Nationale de la Planification du Developpement, Government of Mali (2005)
37. IMF, International Financial Statistics Yearbook 2004. International Monetary Fund, Washington D.C. (2004)
38. IMF, Government Finance Statistics Yearbook 2004. International Monetary Fund, Washington D.C. (2004)
39. IMF, Balance of Payments Statistics Yearbook 2004. International Monetary Fund, Washington D.C. (2004)
40. UN, World Population Prospects: The 2002 Revision. United Nations Population Division, New York (2003)
41. FAO, FAOSTAT. Food and Agriculture Organization of the United Nations, Rome (2004)
42. UNDG, Indicators for monitoring the Millennium Development Goals. United Nations Development Group, New York (2003)
43. Sterman, J.D.: Appropriate summary statistics for evaluating the historical fit of system dynamics models. Dynamica 10(2), 51–66 (1984)
44. Barlas, Y.: Formal Aspects of Model Validity and Validation in System Dynamics. System Dynamics Review 12(3), 183–210 (1996)
45. Richardson, G.P.: Problems with Causal-Loop Diagrams. System Dynamics Review 2(2), 158–170 (1986)
46. Richardson, G.P.: Loop polarity, loop dominance, and the concept of dominant polarity. System Dynamics Review 11(1), 67–88 (1995)
47. Sachs, J.D., et al.: Ending Africa's Poverty Trap. Brookings Papers on Economic Activity 35(1), 117–240 (2004)
48. Ranis, G., Stewart, F., Ramirez, A.: Frances Stewart, Alejandro Ramirez, Economic Growth and Human Development. World Development 28(2), 197–219 (2000)
49. Bloom, D.E., Canning, D., Sevilla, J.: The Effect of Health on Economic Growth: Theory and Evidence. NBER Working Paper Series. National Bureau of Economic Research, Cambridge (2001)
50. Stifel, D., Minten, B., Dorosh, P.: Transaction Costs and Agricultural Productivity: Implications of Isolation for Rural Poverty in Madagascar. MSSD Discussion Paper 2003. International Food Policy Research Institute, Washington D.C. (2003)
51. Hausmann, R., Pritchett, L., Rodrik, D.: Growth Accelerations. NBER Working Paper. National Bureau of Economic Research, Cambridge (2004)

52. IMF, Mali: Fourth Review Under the Three-Year Arrangement Under the Poverty Reduction and Growth Facility and Request for Waiver of Performance Criteria. International Monetary Fund: Washington D.C. (2006)
53. N'Djim, H., Doumbia, B.: Case study: Mali Population and water issues. In: Sherbinin, A.B., Dompka, V. (eds.) Water and Population Dynamics: Case Studies and Policy Implications. American Association for the Advancement of Science, New York (1998)
54. WB, World Development Report 2005: A Better Investment Climate for Everyone. World Bank, Washington D.C. (2005)
55. Dollar, D., Hallward-Driemeier, M., Mengistae, T.: Investment climate and firm performance in developing economies. Economic Development and Cultural Change 54(1), 1–31 (2005)
56. Warren, K.: Strategic Management Dynamics. John Wiley and Sons, Chichester (2008)
57. WB-IMF, Review of the PRS Approach: Balancing Accountabilities and Scaling Up Results. World Bank and International Monetary Fund, Washington D.C. (2005)

Using System Dynamics to Assess the Role of Socio-economic Status in Tuberculosis Incidence

Marisa Analía Sánchez

Dpto. de Ciencias de la Administración, Universidad Nacional del Sur,
Bahía Blanca, Argentina
mas@uns.edu.ar

Abstract. Tuberculosis is one of the diseases that generate more mortality in recent years. Recent research on the impact of DOTS programs for tuberculosis control suggest that, after several years of successful implementation, the incidence is not decreasing as expected. Globally and in most regions, the prevalence and mortality decay, but not quickly enough to achieve the Millennium goals set by the WHO. Many socio-economic determinants and the exposure of the population to risk factors have a major impact on the incidence of tuberculosis. The aim of this paper is to develop a conceptual model based on the dynamics of the tuberculosis epidemiology and its relationship to socio-economic determinants and risk factors. The model will aid in understanding the causes of undesired behavior and designing new policies to eliminate or mitigate them. The work includes results of simulations and projections for Jujuy region of Argentina.

Keywords: tuberculosis, social determinants, risk factors, system dynamics, simulation.

1 Introduction

Health policies do not always achieve their results because of the complexity of both the environment and the policy making process. To solve problems in the area of public health it is necessary to gain an understanding of the systems involved and to adopt a transdisciplinary perspective. Leischow (2008) stresses that the increasing emphasis on a transdisciplinary, translational, and network based science reflects the recognition that most of the causes of disease is multifactorial, dynamics and nonlinear [1]. Transdisciplinary research is defined as the process by which a team of stakeholders from different fields work together for extensive periods of time to develop a conceptual and methodological framework that integrates and transcends their respective disciplinary perspectives [2].

Systems Dynamics modeling is a tool to analyze the dynamic complexity of long-term public health policies. Dynamic problems require a continuous management and monitoring. In the specific context of the management and development of policies, dynamic problems are those of persistent, chronic and recurring nature [3]. When management actions are taken, the results are observed, evaluated and new actions are taken, producing new results, observations and actions. In this way, most of the

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 464–475, 2011.

problems of dynamic management are problems of feedback. The feedback cycle occurs between control activities, the system, and between the components of the system. System dynamics emerged in the mid-1950s as a proposal to consider complex dynamic systems. The fundamental ideas were developed by Jay Forrester at MIT and it postulates that the behavior of such systems results from the underlying structure of flows, delays, and feedback loops [4]. This view is consistent with the systemic thinking Senge calls the "fifth discipline" and considers the dynamics of systems as part of the learning organization [5].

The prevention and control of tuberculosis (TB) fit within the problems requiring a systemic treatment. It's one of the diseases that generate more mortality in recent years. The international standard for tuberculosis control is the World Health Organization's DOT (Direct Observation of Therapy) strategy that aims to reduce the transmission of the infection through prompt diagnosis and effective treatment of symptomatic tuberculosis patients who present at health care facilities. The treatment is based on the strict supervision of medicines intake. Recent research on the impact of DOTS programs for tuberculosis control suggest that, after several years of successful implementation, the incidence is not decreasing as expected. Globally and in most regions, the prevalence and mortality decay, but not quickly enough to achieve the Millennium goals set by the WHO. From the estimated annual incidence of 9.1 million cases of tuberculosis, only 5.27 million have been notified. This means that approximately 40 per cent was not detected or notified to DOTS programs [6]. In this context the WHO stresses the urgent need to accelerate efforts in the early detection and to provide a health service of high quality.

Many socio-economic determinants (such as poverty, education, or access to health services) and the exposure of the population to risk factors (e.g., tobacco, HIV, diabetes) have a major impact on the incidence of tuberculosis. It is therefore necessary to determine causal relationships and key areas in which policymakers should focus resources. Lönnroth [7] observes that one avenue for improved TB prevention is through actions to address the social determinants of TB, as well as the more proximate risk factors (the physical and biomedical factors that directly influence the mechanisms that govern exposure to tuberculosis, risk of acquiring tuberculosis infection, and risk of progression from tuberculosis infection to active tuberculosis disease). The aim of this paper is to develop a conceptual model based on the dynamics of the tuberculosis epidemiology and its relationship to socio-economic determinants and risk factors. The model will aid in understanding the causes of undesired behavior and designing new policies to eliminate or mitigate them. The work includes results of simulations and projections for Jujuy region of Argentina.

The rest of this work is structured as follows: Section 2 introduces the use of System Dynamics in public health; provides a brief overview of the proposals to model the dynamics of tuberculosis and a discussion about their limitations to quantify the effect of socio-economic status and risk factors. Section 3 presents and explains the proposed model. Then, Section 4 shows simulation results. Finally, Section 5 presents conclusions.

2 System Dynamics Modeling in Public Health

System Dynamics modeling was developed by Jay W. Forrester and has gained relevance in recent years because of the need to model complex systems. System Dynamics is a methodology to model the forces of change in a complex system so that their influences can be better understood. There is a tradition in the use of dynamic simulation to study problems in the social sciences. Currently, it is used in public health [8], [9], social welfare [10], sustainable development [11], security [12], among many others. The methodology is iterative, allows various stakeholders to combine their knowledge of a problem in a dynamic hypothesis and then, using computer simulation, formally compare various scenarios on how to lead change [13]. The emphasis of system dynamics is not to forecast the future, but in learning how the actions in the present can trigger reactions in the future [5]. Even though it is not possible to determine with some degree of certainty the value of constants or change rates, the model is used as a learning tool to determine causal paths and relevant factors.

Simulation and system dynamics introduce laboratory conditions in the social sciences research. Ghaffarzadegan [14] emphasizes that the problems associated with the formulation of public policies have features that prevent its resolution with traditional techniques different to simulation, namely resistance of environmental policy; need to experiment and the cost of experience; need to persuade different stakeholders; extreme confidence of policymakers; and need to analyze from an endogenous perspective. Policy resistance occurs when policy actions trigger feedback from the environment that undermines the policy and at times even exacerbates the original problem. This situation is common when there are long delays between the actions and the outcomes. The topic of extreme confidence in management has been treated by numerous investigations. Briefly, it can be noted that too much confidence combined with the complexity of public policy, where many times the benefits are not perceived through intuition, can be a serious obstacle in the formulation of policies. The need to have an endogenous perspective refers to the ability to recognize (both at individual and organizational levels) which events are the results of internal factors. Senge refers to this problem in terms of organizational learning [5].

A central aspect of system dynamics is that complex behavior of organizations and social systems are the result of the accumulation - of people, materials, or financial assets - and balance and feedback mechanisms. The first step to develop a dynamic model is to develop a hypothesis explaining the cause of a problem and define a Causal Loop Diagram. The diagram is a tool to analyze the problem, to then define a formal model using a set of differential equations that can be analyzed mathematically to determine conditions of convergence. In addition, the model is used to develop simulations that allow numerical experiments and analyze scenarios.

Numerous authors highlight the contribution of mathematical modeling in epidemiology [15]. In the case of tuberculosis there are mathematical models to study the dynamics of tuberculosis, the spread of HIV, the appearance of multi-drug-resistance tuberculosis, among others. In [16] the authors group models by their structure: SEIR-type models, age-structured and delayed models, and spatially structured models. In SEIR-type models the population is categorized in one of the states of susceptible, exposed, infectious or recovered. One of the main attributes of these models is that

the infection rate is a function of the number of infectious individuals at an instant t, and then is a non-linear term. Some proposals differ in the way they represent progression of a latent infection to active disease [17], [18], [19], [20] [21], [22]. Another issue among TB models is whether, and how, they choose to represent re-infection. Age-structured models have been developed to explore TB control under DOTS strategy [23], or vaccination strategies analysis [24], [25]. Several TB models focus on the burden of multi-drug-resistance TB [26], [27], [28]. In [29] the authors explore the impact of HIV on TB epidemics which is a topic of great importance in Africa.

In the majority of these models there is a threshold behavior to determine under what conditions an equilibrium state is achieved which indicates that an epidemic is controlled. However, with the aim of making these models tractable, simplifications and somewhat unrealistic assumptions are made, that greatly affect the determination of the equilibrium point of the system.

In the works previously mentioned the impact of risk factors and socio-economic status is not incorporated explicitly. The impact is incorporated implicitly when giving values to the rate of infection, the number of contacts, the length of treatment, etc. Thus, while it is possible to use the model to make projections, it is not possible to analyze and quantify the impact of risks. To the best of our knowledge, only [8] proposes a holistic view of health care. The work examines the interactions between prevalence, adverse living conditions, and the ability of the community to act. It is not developed to any disease in particular; in any case, sensitivity analysis shows the effect of different health care actions in the short and long term.

3 Model Proposal

In the DOTS Expansion Working Group report [30], the WHO includes a conceptual framework for a better early detection of cases of tuberculosis. The model describes processes from infection to notification and explicit delays and other obstacles to the health system. The model puts in context the role of the components of the Stop TB strategy. Unlike early models of the strategy, this proposal makes explicit the factors that interfere with the implementation of the strategy.

This model is one of our sources of inspiration to define an epidemiological model of the dynamics of tuberculosis that allows analyzing the impact of risk factors and social determinants in the incidence of the disease. Our proposal allows studying the impact of different health strategies from a holistic perspective. In what follows, we describe some risk factors and social determinants included in our model.

3.1 Social Determinants and Risk Factors

In [31] the authors analyze the major risk factors related to tuberculosis and provide an estimate of the relative risk of developing tuberculosis (see Table 1).

The impact of risk factors changes for different regions. For example, HIV and malnutrition are very important in Africa; while diabetes has more relevance in Europe or United States. Projections indicate that the prevalence of diabetes will increase.

Table 1. Relative risks of active tuberculosis

	Relative risk for active TB
HIV	26.7
Malnutrition	3.2
Diabetes	3.1
Alcohol	2.9
Active smoker	2.6
Indoor air pollution	1.5

In [31] the authors quantify the risk of death for patients of tuberculosis associated with demographic and clinical factors in fifteen European countries. One of the main findings is that the elderly and resistance to isoniazid and rifampicin were the main determinants of death. The authors conclude that given the high risk of death with multi-drug-resistance TB (MDR-TB), a strict adherence to prescribed treatment regimens, the early testing for susceptibility to drugs, and the proper use of medication are crucial to prevent the growth and spread of drug resistance.

Other risk factor addressed in the literature is alcohol. In [31] the authors highlight the presence of a social pattern that includes a high exposure to alcohol and to environments with high risk of infection; and the increased risk to the activation of the disease due to direct and indirect effect of alcohol on the immune system. The indirect effects are manifested with disorders introduced by alcohol such as poor nutrition, chronic diseases, etc.

In [32] the authors provide a detailed analysis of the various factors interplaying to affect the health-seeking behavior and timely treatment. The study was conducted in seven countries of the WHO Eastern Mediterranean Region. They categorize the factors as either patient or health systems delays. The main determinants of delay were socio-demographic (illiteracy, suburban residence); economic; stigma; time to reach the health facility; seeking care from non-specialized individuals; and visiting more than one health care provider before diagnosis. Delay in diagnosis results in increased infectivity in the community and may also lead to a more advanced disease state at presentation, which contributes to late sequel and overall mortality [32].

Below, we present a model that includes all these issues to understand how they combine to define an undesired state of the evolution of the disease.

3.2 Model Components

Fig. 1 presents the model using a Causal Loop diagram. It shows the interactions between the states of patients with tuberculosis, overcrowding, illiteracy, stigma, and the obstacles to access the health system, the impact of strategies such as DOTS, and risk factors such as diabetes or HIV. The model consists of causal relationships. The causal relationship $x \rightarrow y$ means that the input variable x has some causal influence on the output variable y. A positive influence means "a change in x, being the rest of variables unchanged, causes y to change in the same direction". The symbol + indicates a positive causality. On the other hand, a negative influence means "a change in

x, being the rest of variables unchanged, causes y to change in the opposite direction".

The system is dominated by a cycle of positive feedback that can be avoided decreasing adverse conditions and increasing primary prevention. In this way, the population is less exposed to the disease. Then, once in the "circle of disease", it is not feed improving screening (search for the disease), access delay, health education, or DOTS coverage.

Based on this conceptual framework we defined a model using a system of differential equations. The population is divided into the following epidemiological classes or subgroups: susceptible to infection (Susceptible); exposed to drug-sensitive TB (Latent DSTB); exposed to multi-drug-resistant TB (Latent MDRTB); exposed to TB after treatment (Latent after Treatment); infectious population (Infectious DSTB, Infectious Clinical DSTB, Infectious Detected DSTB, Infectious First Line Treatment, Infectious MDRTB, Infectious Second Line Treatment MDRTB, Infectious MDRTB after Treatment, and Persistent MDRTB); and cured. We allow re-infection to move individuals from the cured class to the latent classes. The most significant difference of this model with respect to others is the distinction among different subgroups for the active tuberculosis class. This allows representing the effect of delays since the individual is infectious until he starts treatment.

We assume that an individual can become infected through contact with other infectious individuals. $Beta$ is the average number of susceptible and treated individuals, infected by an infectious individual per contact per unit of time; *Contacts* is the contact rate per-capita; K is the rate by which an individual leaves the latent class and becomes infectious; and p represents the re-infection level. To incorporate the impact of HIV (or diabetes mellitus) we proceed in the following way. Given the relative risk for active TB, RR_{HIV} (RR_{DM}), and the percentage of population with HIV, $Fraction_{HIV}$ ($Fraction_{DM}$), let define the rate of activation as:

$$(p \cdot Contacts \cdot Beta \cdot Latent\_DSTB \cdot ((Total\_Infectious) \,/\, N)\,) + (Latent\_DSTB \cdot Fraction\_HIV \cdot RR\_HIV \cdot k) + (Latent\_DSTB \cdot (1 - Fraction\_HIV\,) \cdot k). \quad (1)$$

The model makes explicit the main contributors to delays: illiteracy, suburban residence, stigma, time to reach health facility, and adherence to DOTS. In [32] the authors use multivariate logistic regression analysis to adjust for the confounding effect of several identified determinants of diagnostic and treatment delay of tuberculosis patients. Cases with longer delays were categorized as “cases” while with shorter delays were considered as “controls”. The cut-off point for “longer” delay was defined according to the median value obtained for each country. For example, one of the significant risk factors for total delay in Egypt was being illiterate (AOR 2.76). This means that in the model the odds for a longer delay is 2.76 times higher than a shorter delay. The authors provide data for the minimum and maximum delays (in days) and a confidence interval for the adjusted odds ratio (AOR). Then, given the AOR value for a particular risk factor, we estimate the delay variation with respect to the mean. Finally, the delay rates in the model are adjusted using the estimated variation.

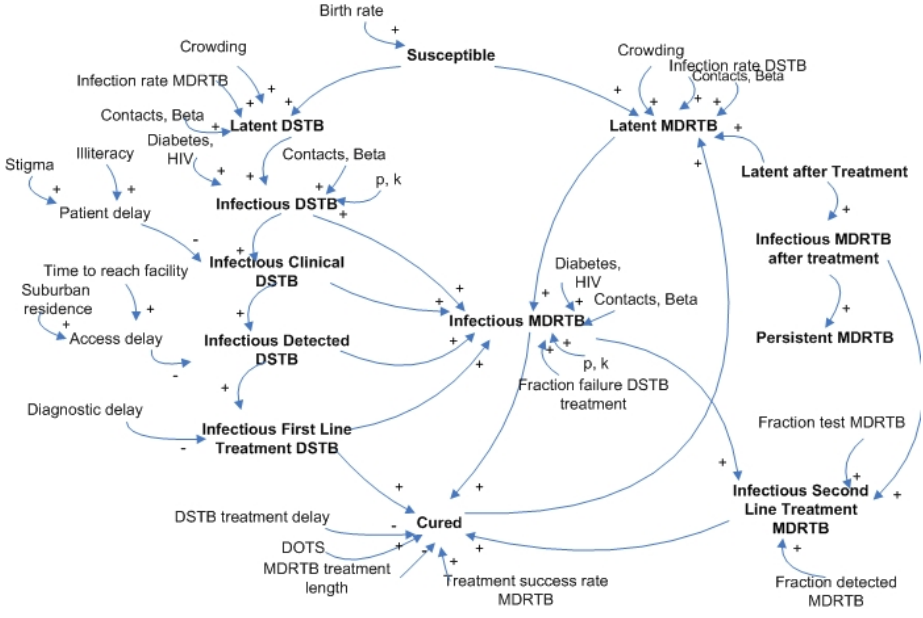
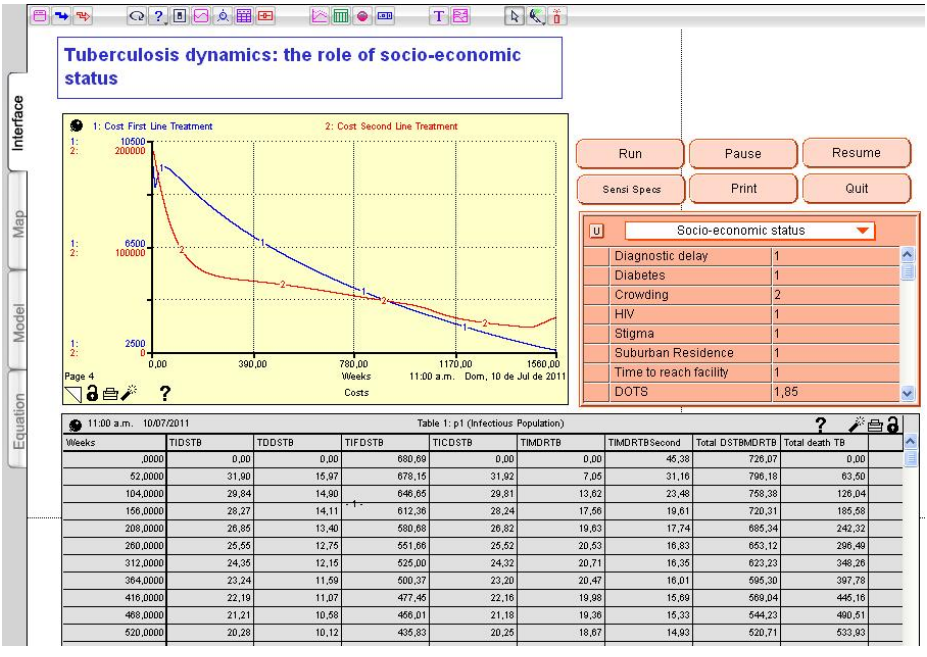


Fig. 1. Global model of the dynamics of tuberculosis

Tuberculosis is curable in most instances. Using combinations of first-line drugs around 90% of people with drug-susceptible TB can be cured in six months. Treatment of multidrug-resistant TB requires use of second-line drugs that are more costly and cause more severe side-effects, and recommended regimens must be taken for up to two years. Cure rates for MDR-TB are lower, ranging from around 50% to 70%. The WHO distinguishes between two types of resistance: acquired resistance (resistance among previously treated patients) and primary resistance (resistance among new cases). We formulate a quite basic transmission sub-model to study the dynamics of multi-drug resistant tuberculosis. However, it allows analyzing the effect (a) to scale up access to testing for resistance to first-line anti-TB drugs among TB patients; and (b) to scale up access to effective treatment for drug-resistant TB. The variable *Fraction test MDRTB* models the proportion of people who have been previously treated for TB that is tested for MDR-TB. The variable *Treatment success rate MDRTB* models the proportion of people who are successfully treated.

This model was translated into the continuous simulation language Stella™. Several simulation experiments were conducted to analyze scenarios and determine which factors are of greater impact on the prevalence of tuberculosis. Fig. 2 includes part of the interface of the simulator tool that was designed to facilitate the updating of parameters. The tool allows using the simulator for different populations and settings by modifying the values of parameters from the interface, and hence it becomes an experimental laboratory. In what follows, we present some results of the simulation.



Weeks	TIDSTB	TDDSTB	TIFDSTB	TICDSTB	TIMDRTB	TIMDRTBSecond	Total DSTBMDRTB	Total death TB	
,0000	0,00	0,00	680,69	0,00	0,00	46,38	726,07	0,00	
52,0000	31,90	15,97	678,15	31,92	7,05	31,16	796,18	63,50	
104,0000	29,84	14,90	646,65	29,81	13,62	23,48	758,38	126,04	
156,0000	28,27	14,11	612,36	28,24	17,56	19,61	720,31	185,58	
208,0000	26,85	13,40	580,68	26,82	19,63	17,74	685,34	242,32	
260,0000	25,55	12,75	551,66	25,52	20,53	16,83	653,12	296,49	
312,0000	24,35	12,15	525,00	24,32	20,71	16,35	623,23	348,26	
364,0000	23,24	11,59	500,37	23,20	20,47	16,01	595,30	397,78	
416,0000	22,19	11,07	477,45	22,16	19,98	15,69	569,04	445,16	
468,0000	21,21	10,58	456,01	21,18	19,36	15,33	544,23	490,51	
520,0000	20,28	10,12	435,83	20,25	18,67	14,93	520,71	533,93	

Fig. 2. Interface of the simulator tool

4 Simulation Results

The model is initialized with population data, and people with TB notified during the period 1985-2005 [33] for the Jujuy region of Argentina. Jujuy is a province of Argentina, located in the extreme northwest of the country, at the border with Chile and Bolivia. The pulmonary tuberculosis incidence rate is 154.15 (each 100.000 inhabitants) in 1985 and 63.01 in 2006 (the estimated national incidence is 24.11 in 2006).

We validated the model with data published by the National Institute of Respiratory Diseases of Argentina [33]. The main obstacle to calibrate the simulation model is the lack of statistical data to evaluate the relative risk of different socio-economic determinants. Therefore, we estimated values using regional data and values published in [32]. We simulated 20 years to calibrate the model parameters ($R^2 = 0.837$). Then, different scenarios are analyzed for the next 10 years. We initialize the variables that represent different classes with the resulting final values resulting from the model calibration. The integration method is Runge-Kutta 4 with a weekly step.

The results of the simulation showed that the prevalence of tuberculosis is very sensitive to delays that occur since an individual becomes infectious until the disease is diagnosed and he begins treatment. Delays contribute to increase the infectious population, which increases the likelihood of infectious contacts and reinforces the cycle of the disease. Hence, a slight increase in relative risk factors such as illiteracy, stigma, time to reach facility, or suburban residence may threaten the targets for 2015.

In what follows, we see the effect certain variables have in the model while holding other assumptions to their baseline value. Fig. 3 shows results for the relative risk of time to reach facility assuming values of 1 (the delay is equal to the mean delay), 2 or 3 (the delay is 2 or 3 times the mean delay). Observe that in 30 years the total infectious population maybe 225.76; 293.07.88; or 377.88.

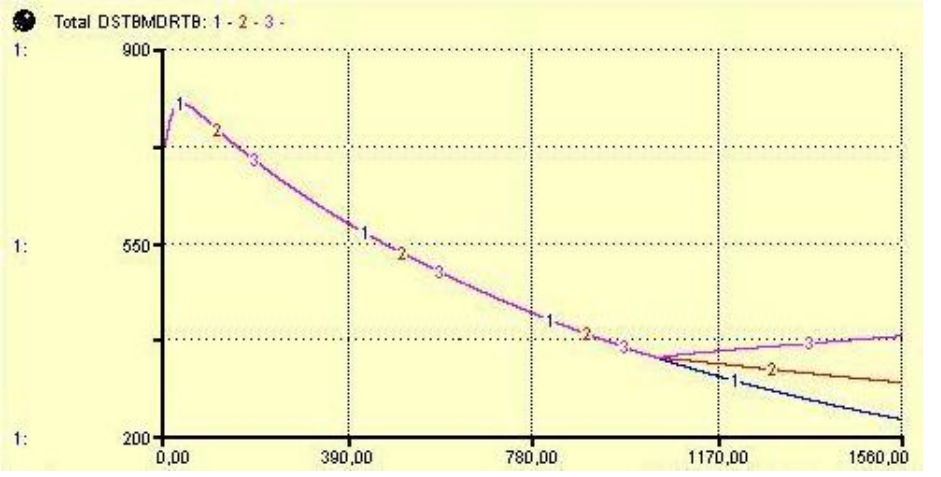


Fig. 3**.** Sensitivity analysis to assess the impact of the Relative Risk of Time to reach facility (RRTF) on the infectious population (1:RRTF=1; 2:RRTF=2; 3:RRTF=3)

The need to scale-up the diagnosis and effective treatment of MDR-TB was clearly recognized in the Global Plan to Stop TB 2006-2015. The target is that the incidence of MDR-TB should be declining by 2015. To achieve this goal, there are six major objectives. For the purpose of showing the simulator performance, we consider two of these objectives.

The first objective we consider is to scale up access to testing for resistance to first-line anti-TB drugs among TB patients. We include simulation results for the variable *Fraction test MDRTB* assuming 7% (baseline 2009) and a gradual increment towards 100% (target 2015) [34]. However, the total infectious population shows a very slight decrement by 2015. Only by 2025 the infectious pool decreases in 4%. In this case, this is due because there are few MDRTB individuals.

Another WHO objective is to scale up access to effective treatment for drug-resistant TB. The treatment success rate among patients with confirmed MDR-TB should increase from the 2009 baseline of 60% to up to 75% by 2015. Simulation results show a decrease in the total number of infectious people of 3%.

The increase in the incidence of diabetes, is supposed to increase the risk of activating the disease, therefore, increases the value for the factor k (rate that an individual leaves the latent class and becomes infectious). The evolution of tuberculosis is very sensitive to these parameters.

5 Conclusions

This work describes the complexity of both the environment and the health policy making process. The tuberculosis prevention and control problem highlights the importance of considering causal relationships, and adopt an endogenous perspective to understand behavior. The paper relates results of various investigations, since the ideas of Jay Forrester, Senge until the latest proposals as transdisciplinary research. A brief overview of the proposals to model the dynamics of tuberculosis is given and a discussion about their limitations to quantify the effect of socio-economic determinants and risk factors.

The main contribution of this paper is the development of an epidemiological model of the dynamics of tuberculosis that allows analyzing the impact of risk factors and socio-economic status in the incidence of the disease. In particular, the model distinguishes among different subgroups for the active tuberculosis class. This allows to focus on the effects of delays, on quantifying the contributors of the delay, and on simulating the impact of reducing such delays on mitigating the TB incidence. Given the importance of these aspects on the success of the DOTS strategy, we propose a more realistic model of the evolution of the disease. In this way, the proposal is less reductionist and expected to be more effective to achieve robust solutions. Based on the model we developed a simulation tool that can be used for different populations and settings by modifying the values of parameters from the interface.

References

1. Leischow, S., Best, A., Trochim, W., Clark, P., Gallagher, R., Marcus, S., Matthews, E.: Systems Thinking to Improve the Public's Health. American Journal of Preventive Medicine 35(2S), 196–203 (2008)
2. Stokols, D., Hall, K., Taylor, B., Moser, R.: The science of team science: overview of the field and introduction to the supplement. American Journal of Preventive Medicine 35(2S), 77–89 (2008)
3. Barlas, Y.: System dynamics: Systemic Feedback Moeling for Policy Analysis. In: Knowledge for Sustainable Development - An Insight into the Encyclopedia of Life Support Systems, pp. 1131–1175. UNESCO-Eolss Publishers, Paris (2002)
4. Forrester, J.: Industrial Dynamics. Pegasus Communications, Massachutses (1961)
5. Senge, P.: The fifth discipline: the art and practice of the learning organization. Doubleday/Curency, New York (1990)
6. World Health Organization: Report of the Meeting of the DOTS Expansion Working Group. Engaging professional Associations in TB Control., Geneve (2009)
7. Lönnroth, K., Jaramillo, E., Williams, B., Dye, C., Raviglione, M.: Drivers of tuberculosis epidemics: The role of risk factors and social determinants. Social Science & Medicine 68, 2240–2246 (2009)
8. Horner, J., Hirsch, G.: American Journal of Public Health (96), 452–458 (March 2006)
9. Thompson, K., Duintjer Tebbens, R.: Using system dynamics to develop policies that matter: global management of poliomyelitis and beyond. System Dynamics Review 24(4), 433–449 (2008)

10. Zagonel, A., Rohrbaugh, J., Andersen, D.: Using simulation models to address "What if" questions about welfare reform. Journal of Policy Analysis and Management 23(4), 890–901 (2004)
11. Dudley, R.: A basis for understanding fishery management dynamics. System Dynamics Review 24(1), 1–29 (2008)
12. Bontkes, T.: Dynamics of rural development in southern Sudan. System Dynamics Review 9(1), 1–21 (1993)
13. Andersen, D., Richardson, G., Vennix, J.: Group model building: adding more science to the craft. System Dynamics Review 13(2), 187–201 (1997)
14. Ghaffarzadegan, N., Lyneis, J., Richardson, G.: Why and How Small Systems Dynamics Models Can Help Plicymakers: A Review of Two Public Policy Models. In: 26th International Conference of the System Dynamics Society, Athens (2008)
15. Chubb, M., Jacobsen, K.: Mathematical modeling and the epidemiological research process. European Journal Epidemiology 25, 13–19 (2010)
16. Colijn, C., Cohen, T., Murray, M.: Mathematical models of tuberculosis: accomplishments and future challenges. In: International Symposium on Mathematical and Computational Biology (2006)
17. Dye, C., Garnett, G., Sleeman, K., Williams, B.: Prospects for worldwide tuberculosis control under the WHO DOTS strategy Directly Observed Shortcourse therapy. The Lancet 367(9514), 938–940 (2006)
18. Blower, S., McLean, A., Porco, T., Small, P., Hopewell, P., Sanchez, M., Ross, A.: The intrinsic transmission dynamics of tuberculosis epidemics. Nat. Med. 1(8), 815–821 (1995)
19. Vynnycky, E., Fine, P.: Lifetime risks, incubation period, and serial interval of tuberculosis. Am. J. Epidemiol. 152(3), 247–263 (2000)
20. Cohen, T., Colijn, C., Finklea, B., Murray, M.: Exogenous Re-Infection and the Dynamics of Tuberculosis Epidemics: Local Effects in a Network Model of Transmission. J. R. Soc. Interface 4(14), 523–531 (2007)
21. Feng, Z., Castillo, C., Capurro, A.: A Model for Tuberculosis with Exogenous Reinfection. Theoretical Population Biology (57), 235–247 (2000)
22. Blower, S., Small, P., Hopewell, P.: Control Strategies for Tuberculosis Epidemics: New Models for Old Problems. Science 273(5274), 497–500 (1996)
23. Castillo-Chavez, C., Feng, Z.: Global stability of an age-structure model for tuberculosis and its applications to optimal vaccination strategies. Math. Biosci. 151(2), 135–154 (1998)
24. Gomes, M., Franco, A., Medley, G.: The Reinfection Threshold Promotes Variability in tuberculosis Epidemiology and Vaccine Efficacy. Proc. Biol. Sci., 617–623 (2004)
25. Vinnicky, E., Fine, P.: The Natural History of Tuberculosis: The Implications of Age-dependent Risks of Disease and The Role of Infection. Epidemiol. Infect. (119), 183–201 (1997)
26. Cohen, T., Sommers, B., Murray, M.: The effect of drug resistance on the fitness of mycobacterium tuberculosis. Lancet Infect. Dis. 3(1), 13–21 (2003)
27. Cohen, T., Murray, M.: Modeling Epidemics of Multidrug-resistant Tuberculosis of Heterogeneous Fitness. Nat. Med. 10(10), 1117–1121 (2004)
28. Blower, S., Chou, T.: Modeling the emergence of the "hot zones": tuberculosis and the amplification dynamics of drug resistance. Nat. Med. 10(10), 1111–1116 (2004)
29. Currie, C., Williams, B., Cheng, R., Dye, C.: Tuberculosis epidemics driven by HIV: is prevention better than cure? AIDS 17(17), 2501–2508 (2003)

30. World Health Organization: Report of the Meeting of the DOTS Expansion Working Group. Engaging professional Associations in TB Control., Geneve (2009)
31. Lönnroth, K., Raviglione, M.: Global Epidemiology of Tuberculosis: Prospects for Control. Semin. Respir. Crit. Care Med. (29), 481–491 (2008)
32. World Health Organization. Regional Office for the Eastern Mediterranean: Diagnostic and treatment delay in tuberculosis, Cairo (2006)
33. Instituto Nacional de Enfermedades Respiratorias: Notificación de Casos de Tuberculosis en la República Argentina. Período 1980-2006. PRO.TB.Doc.Tec. 07/07 (2007)
34. World Health Organization: The Global Plan to Stop TB 2011-2015, Geneve (2010)

Energy Consumption and CO_2 Emissions of Beijing Heating System: Based on a System Dynamics Model

Hefeng Tong and Weishuang Qu

Institute of Scientific & Technical Information of China, Beijing, China
2111 Wilson Blvd. Suite 700, Arlington VA 22201, USA
thf2003@istic.ac.cn, wq@millennium-institute.org

Abstract. Beijing is a typical North China city, and it uses about 15–18% of its total energy consumption for heating. The building construction industry is also a key source of CO_2 emissions. This article, based on a system dynamics model, aims to simulate and forecast Beijing's energy consumption and CO_2 emissions under different scenarios. Under the baseline scenario, the energy consumption of Beijing's heating system in 2030 will be 15.44 MTce and the corresponding CO_2 emissions will be 9.71 MT. Gas is the major energy source for heating systems, accounting for more than 60% of the energy used. In the less building scenario, the energy used for heating in 2030 is projected to be 13.91 MTce, 9.88% less than baseline scenario. The cumulative saving in energy used for heating will be 19.39 MTce, with CO_2 reductions of 12.38 MT. In the energy efficiency scenario, the energy consumed for heating in 2030 is projected to be 13.16 MTce, 14.73% less than baseline scenario. The cumulative saving in energy used for heating is projected to be 21.02 MTce, with a CO_2 reduction of 12.13 MT. Thus, to achieve greater energy savings, a combination of policy measures, from both the demand side (smaller residential properties) and the technology side is needed.

Keywords: Heating System, Energy Consumption, CO_2 Emissions, System Dynamics.

1 Introduction

The building sector is a key area for energy consumption and CO_2 emissions, according to estimates by the United Nations Intergovernmental Panel on Climate Change's Fourth Assessment Report [1]. In 2004, the global construction industry's direct emissions of greenhouse gases (excluding emissions from electricity) were about 5 billion ton. Heating for buildings is a major part of the energy consumption and CO_2 emissions in this sector.

China is one of the largest construction markets in the world. In 2004, the total construction area of buildings in China was 38.9 billion m^2, and the energy consumption in buildings was about 510 MTce (Million ton coal equivalent), accounting for 25.5% of the total energy consumption of the entire country. CO_2 emissions from the building sector were about 1.25 billion ton. Heating is necessary in winter in northern China.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 476–485, 2011.

The energy consumed to heat urban buildings accounts for nearly 40% of the total energy consumption in these northern cities. In north China's urban buildings an area of approximately 6.4 billion m^2 is heated, and the energy consumed for heating is equivalent to about 130 MTce per year. The average amount of energy needed for heating per square meter per year is about 20 kgce (kilogram coal equivalent) [2]. There is considerable potential for savings in the amount of energy used for heating in China. As Beijing is a typical northern city in China, we chose it for our case study [3].

System dynamics (SD) is a feedback-based, object-oriented modeling paradigm which originated in work done by Forrester [4]. SD is used to model complex systems which are often nonlinear and are governed by feedbacks. It mainly focuses on the interrelationships rather than individual objects. In the feedback loops of a SD model, a change in one variable affects other variables in the system over time, which in turn affects the original variable. Establishing all these relationships correctly and explicitly is helpful for understanding complex systems and improves the understanding of the behavior of the system. The SD model can simulate the short- and long-term consequences of alternative policies and permits easy comparison with the reference scenarios. It also supports advanced analytical methods, such as sensitivity analysis and optimization. SD has been applied to many different complex industrial, economic, social, and environmental systems, to simulate situations related to policy analysis, economics, biology, energy policy, medicine, industrial engineering, urban planning, climate change, water resources systems, and the air travel industry [5-9]. An urban heating system is a typical complex system, so we chose SD as our methodology.

2 Case study of Beijing City

Beijing is located in the North China Plain and has a typical continental climate. In 2008, the average temperature in Beijing was 13.4 ℃. Heating is used in Beijing from November 15 to March 15, a period of about 4 months. With the improvement in living standards, the period during which heating is used has been extended, and in some areas in Beijing, it has reached 5–6 months. Key factors determining the energy consumption for urban heating and CO_2 emissions include: population size, per capita living area, temperature setting, energy price, and energy type.

In 2000, the total population in Beijing was 13.63 million, and in 2007 it reached 16.33 million, an increase of 19.76% in seven years. According to the Beijing Statistical Yearbook 2010, 8.9 MTce, about 17.3% of Beijing's total consumption of energy, was consumed to heat buildings in 2004. At the end of 2004, the total area of Beijing being heated was 428.5 million m^2, including 134.39 million m^2 heated by gas, 180.3 million m^2 heated by coal, and 84.87 million m^2 heated by central heating (using heat from power plants), and some other sources of energy (electricity and oil). In Beijing, coal and natural gas are the main sources of heating energy, the other sources are electricity, its byproduct (heat), and oil. There is a very small amount of heating from geothermal, solar, and other renewable energy sources.

3 Model Structure

3.1 Major Assumptions

Models are only valid under certain assumptions. For the sake of simplicity, we have made the following assumptions for this particular model:

1. Heating in the model refers to the heating of rooms. Domestic hot water and cooking are not included.
2. In the model, the building area only covers urban residential and public buildings. Rural residential and office buildings are not included.
3. In the model, there are five types of heating: the central heating network, coal-fired heating, gas heating, oil-fired heating and electrical heating. There are four types of heating energy resource: coal, gas, oil, and electricity. It should be noted that the central heating network has two energy sources: coal and gas.

3.2 Construction of the Model

We chose Vensim as our modeling tool. It is an integrated framework for conceptualizing, building, simulating, analyzing, optimizing, and applying models of complex dynamic systems. Vensim supports users in creating stock and flow diagrams to represent systems in greater structural detail. Stocks and flows can be imbedded in causal loops to clearly show the different components of the system. The simulation period used for the model was 2000–2030. For the period 2000–2007 we can compare the simulation results with the actual historical data.

As mentioned earlier, population size, living area per person, temperature setting, energy price and type of heating energy are the key factors in the urban consumption of energy for heating and CO_2 emissions. The plus or minus signs in Figure 1 indicate a positive or negative causal effect of the independent variable on the dependent variable. The loop between living area per person, total built-up area, total area being heated, and the consumption of heating energy is a negative feedback loop.

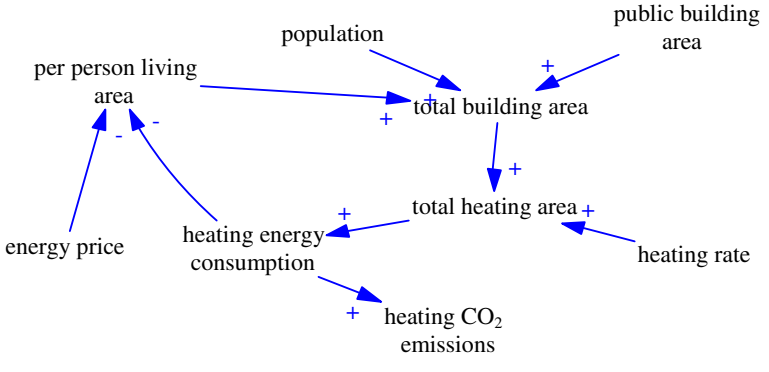


Fig. 1. Main causal relations of the model

The model includes four major subsystems: population, area of buildings, the energy consumption per meter squared, and heating. Variables in angle brackets, such as <PB>, <PPLA>, <TP> and <PSMCC> (known as shadow variables in Vensim) indicate that they are calculated in other modules and are then used in this module. The relationships between the variables are modeled by mathematical functions. Some of the influencing factors, such as temperature setting and energy price, were modeled exogenously, which makes the handling of the model less complex. The main equations of the heating subsystem and explanations of variables are given below:

1. $UP = TP \times UR$

Where UP is the urban population, TP is the total population, and UR is the urbanization rate.

2. $URB = UP \times PPLA$

Where URB denotes urban buildings, and PPLA is the per person living area.

3. $TBA = URB + PB$

Where TBA is the total building area, and PB denotes public buildings.

4. $THA = TBA \times RBH$

Where THA is the total heating area, and RBH is the ratio of buildings heated.

5. $BACC = THA \times ROCHS + THA \times RCHS \times ROCCICHS$

Where BACC is the building areas of coal consumption, ROCHS is the ratio of coal heating system, RCHS is the ratio of central heating system, and ROCCICHS is the ratio of coal consumption in central heating system.

6. $HCC = BACC \times PSMCC$

Where HCC denotes heating coal consumption, and PSMCC is the per m^2 coal consumption.

7. $THEC = HCC + HGC + HOC + HEC$

Where THEC is the total heating energy consumption, HGC is the heating gas consumption, HOC is the heating oil consumption, and HEC is the heating electricity consumption.

8. $TCE = THEC \times CEF$

Where TCE is the total CO_2 emissions, and CEF is the carbon emission factor (see Figure 2).

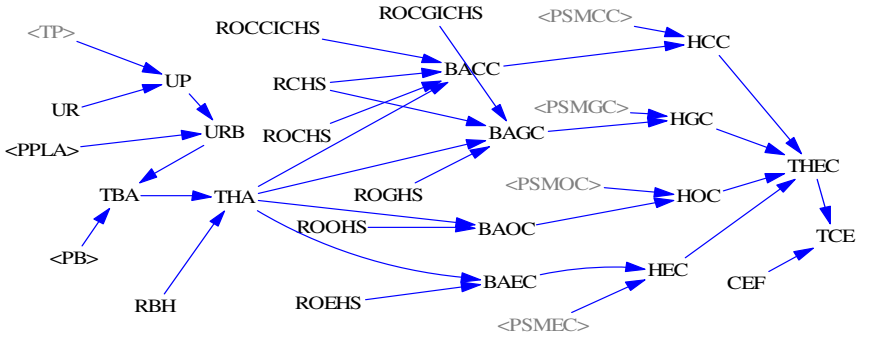


Fig. 2. Causal diagram for heating subsystem

3.3 Check on Model

First, the model generated a good fit to the historical data for the period 2000–2007. The variables selected for historical comparison are: total population, total consumption of primary energy, and total area of urban buildings. As shown in Table 1, most deviations between the historical data and simulation results are under 5%.

Table 1. Historical check of main variables

	Total population (M)			Total primary energy (M)			Total area of urban buildings (M)		
	H	B	D	H	B	D	H	B	D
2000	13.64	13.57	0.49%	42.02	40.90	−2.66%	291.09	291.09	0.00%
2001	13.85	13.91	0.44%	42.29	44.34	4.85%	377.17	379.24	0.55%
2002	14.23	14.24	0.07%	44.36	45.57	2.71%	404.83	428.88	5.94%
2003	14.56	14.58	0.11%	46.48	49.40	6.27%	431.22	462.34	7.22%
2004	14.93	14.94	0.11%	51.40	52.91	2.95%	465.23	489.40	5.20%
2005	15.38	15.35	−0.18%	55.22	56.23	1.83%	505.07	524.04	3.76%
2006	15.81	15.80	−0.08%	59.04	60.70	2.81%	542.82	567.14	4.48%
2007	16.33	16.26	−0.46%	62.85	63.55	1.12%	573.61	604.30	5.35%

H signifies historical data, B signifies baseline of simulation results, and D signifies the deviation between the first two. M signifies million.

After the historical test, we developed a sensitivity test (see Figure 3) to see how sensitive the model is to different assumptions. While it is easy to test for sensitivity by comparing a number of different simulations, this is often impractical when many parameters need to be tested together. Vensim has a sensitivity capability that makes it easy to run multivariate sensitivity simulations (Monte Carlo) and Latin hypercube simulations. You can simply select the parameters that you want to test, and the resulting variables that you want to see, and Vensim's sensitivity graph then will then show either the simulation traces or confidence bounds, as required.

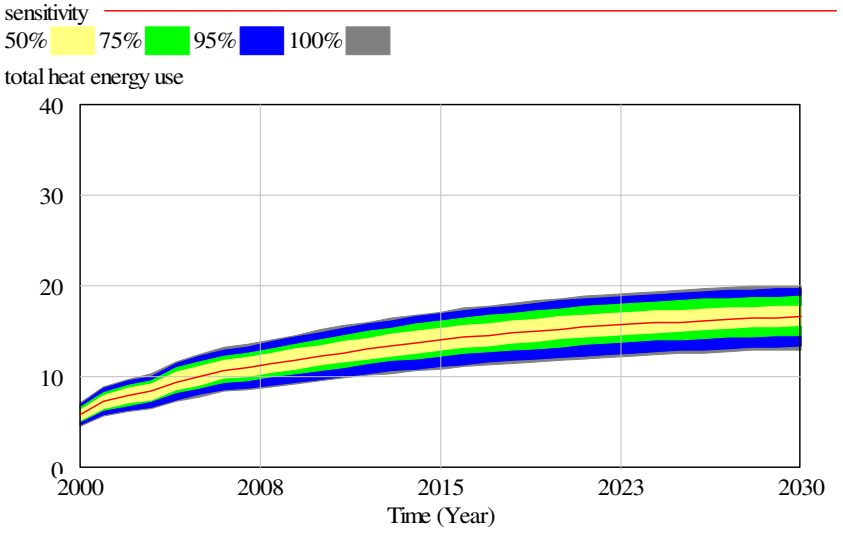


Fig. 3. Sensitivity testing of total energy used for heating

4 Scenario Analysis

In the model, all input variables can be changed to complete a simulation (or run a scenario). In order to have a clear picture of future heating energy consumption and CO_2 emissions, we selected several optimistic values to build our scenarios to assess future policy.

4.1 Baseline Scenario

The baseline scenario is based on the existing trends and policies for future development. In the baseline scenario (see Table 2), it is assumed that in the period 2008–2030, Beijing will continue its energy efficiency and conservation policies. In 2030, the total population of Beijing is projected to be 22.63 million, increasing by 66.78% over the 2000 figure. The total consumption of energy for heating is projected to be 15.44 MTce, increasing by 166.22% over the 2000 figure. The total emissions of CO_2 from heating are projected to be 9.71 MT, increasing 158.88% over the 2000 value. The rate of growth of CO_2 emissions is projected to be slower than that of energy consumption, as

in the future gas will have a larger share in the energy source while coal will have a lesser share, and the emission factor of gas is lower than that of coal. The proportion of Beijing's total consumption of energy that will be used for heating is projected to be 11.67% in 2030. The proportion of Beijing's CO_2 emissions in 2030 which will be due to heating is projected to be 5.67% of Beijing's total CO_2 emissions.

Table 2. Simulation results for baseline scenario

Variable	2010	2015	2020	2025	2030
TP (M)	17.63	19.70	21.28	22.28	22.63
THA (M)	704.00	904.23	1102.43	1293.89	1471.14
THEC (M)	12.23	13.87	14.90	15.42	15.44
TCE (M)	8.08	9.03	9.54	9.79	9.71

4.2 Less Building Scenario

In the period 2000–2007, Beijing's urban per capita floor space used for housing increased from 22.3 m^2 to 27.03 m^2, and the rural per capita living space increased from 28.91 m^2 to 39.54 m^2. Larger houses needs more energy for heating. Based on past trends, the urban and rural per capita housing floor space is projected to be 45 m^2 in 2030 in the base scenario. In the less building scenario, the urban and rural floor space for housing per capita in 2030 is projected to be 40 m^2.

In the less building scenario in 2030, the total population will not change. The total area being heated, energy consumption, and CO_2 emissions will be reduced (see Table 3). The total consumption of energy for heating is projected to be 13.91 MTce, 9.88% less than the baseline scenario. The total CO_2 emissions from heating are projected to be 8.75 MT, 9.88% less than the baseline scenario. The cumulative saving in energy consumption will be 19.39 MTce, and the cumulative reduction in CO_2 emissions will be 12.38 MT.

Table 3. Simulation results for less building scenario

Variable	2010	2015	2020	2025	2030
THA (M)	693.62	865.77	1031.10	1186.28	1325.74
THEC (M)	12.05	13.28	13.94	14.14	13.91
TCE (M)	7.96	8.65	8.93	8.98	8.75

4.3 Energy Efficiency Scenario

In the 'Eleventh Five Year Plan' (2006–2010) for energy efficiency in Beijing's buildings, the actual average energy consumption in 2010 is projected to be 10% lower than that in 2006, so the model set the yearly increase in energy efficiency at 2% for gas and coal. In the energy efficiency scenario, we set a more ambitious goal for energy efficiency: a 3% yearly increase in energy efficiency for gas and coal.

In the energy efficiency scenario, the total population and total area being heated are not projected to change in 2030. The total consumption of energy for heating and CO_2 emissions are projected to be reduced. The total consumption of energy for heating will be 13.16 MTce, 14.73% less than the baseline scenario. Total CO_2 emissions from heating are projected to be 8.43 MT, 13.23% less than the baseline scenario. The cumulative saving in energy consumption is projected to be 21.02 MTce, and the cumulative reduction in CO_2 emissions is projected to be 12.13 MT.

Table 4. Simulation results for energy efficiency scenario

Variable	2010	2015	2020	2025	2030
THEC (M)	12.19	13.55	13.99	13.80	13.16
TCE (M)	8.05	8.84	9.01	8.86	8.43

5 Conclusions

In the above analysis we have selected two optimistic scenarios to achieving lower energy consumption and CO_2 emissions than the baseline scenario, as shown in Figures 4 and 5, respectively.

During the first period (2010–2020), the less building scenario produces better reductions than the energy efficiency scenario, but during the second period (2020–2030), the energy efficiency scenario shows a stronger reduction. For the entire period, the technology scenario performs more efficiently.

Some other factors not included in this paper could have a substantial impact on energy and carbon reduction. Such factors include: population, energy price, and employment. Each of these factors includes multiple sub-factors. For instance, population is influenced by many factors, including: urban population size, total fertility rate, and life expectancy at birth. The model includes these factors endogenously so that the policy analysis can be expanded when necessary.

It seems that the following three points may be especially helpful for Beijing to achieve the goal of energy and carbon reduction:

- Adopt a combination of policy measures, on both the demand side (such as residential areas) and the technology side.
- Improve the building code to require higher energy efficiency and a longer lifespan for buildings. More expensive and valuable buildings could also encourage sustainable living, maintaining the quality with a smaller living area.
- Raise the public awareness of sustainable living, including slower growth in the number of living spaces.

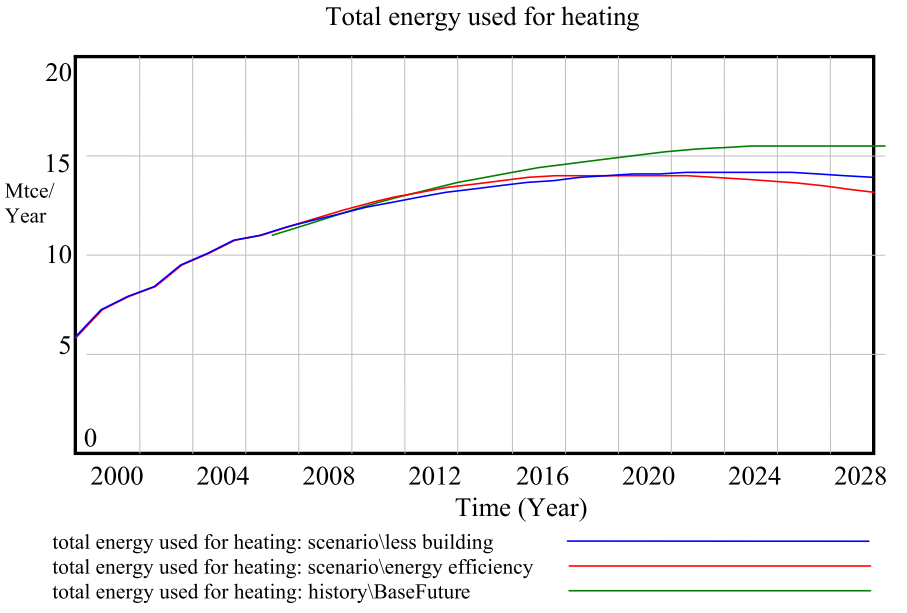


Fig. 4. Comparison of scenarios for total amount of energy used for heating

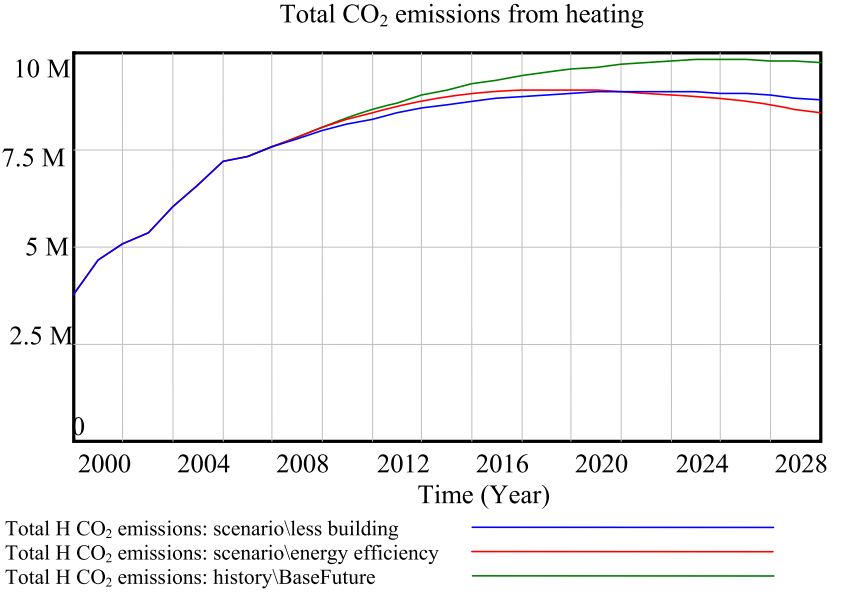


Fig. 5. Sensitivity testing of total energy used for heating

References

1. IPCC (Intergovernmental Panel on Climate Change): Climate Change 2007: Mitigation of Climate Change. United Kingdom: Cambridge University Press (2007)
2. THUBERC: Development of China Building Energy Efficiency Annual Report 2008. China Building Industry Press, Beijing (2009)
3. Zhao, J., Zhu, N., Wu, Y.: Technology line and case analysis of heat metering and energy efficiency retrofit of existing residential buildings in northern heating areas of China. Energy Policy 37(6), 2106–2112 (2009)
4. Forrester, J.W.: Industrial Dynamics. MIT Press, Cambridge (1961)
5. Naill, R.F.: A System Dynamics Model for National Energy Policy Planning. System Dynamics Review 8(1), 1–19 (1992)
6. Ahmad, S., Simonovic, S.P.: Dynamic modeling of flood management policies. In: Proceedings of the 18th International Conference of the System Dynamics Society: Sustainability in the Third Millennium, Bergen, Norway, pp. 6–10 (2000)
7. Rodriguez-Ulloa, R., Paucar-Caceres, A.: Soft system dynamics methodology (SSDM): combining soft system methodology (SSM) and system dynamics (SD). Systemic Practice and Action Research 18(3), 303–334 (2005)
8. Duran, Encalada, J., Paucar-Caceres, A.: System Dynamics Urban Sustainability Model for Puerto Aura in Puebla, Mexico. System Practice and Action Research 22(2), 77–99 (2009)
9. Fong, W.K., Matsumoto, H., Lun, Y.F.: Application of System Dynamics model as decision making tool in urban planning process toward stabilizing carbon dioxide emissions from cities. Building and Environment 44(7), 1528–1537 (2009)

A Formal Approach to Analysing Knowledge Transfer Processes in Developing Countries

Jin Tong, Siraj A. Shaikh, and Anne E. James

Department of Computing, Faculty of Engineering and Computing,
Coventry University, Coventry, CV1 5FB, United Kingdom
{tongj3,s.shaikh,a.james}@coventry.ac.uk

Abstract. An organisation's competitive advantage depends on its ability to transfer knowledge effectively. Research suggests that knowledge transfer (KT) remains a problem for many organisations, particularly those in developing countries with emerging knowledge economies. Analysis of organisational KT problems in this context could be helpful in diagnosing particular challenges for sustainable development. In this paper, we apply a formal modelling approach to represent the KT processes within organisations. *Communicating Sequential Processes* (*CSP*) are used to model human interactions during the transfer and model-checking techniques are used to analyse obstacles to effective KT. The application of *CSP* in analysing KT models in such a way is a novel idea. To demonstrate our approach we present a case study of a Chinese company and show how cultural attitudes in organisations, widely prevalent in some developing countries, could lead to problems in effective KT.

Keywords: Knowledge transfer, Formal modelling, Communicating Sequential Processes (*CSP*), Model-checking, Emerging economies.

1 Introduction

It is widely agreed that knowledge transfer (KT) contributes to organisational performance [3]. Although increasingly organisations are keen to improve their KT ability, it remains a challenge for many of them, particularly in the developing world. Current research of knowledge management (KM) is mostly based on experience in developed countries that are already becoming knowledge economies [5]. Applications of their KM models and frameworks might not yield the expected results in developing countries however. It is therefore imperative to help organisations in developing countries to understand the issues of KM in their local context. Towards this goal, this paper uses a case study of current KM practice in China through a recently created Chinese mobile phone company (referred to as *Lotus*) [9] to model the KT processes. While studied here in the Chinese context, this model is more applicable in the wider developing countries context.

Tong and Ayres present a model [8] to analyse low-level details of KT processes. The model allows specific transactions that take place in the process of

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 486–501, 2011.

transferring knowledge to be analysed. We strengthen this model by introducing a formal approach to represent the KT processes. The process algebra Communicating Sequential Processes (*CSP*) [4,6] is used to model interactions that take place between various actors in KT; earlier work has successfully applied *CSP* [10].

The rest of the paper is organised as follows. Section 2 reviews the context of this research. Section 3 focuses on describing the transfer model formally defined using *CSP* building blocks. An initial model analysis is then reported in Section 4. In this section, we also demonstrate the diagnosis value of this model using a case study in a Chinese company. The paper concludes with an observation of the implication of this formal model in Section 5.

2 Related Work

We review current understanding of KT in the related literature in Section 2.1. The choice of CSP in our study is briefly justified in Section 2.2.

2.1 Current Understanding of Knowledge Transfer

Effective knowledge transfer strategies have gained attentions from organisations from both developed and developing countries. The available literature on this subject often focuses on two aspects. One perspective is looking at various strategies or mechanisms that can facilitate and accelerate KT, such as setting communities of practice and implementing knowledge maps. Another perspective is focusing on key factors affecting peoples decisions and behaviours in the transfer process, such as trust and cultural issues. However, these discussions are loosely associated and often decoupled from the context of transfer processes. Consequently, very few of these existing studies can be used directly to diagnose any transfer problems and identify appropriate strategies in practice. This situation could be improved if we relate these studies to a low-level transfer model.

Tong and Ayres' model [8] is one of few available frameworks investigating KT at a lower level and view it as the overall process by which knowledge is transferred between people. This model provides sufficient details of the transfer process and is used directly as the basis to investigate the formal modelling approach presented in this paper. Their model is presented as Figure 1 below. In this model, KT involves a sequence of specific steps (actions taken and decisions made) and transactions (interactions between people or between people and knowledge repositories). An individual can take more than one role at the same time in different transfer processes. Detailed steps and transactions followed by each role are described in this model. These roles are presented individually rather than in an integrative framework because the model is designed to represent a variety of KT processes and these individual roles are used as the basic structural elements to form dialogues between different parties involved in the transfer process.

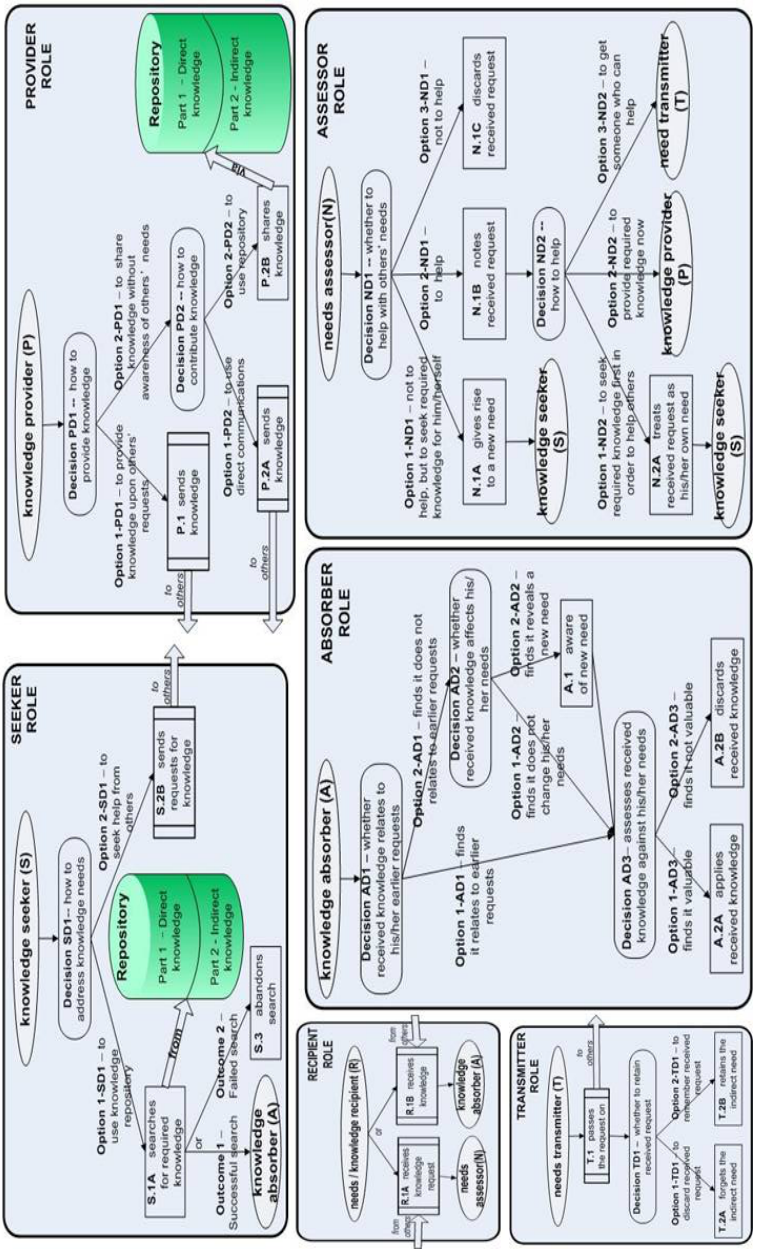


Fig. 1. Tong and Ayres low-level KT model (*Adapted from* [8]*: p.171*)

2.2 A Formal Approach for Knowledge Transfer

Although Tong and Ayres' model [8] has several benefits, their model presentation is very informal. Their model is presented in a graphical form and a potential problem is that people may misinterpret the graphical sequence of the captured transfer details of the model.

A KT process is normally complicated and involves interactions between several parties. These parties often execute in parallel and the process complexity arises from the combinations of ways in which they choose to behave. The concurrency theory in process algebras provides a way of understanding and thereby representing the dynamics and complexity in the process of KT. The choice of *CSP* is therefore very suitable. It has already been used to model and analyse human-machine interactions [1,2,7] very effectively.

The level of analysis for concurrency required for modelling KT processes is well supported in *CSP* in terms of nondeterminism, communication, recursion, abstraction, divergence and deadlock. Communications of events can be modelled sequentially and concurrently along with introducing choice, composition and synchronisation. *CSP* also provides a mature framework for analysis including model checking, which allows us to check for refinement, deadlocks, livelocks and determinism, all of which are relevant here. The tool support for *CSP* in terms of *ProBE* and *FDR* is also helpful.

3 A Formal Model for Knowledge Transfer

We introduce the basic building blocks of *CSP* describing the notation and features of the language relevant to our usage in Section 3.1. In section 3.2, we set out several simple criteria for formalising Tong and Ayres' model [8]. Each of the individual tranfer processes from their model is then formalised using *CSP* in Sections 3.3-3.8. Finally, Section 3.9 presents the entire KT model showing critical synchronisations between individual processes.

3.1 CSP

A *CSP* system is modelled in terms of *processes* and *events* that these processes can perform, where events may be atomic in structure or may consist of distinct components. The *CSP* expression $a \rightarrow P$ describes a process that initially performs event a and then behaves as process P. An external choice operator $\Box$ provides the option of running either of the two processes P or Q when put together as $(P \Box Q)$

where the choice between these two processes is determined by the first event that is performed, which can be chosen by the environment. The parallel operator $[\![A]\!]$ is used to force P and Q to run in parallel and synchronise on events in the set of events A, whereas any of their events that are not in A are performed independently. This is written as $(P \,[\![\, A \,]\!]\, Q)$

We briefly introduce the failures model in *CSP*, which is used for the purpose of analysis in this paper. The failures model allows us to reason about events that a process is ready to perform. It is not possible to judge whether a certain event will always be performed by a process as its environment may not allow it to do so. The approach taken in this model is to reason about processes in terms of events that they are not able to (or fail to) perform. A failure (tr,X) of a process P is the set of all events X which P would refuse after performing the events in the sequence tr. The set of all possible failures of P is written as $failures\,[\![P]\!]$. For example, for $a \rightarrow P$ there are two possibilities. First, if a has not occurred then it has performed an empty trace $\langle\rangle$ and is able to refuse any event other than a. Second, event a has occurred in which case the rest of the failures are those of P. More formally,

$$\begin{aligned} failures\,[\![a \rightarrow P]\!] = &\{(\langle\rangle, X) \mid a \notin X\} \\ &\cup \{(\langle a\rangle \frown tr, X) \mid (tr, X) \in failures\,[\![P]\!]\} \end{aligned}$$

where $x \frown y$ denotes appending x with y. P is said to be failure refined by Q which is written as $P \sqsubseteq_F Q$ if all failures of P are also all the failures of Q

$$failures\,[\![Q]\!] \subseteq failures\,[\![P]\!]$$

3.2 Formalising Interactions

In Tong and Ayres' model [8], transfer roles are defined according to the nature of the actions taken by people but not the sequential logic of peoples behaviours. Since the purpose of introducing *CSP* in this research is to capture detailed sequences of peoples behaviours, the original transfer roles need to be redefined in order to have a consistency with the *CSP* notations. This does not change the captured details in the original model, as it is just represented from a different perspective. People's transfer actions are treated as *events*. Individual transfer roles in the model are seen as separate *processes*. Similar to a *CSP* system formed by processes and events, KT in the model needs to be seen as a system. Six separate processes are defined in our formal KT model, including Knowledge Seeker, Knowledge Recipient, Knowledge Provider, Needs Recipient, Needs Transmitter, and Knowledge Repository.

3.3 Knowledge Seeker Process

Process *SEEKER* is activated when a person starts to search for required knowledge (*SEEK*). In subprocess *SEEK*, a seeker can either use a *REPOSITORY*, or request knowledge from other people. If he succeeds in searching the repository and discovers new knowledge, he then becomes a knowledge recipient (*K_RECIPIENT*). But if he does not find anything, he then returns to the starting point of the entire process and be ready for another seeker attempt.

On the other hand, if the seeker had decided to send request to others for help, he would interact with a *N_Recipient* and wait for responses.

SEEKER can also be a consequent process triggered by other processes. For example, a needs recipient (*N_RECIPIENT*) becomes a *SEEKER* when a new knowledge need is raised (*r_raised* → *SEEK*).

Knowledge Seeker

process
$SEEKER = SEEK \ \square \ n\_raised \rightarrow SEEK$
process
$SEEK = search\_repository \rightarrow$
$(s\_succeed \rightarrow k\_retrieved \rightarrow SEEKER$
$\square \ s\_failed \rightarrow SEEKER)$
$\square \ r\_sent \rightarrow (r\_responded \rightarrow k\_provided \rightarrow SEEKER$
$\square \ SEEKER)$

3.4 Knowledge Recipient Process

A person becomes a *k_recipient* if he previously acted as a seeker and succeeded in seeking knowledge through a repository (*k_retrieved*). Then in *ABSORB* he has to assess the received knowledge and decide how to deal with it. If it corresponds to his earlier request, he will assess its quality and choose either to apply or discard it. If he finds it irrelevant to his earlier request, he then has to check if his knowledge needs remain the same or the received knowledge

Knowledge Recipient

process
$K\_RECIPIENT = k\_retrieved \rightarrow ABSORB$
$\square \ k\_provided \rightarrow ABSORB$
$\square \ k\_offered \rightarrow ABSORB$
process
$ABSORB = k\_relevant \rightarrow (k\_applied \rightarrow K\_RECIPIENT$
$\square \ k\_discarded \rightarrow K\_RECIPIENT)$
$\square \ k\_irrelevant \rightarrow (same\_need$
$\rightarrow (k\_applied$
$\rightarrow K\_RECIPIENT$
$\square \ k\_discarded$
$\rightarrow K\_RECIPIENT)$
$\square \ new\_need$
$\rightarrow (k\_applied$
$\rightarrow K\_RECIPIENT$
$\square \ k\_discarded$
$\rightarrow K\_RECIPIENT)$
$)$

has helped reveal a new need. In either of the these two cases, he then decides if he will apply the received knowledge or discard it.

He can also become a k_recipient if a provider (*PROVIDER*) has provided him knowledge either upon his earlier request (*k_provided* → *ABSORB*) or without him asking for it (*k_offered* → *ABSORB*).

3.5 Knowledge Provider Process

A person may provide knowledge upon others' requests (*provide_upon_request* → *PROVIDER*). He can also choose to share his knowledge without people asking for it in two ways, either providing it to them directly (*k_offered* → *PROVIDER*) or contributing to knowledge repositories where people can access when they need (*add_to_repository* → *PROVIDER*).

Knowledge Provider

process

$$PROVIDER = k\_offered \rightarrow PROVIDER \\ \Box\ provide\_upon\_request \rightarrow PROVIDER \\ \Box\ add\_to\_repository \rightarrow PROVIDER$$

3.6 Needs Recipient Process

A person becomes a needs recipient once received a knowledge request from either a seeker (*r_sent* → *ASSESS*) or a transmitter (*n_transferred* → *ASSESS*). He then has to *ASSESS* the request and decide how to respond. If he does not have the requested knowledge and receiving this request helps him raise a new knowledge need (*r_raised*), he may start to search for knowledge for himself. On the other hand, if he decides to help the knowledge requester, he can either reply to this person by acting as a *PROVIDER* or pass on the request to someone else more capable as *TRANSMITTER*. Otherwise, he may just ignore the request.

Needs Recipient

process

$$N\_RECIPIENT = r\_sent \rightarrow ASSESS \\ \Box\ n\_transferred \rightarrow ASSESS \\ \Box\ r\_retained \rightarrow ASSESS$$

process

$$ASSESS = n\_raised \rightarrow N\_RECIPIENT \\ \Box \\ r\_noted \rightarrow (r\_responded \rightarrow provide\_upon\_request \\ \rightarrow N\_RECIPIENT \\ \Box\ passon\_request \rightarrow n\_transferred \\ \rightarrow N\_RECIPIENT) \\ \Box\ r\_ignored \rightarrow N\_RECIPIENT$$

A person also becomes a *n_recipient* if he acted as a needs transmitter previously and decided to retain the request, so that he can reassess the same request and help the original seeker again.

3.7 Needs Transmitter Process

The process *TRANSMITTER* is activated when a n_recipient decides to pass on the knowledge request (that was received earlier) to another person. Then the transmitter can either discard the request or retain it. If he retains the request, he may prefer to reassess it later and decide whether or not to provide further help to the requester ($r\_retained \rightarrow ASSESS$). Otherwise if he chooses to discard the request, he then returns to the starting point of the process.

Needs Transmitter

process

$$TRANSMITTER = passon\_request \rightarrow n\_transferred \rightarrow (r\_discarded \rightarrow TRANSMITTER \;\square\; r\_retained \rightarrow TRANSMITTER)$$

3.8 Knowledge Repository Process

Without exploring too many details of how a knowledge repository operates, we only define the events representing the interactions between *REPOSITORY* and other processes including *SEEKER* (*s_succeed*) and *PROVIDER* (*add_to_repository*). We treat all operations within a repository as internal events that are not visible to external parts of the system.

Knowledge Repository

process

$$REPOSITORY = s\_succeed \rightarrow REPOSITORY \;\square\; add\_to\_repository \rightarrow REPOSITORY$$

3.9 Formal KT Model

The above six processes are executed in parallel. Several events allow these processes to relate to others and also to form a system representing the formal KT model. This system consists of several sub-systems. *NESYS* represents a knowledge needs exchange system emphasising the synchronised events (*passon_request*, *n_transferred* and *n_retained*) between *TRANSMITTER* and *N_RECIPIENT*. *KSSYS* is a knowledge supply system where *PROVIDER* synchronises with *REPOSITORY* on the event *add_to_repository*. This system captures behaviours of providers adding knowledge into repository. *KPSYS* allows systems *NESYS* and *KSSYS* to be synchronised on event *provide_upon_request*. *KRSYS* represents a knowledge retrieval system, which captures people's behaviours in seeking / retrieving knowledge from others. Within this system,

SEEKER synchronises with *K_RECIPIENT* on *k_retrieved* and *k_provided*. *KTMODEL* is formed when *KPSYS* is in parallel with *NESYS*. They need to synchronise on events *r_sent*, *n_raised* and *r_responded*, *s_succeed*, and *s_failed*. This can be viewed as the overall KT system *KTMODEL*.

Knowledge Transfer Model

process

$NESYS = N\_RECIPIENT \,[\![\, \{passon\_request, n\_transferred, n\_retained\} \,]\!]\, TRANSMITTEER$

$KSSYS = REPOSITORY \,[\![\, \{add\_to\_repository\} \,]\!]\, PROVIDER$

$KPSYS = KSSYS \,[\![\, \{provided\_upon\_request\} \,]\!]\, NESYS$

$KRSYS = K\_RECIPIENT \,[\![\, \{k\_retrieved, k\_provided\} \,]\!]\, SEEKER$

$KTMODEL = KPSYS \,[\![\, \{r\_responded, s\_succeed, k\_offered, r\_sent, n\_raised\} \,]\!]\, KRSYS$

4 Analysing the Knoweldge Transfer Model

The model introduced in this paper captures people's behaviours in a general KT situation. It can be seen as an ideal KT system. In reality, KT systems vary and often have deficiencies in many organisations, pariticularly those in developing countries [9]. Their transfer problems can be diagnosed and represented through model refinement checks against our KT model. We can use the model-checker *FDR* to analyse our KT model and specify the desired property of the system using a simple *CSP* process. Once this simple *CSP* specification process is failure refined by the *KTMODEL*, we can use it to check KT systems in different situations, particularly those with transfer problems.

The property we are concerned with are people's options when they decide to participate in a KT and allow the KT system to be activated. We specify the *SPEC* process to model a person's available choices in this situation. He can choose to perform one of the four events *r_sent*, *search_repository*, *k_offered* or *add_to_repository*.

Specification

process

$SPEC = r\_sent \rightarrow SPEC \;\Box\; search\_repository \rightarrow SPEC \;\Box\; k\_offered \rightarrow SPEC \;\Box\; add\_to\_repository \rightarrow SPEC$

Checking a seeker and a provider's transfer options

assert

$SPEC \sqsubseteq_F KTMODEL \setminus \{s\_succeed, s\_failed, provide\_upon\_request, k\_retrieved, k\_provided, k\_relevant, k\_irrelevant, same\_need, new\_need, k\_applied, k\_discarded, passon\_request, n\_raised, r\_noted, r\_ignored, r\_responded, n\_transferred, r\_discarded, r\_retained\}$

The *FDR* tool allows us to check whether a system always provides the use with a choice of the four inital events. A refinement check on the respective transfer system can be performed using *FDR* with respect to *SPEC*. Events in a system process that do not appear in *SPEC* are explicitly hidden allowing the model_checker to observe only events common to both processes.

The system *KTMODEL* satisfies the refinement check. This means that every time a person is given a choice of event among *r_sent*, *search_reposiotory*, *k_offered*, and *add_to_repository*. Their action here activates the whole KT system. Formally, the *KTMODEL* never refuses any of the above four events. Thus *SPEC* is failure refined by *KTMODEL*, where

$$failures\,[\![KTMODEL]\!] \subseteq failures\,[\![SPEC]\!]$$

4.1 A Case Study at Lotus

We now use three transfer problems observed in a Chinese company named *Lotus* to demonstrate the diagnosis value of our KT model. *Lotus* is a recently created mobile phone manufacturing company. Established by a group of 15 active professionals from the mobile phone industry, who decided to come together in 2005 to form *Lotus*, it designs and manufactures tailor-made mobile phones and other wireless terminal products for markets in China, South America and Europe. Their clients are brand manufactures, mobile phone distributors and small-medium sized wireless product operators. *Lotus* is very representative of a typical small organisation in China and KM practices observed in this company reflect on the wider sector in the county. Due to several cultural barriers (such as fear of loosing face, a sense of modesty, hierarchy consciousness, competitiveness and a preference for face-to-face communication) unveiled within *Lotus*, KT was not as effective as expected.

First, some people were too shy to request knowledge from others directly in several occasions. For instance, some senior *Lotus* employees were too embarrassed to ask for help from juniors, potentially restricting their knowledge seeking because of the fear of loosing face. A shy knowledge seeker normally prefer to use external knowledge repositories to search for answers, such as using a web-based search engine. In some cases, external repositories were the only sources of knowledge. A shy knowledge seeker often failed to access the required knowledge because of their limited resources. Their work efficiency could also be seriously affected as repository search is time consuming and, at times, the same piece of knowledge could be accessed more easily if they chose to request it from others directly. We describe a shy knowledge seeker accordingly.

A Shy Knowledge Seeker

process

$S\_SEEKER = search\_repository \rightarrow$
$(s\_succeed \rightarrow k\_retrieved \rightarrow S\_SEEKER$
$\Box\ s\_failed \rightarrow S\_SEEKER)$

Secondly, some people were unwilling to offer others knowledge if it was not requested by them first. Some *Lotus* employees never offer their knowledge to others spontaneously. They only share what they know when such knowledge is requested by others or as part of their job responsibilities (i.e. mentoring a new employee). A strong sense of competitiveness was not the only cause of this kind of behaviours. A sense of modesty and the hierarchy consciousness were also the contributors. An unwilling knowledge provider can critically slow down the KT processes within an organiation, as knowledge is only flowed after work problems have already arisen (normally when people are already in needs of certain knowledge). Since an unwilling provider does not offer knowledge to others directly or contribute to the knoweldge repository by themselves, it can be defined as:

An unwilling Knowledge Provider

process
$U\_PROVIDER = provide\_upon\_request \rightarrow U\_PROVIDER$

Third, it was also discovered that people prefer to keep their knowledge implicit and share it informally. Many *Lotus* employees believed that through face-to-face communications you are showing more respect to people who actually shared knowledge with you, so that you are building a trustworthy relationship with them and your future requests for help are more likely to be responded. Because of people's preference for face-to-face communications, the effort of establishing an organisational knowledge repository within *Lotus* has failed. The KT system in this situation does not include a knowledge repository. Without a repository available, both knowledge seekers and providers' transfer options are restricted. Thus this transfer system is certainly less effective.

A Seeker and a Provider without a Repository

process
$SEEKER\_R = r\_sent \rightarrow$
$(r\_responded \rightarrow k\_provided \rightarrow SEEKER\_R$
$\Box\ SEEKER\_R)$
$PROVIDER\_R = provide\_upon\_request \rightarrow PROVIDER\_R$
$\Box\ k\_offered \rightarrow PROVIDER\_R$

Since process *SPEC* is failure refined by *KTMODEL*, we can use it to check the transfer system in the above three conditions. Thus the problematic part of the transfer systems within *Lotus* can be demonstrated formally.

4.2 A KT System with a Shy Seeker

The KT system with a shy knowledge seeker can be defined as below.

KT Model with a Shy Knowledge Seeker

process

$NESYS = N\_RECIPIENT \,[\![\, \{passon\_request, n\_transferred, n\_retained\} \,]\!]\, TRANSMITTEER$

$KSSYS = REPOSITORY \,[\![\, \{add\_to\_repository\} \,]\!]\, PROVIDER$

$KPSYS = KSSYS \,[\![\, \{provided\_upon\_request\} \,]\!]\, NESYS$

$FKRSYS = K\_RECIPIENT \,[\![\, \{k\_retrieved, k\_provided\} \,]\!]\, S\_SEEKER$

$FKTMODEL1 = KPSYS \,[\![\, \{r\_responded, s\_succeed, k\_offered, r\_sent, n\_raised\} \,]\!]\, FKRSYS$

Now we check this system with the key property *SPEC* in our general KT model.

Checking a shy knowledge seeker's options

assert

$SPEC \sqsubseteq_F FKTMODEL1 \setminus \{s\_succeed, s\_failed, r\_responded, k\_retrieved, k\_provided, k\_relevant, k\_irrelevant, same\_need, new\_need, k\_applied, k\_discarded, n\_raised, r\_noted, r\_ignored, n\_transferred, provide\_upon\_request, passon\_request, r\_discarded, r\_retained\}$

The system *FKTMODEL*1 does not satisfy the refinement check. The *FDR* tool demonstrates that the process *FKTMODEL*1 allows the possibility of the events *k_offered*, *add_to_repository* and *search_reposiotory*, but refuses the event *r_sent*. More formally,

$$(\langle\rangle, \{r\_sent\}) \in failures\,[\![FKYMODEL1]\!]$$

whereas

$$(\langle\rangle, \{r\_sent\}) \notin failures\,[\![SPEC]\!]$$

hence *SPEC* is not failure refined by *FKTMODEL*1

$$failures\,[\![FKTMODEL1]\!] \not\subseteq failures\,[\![SPEC]\!]$$

4.3 A KT System with an Unwilling Provider

We analyse the system *FKTMODEL2* where there is an unwilling knowledge provider as described earlier. This system can be defined as follows.

KT Model with an unwilling Knowledge Provider

process

$NESYS = N\_RECIPIENT \,[\![\, \{passon\_request, n\_transferred, n\_retained\} \,]\!]\, TRANSMITTEER$

$FKSSYS = REPOSITORY \,[\![\, \{add\_to\_repository\} \,]\!]\, U\_PROVIDER$

$KPSYS = FKSSYS \,[\![\, \{provided\_upon\_request\} \,]\!]\, NESYS$

$KRSYS = K\_RECIPIENT \,[\![\, \{k\_retrieved, k\_provided\} \,]\!]\, SEEKER$

$FKTMODEL2 = KPSYS \,[\![\, \{r\_responded, s\_succeed, k\_offered, r\_sent, n\_raised\} \,]\!]\, KRSYS$

Now we check this system with the process *SPEC*.

Checking an unwilling knowledge provider′s options

assert

$SPEC \sqsubseteq_F FKTMODEL2 \setminus \{s\_succeed, s\_failed, r\_responded, k\_retrieved, k\_provided, k\_relevant, k\_irrelevant, same\_need, new\_need, k\_applied, k\_discarded, n\_raised, r\_noted, r\_ignored, n\_transferred, provide\_upon\_request, passon\_request, r\_discarded, r\_retained\}$

The system *FKTMODEL2* does not satisfy the refinement check. The *FDR* tool demonstrates that the process *FKTMODEL2* allows the possibility of the events *r_sent* and *search_reposiotory*, but refuses the events *add_to_repository* and *k_offered*. More formally,

$$(\langle\rangle, \{add\_to\_repository, k\_offered\}) \in failures\,[\![FKYMODEL2]\!]$$

whereas

$$(\langle\rangle, \{add\_to\_repository, k\_offered\}) \notin failures\,[\![SPEC]\!]$$

hence *SPEC* is not failure refined by *FKTMODEL2*

$$failures\,[\![FKTMODEL2]\!] \not\subseteq failures\,[\![SPEC]\!]$$

4.4 A KT System without a Repository

A KT system without a knowledge repository is defined as follows.

KT Model without a Knowledge Repository

process

$NESYS = N\_RECIPIENT \,[\![\, \{passon\_request, n\_transferred, n\_retained\} \,]\!]\, TRANSMITTEER$

$FKPSYS = PROVIDER\_R \,[\![\, \{provided\_upon\_request\} \,]\!]\, NESYS$

$FKRSYS = K\_RECIPIENT \,[\![\, \{k\_provided\} \,]\!]\, SEEKER\_R$

$FKTMODEL3 = FKPSYS \,[\![\, \{r\_responded, k\_offered, r\_sent, n\_raised\} \,]\!]\, FKRSYS$

Now we check this system with the process *SPEC*.

Checking people's options when the repository is unavailable

assert

$SPEC \sqsubseteq_F$ FKTMODEL3 $\setminus \{s\_succeed, s\_failed, r\_responded, k\_retrieved, k\_provided, k\_relevant, k\_irrelevant, same\_need, new\_need, k\_applied, k\_discarded, n\_raised, r\_noted, r\_ignored, n\_transferred, provide\_upon\_request, passon\_request, r\_discarded, r\_retained\}$

The system *FKTMODEL3* does not satisfy the refinement check. The *FDR* tool demonstrates that the process *FKTMODEL3* allows the possibility of the events *k_offered* and *r_sent*, but refuses to perform the events *add_to_repository* and *search_reposiotory*. More formally,

$$(\langle\rangle, \{add\_to\_repository, search\_reposiotory\}) \in failures\,[\![FKYMODEL3]\!]$$

whereas

$$(\langle\rangle, \{add\_to\_repository, search\_reposiotory\}) \notin failures\,[\![SPEC]\!]$$

hence *SPEC* is not failure refined by *FKTMODEL3*

$$failures\,[\![FKTMODEL3]\!] \not\subseteq failures\,[\![SPEC]\!]$$

4.5 Analysing Transfer Deadlocks and Livelocks

We also use *FDR* to check the system for deadlocks and livelocks. Deadlocks arise when certain processes within a system are awaiting an interaction with other processes before they can continue their own events. This means that some of their events will never take place if related processes are not activated. Such a situation is undesirable as it ultimately halts the execution of a system. Livelocks arises when processes descend into an endless sequence of interaction among themselves, excluding any other processes and the external environment. This is particularly undesirable as it means the system gets into an endless cycle of execution with no further progress and possibly an unnecessary consumption

of resources. Both problems arise not due to the design of individual processes but due to the way they are combined [6].

The KT model introduced in this paper has been successfully checked for deadlock and livelock freedom. This means all processes can be activated in our model and people's KT attempts can always have definite results. This reflects on the feasibility of such a system where all processes will complete no matter what knowledge search results these processes bring (be it successful or failed). However in reality a knowledge seeker may send a knowledge request and never get a response. This could be a deadlock in the transfer system. Or as we described in the *Lotus* case, a livelock problem could appear when a shy knoweldge seeker and an unwilling knowledge provider both exist in the same knoweldge transfer system. A seeker may use search knowledge repositories as his only seeking option. If his search through repository always fails because the providers never contribute to the repository, he may need to repeat the *SEEKER* process endlessly. We propose to analyse for such conditions further, essentially highlighting a minimal set of true conditions necessary to guarantee progress of knowledge search queries.

5 Conclusion and Future Work

As the major contribution, a formal approach using *CSP* is proposed to analyse the process of KT in this paper. Such an approach provides a framework for a rigorous and detailed analysis of KT processes in organisations. The formal KT model presented here is used in the *Lotus* case study to demonstrate the organisation's transfer disfunctions through *CSP* failures analysis. This model also allows us to diagnose where problems in KT lie within an organisation, so that they could be addressed. We plan to report further on using our formal model as a key property (another *SPEC* process) to identify the problematic parts of KT systems within organisations. Based on a formal analysis using our approach, organisations would be able to identify specific challenges in better business performance, propose corresponding transfer strategies to change organisational behaviours, recommend further training to facilitate a more efficent knoweldge transfer environment. In future studies we plan to demonstrate how to help an organisation overcome its transfer barriers by introducing agent processes to compliment its current KT system. This could particularly benefit organisations in developing countries where KT mechanisms are often ineffective.

The application of a process algebra in analysing KT models is a novel idea. It explores a new direction of studying human knowledge related processes in the field of knowledge management. *CSP* is particularly suited as it allows us to model individual behaviour to meticulous detail, so that specific organisational work patterns can be represented formally. This is particularly useful for organisations in developing countries to speed up necessary organisational changes and enhance their competitiveness. Similar formal analysis using *CSP* can also be applied in studying other knowledge management processes, such as knowledge innovation process and knowledge exploitation process. We hope our effort

serves to inspire new ideas and approaches to the wider knowledge management community.

References

1. Cerone, A., Connelly, S., Lindsay, P.: Formal analysis of human operator behavioural patterns in interactive surveillance systems. Software and System Modeling 7, 273–286 (2008)
2. Cerone, A., Shaikh, S.A.: Formal Analysis of Security in Interactive Systems. In: Handbook of Research on Social and Organizational Liabilities in Information Security, ch. 25, pp. 415–432. Information Science Reference (2008)
3. Davenport, T., Prusak, L.: Working Knowledge: How Organizations Manage What They Know. Harvard Business School Press, Boston (1998)
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs (1986)
5. Okunoye, A.: Towards a framework for sustainable knowledge management in organisations in developing countries. In: Brunnstein, K., Berleur, J. (eds.) Human Choice and Computers: Issues of Choice and Quality of Life in Developing Countries, pp. 225–237. Springer, Heidelberg (2002)
6. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. John Wiley and Sons, Ltd., Chichester (2000)
7. Shaikh, S., Krishnan, P., Cerone, A.: Formal approach to human error recovery. In: The Pre-proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007), pp. 101–135 (2007)
8. Tong, J., Ayres, R.: A low-level model for knowledge transfer. In: Proceeding of the IADIS International Conference on Information Systems, Barcelona, Spain, pp. 169–176 (2009)
9. Tong, J., Shaikh, S.: ICT driven knowledge management in developing countries: A case study in a chinese organisation. In: Pont, A., Pujolle, G., Raghavan, S.V. (eds.) WCITD 2010. IFIP AICT, vol. 327, pp. 60–71. Springer, Heidelberg (2010)
10. Tong, J., Shaikh, S., James, A.: A formal approach to modelling knowledge transfer processes. In: Tome, E. (ed.) Proceedings of the 11th European Conference on Knowledge Management, Famalicao, Portugal, pp. 1012–1021 (2010)

Author Index